An LLM Engine in C and Metal

Gemma 4 on a 16 GB Mac, read from its own source

Ekkachai Thawinan
Independent Researcher, Chiang Mai, Thailand

Draft, October 2026

PREPRINT

PREPRINT

© 2026 Ekkachai Thawinan. This book is licensed under CC BY 4.0
(`https://creativecommons.org/licenses/by/4.0/`); its code under the MIT License.

Draft, October 2026. Not peer reviewed. Gemma 4 is a trademark of its owner; no model weights are included, and the weights keep their own licence terms.

How to use this book

This book has 24 chapters in six Parts, followed by appendices A to F. Choose a reader, builder, or skimmer route. Every experiment carries a level tag: W0 needs no weights, W1 needs only the model file's header, W2 needs the model and runs inference, and W3 also needs the encoder file. W0 and W1 results come from runs on the frozen snapshot. W2 and W3 are experiments for you to run on your own machine: the book says what to inspect and what shape of result to expect, and it quotes earlier dated project measurements only as history.

Use only the frozen files in `code/snapshot/`. Build artifacts belong in `code/build/`. Weights are not included.

PREPRINT

Contents

I GROUND 1

1 The map: what an engine does 2

2 Words to numbers: the tokenizer 14

3 The model file: GGUF and mmap 28

4 Small numbers: fp16 and quantised blocks 41

5 The plain-C reference 57

II METAL 69

6 Metal for C programmers 70

7 Decode kernels: matrix × vector 85

8 Prefill kernels: matrix × matrix 102

9 Small steps and fusing them 118

III THE MODEL 127

10 Attention, step 1: queries, keys, RoPE 128

11 The KV cache 141

12 Attention, step 2: the fast kernels 158

13 The feed-forward block and per-layer embeddings 180

14 One layer, then the whole model 193

15 Choosing the next token 206
The question . . . 206
15.1 A vocabulary-sized choice . . . 206
15.1.1 What does “largest” mean when values tie? . . . 206
15.2 The reduction, first on the CPU . . . 207
15.2.1 Why keep the ID beside each partial score? . . . 207
15.2.2 How does each lane cover the vocabulary? . . . 208
15.3 The next step is already in flight . . . 209
15.3.1 A concrete pass through 262,144 candidates . . . 211

PREPRINT

Draft book, October 2026. Not peer reviewed.

15.3.2 What does the benchmark compare? . . . 211
15.3.3 When does a selected token become visible text? . . . 213
15.3.4 A real dependency: per-layer embeddings . . . 213
15.4 What random sampling would change . . . 214
15.5 A rare race, found by a distribution test . . . 215
15.6 Experiment: inspect the token choice without weights . . . 215
15.7 Exercises . . . 216
15.8 Rebuild checkpoint m6b . . . 216

IV CHAT 217

16 The chat loop, templates and the cache prefix 218

V EYES AND EARS (optional; needs the encoder file; 56 pp.) 236

17 Seeing: the vision encoder 237

18 Hearing: the audio encoder 256

19 Media in the conversation 276

VI RIGHT AND FAST 290

20 Is it right? 291

21 Measuring speed on a shared machine 308

22 Making it fast: the story 322

23 Memory 344

24 Packaging: from engine to tool 353

A Weight formats in full 363

Appendices 363

B Command line and environment reference 369

C The 26B-A4B: what is different 375

D Rebuild milestones 379

E Glossary 385

F Exercise hints 389
F.1 Chapter 1 . . . 389
F.2 Chapter 2 . . . 389
F.3 Chapter 3 . . . 390
F.4 Chapter 4 . . . 390
F.5 Chapter 5 . . . 390

PREPRINT

Draft book, October 2026. Not peer reviewed.

F.6 Chapter 6 . . . 391
F.7 Chapter 7 . . . 391
F.8 Chapter 8 . . . 392
F.9 Chapter 9 . . . 392
F.10 Chapter 10 . . . 393
F.11 Chapter 11 . . . 393
F.12 Chapter 12 . . . 393
F.13 Chapter 13 . . . 394
F.14 Chapter 14 . . . 394
F.15 Chapter 15 . . . 394
F.16 Chapter 16 . . . 395
F.17 Chapter 17 . . . 395
F.18 Chapter 18 . . . 396
F.19 Chapter 19 . . . 396
F.20 Chapter 20 . . . 397
F.21 Chapter 21 . . . 397
F.22 Chapter 22 . . . 397
F.23 Chapter 23 . . . 398
F.24 Chapter 24 . . . 398

PREPRINT

Part I

GROUND

PREPRINT

Chapter 1

The map: what an engine does

The question

What happens between a model file on disk and a word on the screen, and what makes producing one more token take time?

The idea

A language model takes a list of numbers and returns another list: a score for every possible next token. The engine is the program that makes this function useful on a real machine. It locates the weights, turns text into token IDs, runs the layers, chooses a token, turns it back into text, and repeats. The diagram shows the route for text generation; image and audio inputs join at the model-input stage and are covered in Part V.

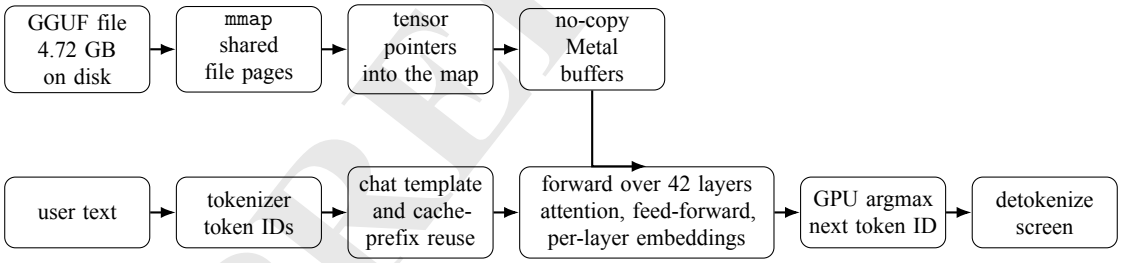


Figure 1.1: The text-generation path. The upper route prepares weights; the lower route carries a conversation through the model.

Why does decode take time even though it produces only one token? Nearly every layer needs its weights for that token. The README estimates 2.61 GB read per decode step for IQ4_XS and 2.77 GB for Q4_0 (README, line 159). At an illustrative sustained 100 GB/s, moving 2.7 GB alone takes at least 0.027 seconds, or roughly 37 tokens per second. That is a ceiling estimate, not a prediction: attention, the output head, dispatch gaps, and memory traffic all matter. Prefill is different. It processes many prompt tokens together, so a weight tile can contribute to many output rows before it is read again.

Gemma 4 E4B has 42 layers. Twenty-four layers own key/value (KV) cache entries; the later 18 reuse earlier layers' keys and values. Per-layer embeddings (PLE) add a small learned contribution at each layer. Those are architectural facts that change what the engine stores and computes; this map names them so later chapters can explain their mechanics.

PREPRINT

The real code

How does the route divide across the two large source files? `gemma4.c` contains model parsing, CPU reference work, scheduling, and the application flow. `gemma4_metal.m` contains the embedded GPU kernels and their Objective-C host. The following map follows PLAN.md's inventory and is checked against the frozen snapshot, whose files are the source of every listing in this book.

Where to look in `gemma4.c`

The line ranges below are useful landmarks, not separate programs. The early file builds the vocabulary of data structures and operations used later; the later file orchestrates them.

Lines	What lives there
1–92	Header, build notes, types and constants.
93–155	Worker threads and the parallel task pool.
156–367	GGUF metadata, tensor records, mmap reader and lookup.
368–613	Float formats and quantized-weight decoding.
614–849	GPU dispatch wrappers and CPU matrix operations.
850–1228	Model and layer structures, loading, activations, KV allocation.
1229–1512	Attention, feed-forward blocks and supporting operations.
1513–1922	Optional expert-weight cache for the larger model.
1923–2123	Per-layer embeddings and the language-model forward pass.
2124–3423	Optional image/audio encoders and media handling.
3424–3684	Vocabulary, BPE tokenizer and detokenizer.
3685–3833	Perplexity and KL-divergence tools.
3834–4513	Generation, benchmark and chat loop.
4514–4881	Weight mapping, usage text, option parsing and `main`.

Where to look in `gemma4_metal.m`

Why can this project compile GPU programs without an Xcode project or prebuilt Metal library? Lines 9–1615 embed Metal Shading Language (MSL) as a C string. The host code later passes that string to the Metal runtime compiler. Within the string, helper and dequantization kernels occupy roughly 9–296, matrix operations 297–750, normalization and smaller fused operations 752–1047, attention 1048–1479, and vision and token-scoring kernels 1480–1615. Lines 1616–2199 are Objective-C host code: device setup, pipeline creation, mapping, command encoding, and the C-callable interface. The header `gemma4_metal.h` connects this host to the C engine.

The embedded source is declared by the C string and MSL stringification macro at lines 7–10. At startup, the host passes that string to the runtime compiler. The excerpt shows the call and its failure check. This keeps the build simple and lets Metal compile specialized pipelines on the target machine at initialization.

```
1772          library = [device newLibraryWithSource:@(kernel_src) options:nil
       error:&err];
1773          if (!library) { fprintf(stderr, "metal: %s\n", err.
      localizedDescription.UTF8String); return 0; }
```

Listing 1.1: `gemma4_metal.m`:1772–1773

What `main` lets you ask

The final section has mapping and usage helpers before `main`, which begins at line 4589. Its option parser turns command-line flags into one of several jobs. A prompt request runs text generation; `--chat` or a system instruction enables the conversation loop; `--bench P N` measures a prefill of P tokens followed by N decode steps; a text file selects scoring/perplexity; and a base-log-probability file enables the KL

FINGERPRINT

comparison. The media options route into the optional image/audio path. All these modes share the model loader and forward pass, but they prepare inputs and report results differently.

This is why the same engine can be studied through several kinds of evidence. A generated answer demonstrates the end-to-end path, but a plausible sentence cannot reveal a small numerical difference. A benchmark reports throughput, but it does not tell whether the output scores match a reference. Perplexity and KL comparisons help with numerical behavior, while source inspection explains why a path took that shape. The book keeps those questions separate: build and command-line checks first, file inspection next, and inference comparisons only in chapters that explicitly have weights available.

The W0 help command exercises only the usage path. The parser prints the supported modes and exits before opening a GGUF. A generation, benchmark, or KL command would require model data and is therefore not part of this pilot's experiment. The exact parser and mode dispatch are mapped at `gemma4.c:4589--4881`; later chapters return to the individual modes when they need them.

1. The GGUF file on disk

What is the several-gigabyte file made of? It begins with a header, metadata key/value pairs, a tensor directory, alignment padding, and then tensor payloads. The directory says each tensor's name, shape, data type, and offset; the tensor bytes are not separate files. For E4B the complete file is about 4.72 GB. The reader starts at `gemma4.c:156`; `gguf_open` at lines 313–359 reads the directory and establishes where each payload begins. It is designed to keep the file's declared layout as the source of truth, instead of hard-coding one fixed tensor order.

2. mmap and tensor pointers

How can the program work with a file this large without reading it into a second allocation? `gguf_open` maps the file and makes tensor records point into those mapped pages. The operating system brings pages in when the CPU or GPU touches them. The loader then resolves tensor names into fields of `Model` and `Layer`; see the model structures and load path at `gemma4.c:850--1097`. A pointer here is an address into the mapping, not a private copy of a matrix. That choice reduces load-time copying and lets the operating system manage file-backed memory.

3. No-copy Metal buffers

If the GPU needs the weights, why not copy them into a separate GPU allocation? Apple silicon shares physical memory between CPU and GPU. `map_model` at `gemma4.c:4518--4557` selects the needed tensor regions and calls the Metal mapping interface; `metal_map` is in `gemma4_metal.m:1726--1735`. The Metal host wraps page-aligned regions of the mapping as no-copy buffers and makes the used weights resident. This avoids a second weights-sized allocation and its copy.

The E4B file is 4.72 GB, while the launch message reports 2.51 GB of weights wired for GPU use (ENGINE-MATH.md §11.1). The difference is intentional: the large per-layer token embedding table is read on the CPU, and stored K/V matrices for layers that share cache entries are not needed by the GPU path. "Wired" means the pages are made resident for GPU access; it does not mean 2.51 GB was copied to a discrete card. The snapshot contains the mapping code; the size is historical inventory data, not measured by this chapter's W0 run.

4. The tokenizer

How does a line of text become numbers the model recognizes? The vocabulary and tokenization implementation sit at `gemma4.c:3424--3684`. `tokenizer_load` reads the vocabulary and merge ranks (line 3481); `bpe_run` repeatedly merges adjacent pieces according to those ranks (line 3595). The tokenizer preserves the model's own vocabulary rules, including the visible space marker and special control tokens. The point of doing this in the engine is exact agreement with the vocabulary shipped in the GGUF: token IDs are the model's input alphabet, not interchangeable word numbers.

PREPRINT

5. The chat template and cache-prefix reuse

Why does a chat prompt contain more than the user's sentence? At `gemma4.c:4143`, `chat_render` writes the role markers, message boundaries, and assistant prefix expected by the model. `chat_run` at lines 4262 onward manages the conversation. When a new rendered conversation begins with tokens already present in the cached prefix, the engine can continue from the longest matching prefix and compute only the suffix. This saves repeated prefill work during a multi-turn chat. The template belongs beside tokenization in the input path because its markers become ordinary model token IDs before forward begins.

6. Forward over 42 layers

What happens after the IDs enter the model? `forward` begins at `gemma4.c:2054`. It gathers token embeddings, normalizes rows, then iterates over all 42 layers. Each layer applies attention, residual and normalization work, and its feed-forward block; when PLE exists, its projection and block are added as well. KV state is updated only where required by the model's sharing scheme. The function finally applies the output head to produce scores, or logits, for candidate next tokens.

The excerpt starts on the first executable statement of the layer-loop section, avoiding a listing-boundary artifact in the earlier cut. It shows the loop and the central operations in source order. The called helpers carry most of the arithmetic; the excerpt is a map of control flow, not a substitute for their later CPU and Metal chapters.

```
2090      int Tl = T, pos = st->n_past;  // the layers' rows, and the position
          of the first
2091      for (int l = 0; l < m->n_layer; l++) {
2092          const Layer *L = &m->layers[l];
2093          const float *next_norm = l + 1 < m->n_layer ? m->layers[l + 1].
          attn_norm : m->output_norm;
2094          if (l == m->n_kv && Tl > n_out && first >= n_out && !a->bidir) {
2095              // The layers from here on write no KV cache, so the rows
          whose logits aren't wanted affect nothing
2096              // any more: only rows [first, first + n_out) go on (in a
          prefill, just the last). (Rows are moved
2097              // down, which needs first >= n_out so that no row is
          overwritten before it is read.)
2098              for (int t = 0; t < n_out; t++) a->rows[t] = first + t;
2099              const int P = m->n_layer * m->n_pl;
2100              Tensor xr = {.type = T_F32, .ne = {E, T, 1, 1}, .data = (
          const uint8_t *)a->x};
2101              Tensor pr = {.type = T_F32, .ne = {P, T, 1, 1}, .data = (
          const uint8_t *)a->pl};
2102              EmbedJob xj = {&xr, a->rows, a->x, 1.0f}, pj = {&pr, a->rows
          , a->pl, 1.0f};
2103              GatherJob gj = {a->xn, a->xn, a->rows, E};
2104              STEP(metal_get_rows(a->x, a->x, E * sizeof(float), T_F32, a
          ->rows, n_out, E, 1.0f), n_out, embed_row, &xj);
2105              STEP(metal_gather(a->xn, a->xn, a->rows, n_out, E), n_out,
          gather_row, &gj);
2106              if (P) STEP(metal_get_rows(a->pl, a->pl, P * sizeof(float),
          T_F32, a->rows, n_out, P, 1.0f), n_out, embed_row, &pj);
2107              Tl = n_out, pos += first, first = 0;
2108          }
2109          attention(m, L, l, a, st, Tl, pos);
2110          add_norm(a->x, a->a, L->post_attn_norm, Tl, E, m->eps, 1.0f, a->
          xn, L->ffn_norm);
2111          ffn(m, L, a, Tl, next_norm);
2112          if (m->n_pl) {
```

```
2113             if (l == 0) pl_inputs(m, a, Tl, prev);  // (the GPU runs
       layer 0 meanwhile)
2114             pl_block(m, L, l, a, Tl, next_norm);
2115         }
2116         gpu_submit();  // start the GPU on this layer while the next is
       encoded
2117     }
2118
2119     matmul(logits, a->xn + (size_t)first * E, m->output, n_out, 0);
```

Listing 1.2: `gemma4.c`:2090–2119

7. GPU argmax

How does a vector with hundreds of thousands of scores become one token? The engine needs only the largest score for greedy decoding, so it need not copy every logit to the CPU. The GPU argmax kernel is in `gemma4_metal.m:1544`. The C decode path invokes it around `gemma4.c:3887--3895`. Keeping selection on the GPU avoids a large readback and returns the chosen integer token ID. Sampling modes add their own selection logic; this chapter's map focuses on the greedy path.

8. Back to text and the screen

Where does the chosen integer become readable language? The detokenizer is at `gemma4.c:3546`; `print_token` is at line 4017 and the generation loop streams the result. A token may be a whole word, a word fragment, a space-prefixed piece, or a special marker, so output is decoded using the same vocabulary conventions as input. The selected token is appended to the sequence, and the next decode step repeats the route through the layers.

How to read the rest of the book

The diagram is also the book's table of contents. Chapter 2 follows text into token IDs; Chapter 3 follows GGUF, mmap, and tensor pointers; Chapter 4 explains the small numeric formats stored in the weights; Chapter 5 builds the CPU reference path. Chapter 6 explains Metal buffers, runtime shader compilation, and dispatch. Chapters 7 and 8 focus on decode matvecs and prefill matrix operations. Chapters 9–15 follow activations, attention, KV cache, and the 42-layer forward path, ending with output selection. Chapters 16–20 cover chat, media, measurement, and correctness. Chapter 21 remeasures the performance and memory figures introduced here; the remaining chapters complete the rebuild and reference material. The exact chapter titles are in OUTLINE.md.

A few reference numbers

What do the published numbers tell us before we measure again? These values are historical observations recorded in the snapshot documentation, not results from the W0 experiment. The timing row is the README's Q4_0 benchmark on an M3 MacBook Air with an 8-core GPU, captured October 3–4, 2026 in two alternating rounds with 30-second pauses. The README says 15 of 16 GB of memory was in use by other apps; it records no idle-gate reading, so treat these as historical results rather than a quiet baseline. The memory and KL entries come from different documented runs and configurations, so their sources and model variants are stated explicitly. Chapter 21 will remeasure them under a recorded quiet-GPU protocol.

Historical measurements: M3 MacBook Air, 2026-10-03/04; remeasure in Chapter 21.

PREPRINT

Measure	Historical value	Context / source
Prefill	440 tok/s	Q4_0, pp512; README, M3 Air, 2026-10-03/04.
Decode	29–30 tok/s	Q4_0, tg128, depth 512; same README runs.
Wired weights	2.51 GB of 4.72 GB	E4B mapping banner / ENGINE-MATH §11.1; no-copy Metal regions.
Memory used	3.08 GB	Q4_0, generated 128 tokens; wired + anonymous + compressed, README Results.
Mean KL divergence	0.000012	E4B IQ4_XS, c=512, four chunks; README model comparison.

The prefill and decode values describe the Q4_0 benchmark file; 3.08 GB is the accompanying Q4_0 system memory measure while generating 128 tokens. The KL figure is recorded for the separate IQ4_XS model comparison. They should not be read as one single run. They are useful orientation points, and Chapter 21 is where we will repeat the measurements and report machine load, idle-gate state, statistic, and timing method.

What the GGUF directory buys us

A useful way to picture the GGUF is as a table of contents followed by a book-sized body of tensor bytes. A metadata record can say that the model has 42 blocks, a particular vocabulary size, a sliding-window length, or a chat template. Each tensor record adds a name, up to four dimensions, a storage type, and a relative byte offset. For a matrix, the dimensions tell the loader how many input values belong to each row; the type tells the decoder how those values are packed. The file does not contain a C structure the program can cast directly. It contains a serialized format, so the reader advances a checked cursor and reads each field in the format's declared width.

The opening of `gguf_open` is a compact example of this approach. It records the file size, maps the whole file read-only and shared, verifies the magic and version, then reads counts before allocating small arrays for metadata and tensor descriptors. The map can cover gigabytes while those descriptor arrays remain comparatively small. A shared mapping matters later: if a GPU buffer wraps those pages, a private copy would defeat the memory-saving design.

Each metadata entry has a key, a type, and a value. The reader keeps the value's address while a type-aware helper advances the cursor; this lets later code ask for settings by name without making assumptions about their order. Tensor shapes are read dimension by dimension, with unused dimensions filled as one so downstream code has a uniform four-slot shape. The offset in each record is relative to the aligned start of tensor data, not the beginning of the file. The reader obtains the alignment from metadata (32 bytes by default), rounds the cursor forward, and then adds each relative offset. Before accepting a pointer, it checks that the tensor's calculated end remains inside the mapped file. A malformed model should fail at load time rather than hand an invalid address to a GPU kernel.

The listing shows the transition from serialized records to usable addresses. The temporary offset array is freed once all tensor pointers have been fixed up. The ‘data‘ fields themselves point into the shared mapping; no payload-sized tensor copy is made.

```
320     g.base = mmap(NULL, g.size, PROT_READ, MAP_SHARED, fd, 0);  //
      shared: wiring pages for the GPU must not copy them
321     close(fd);
322     if (g.base == MAP_FAILED) die("mmap failed: %s", path);
323
324     Cursor c = {g.base, g.base + g.size};
325     if (rd_u32(&c) != 0x46554747) die("%s: not a GGUF file", path);  //
      "GGUF"
326     uint32_t version = rd_u32(&c);
327     if (version < 2) die("%s: GGUF v%u not supported", path, version);
328     g.n_tensors = (int)rd_u64(&c);
```

PREPRINT

```
329     g.n_kv = (int)rd_u64(&c);
330
331     g.kv = xmalloc(g.n_kv * sizeof(KV));
332     for (int i = 0; i < g.n_kv; i++) {
333         g.kv[i].key = rd_str(&c);
334         g.kv[i].type = rd_u32(&c);
335         g.kv[i].val = c.p;
336         skip_value(&c, g.kv[i].type);
337     }
338
339     g.tensors = xmalloc(g.n_tensors * sizeof(Tensor));
340     uint64_t *offs = xmalloc(g.n_tensors * sizeof(uint64_t));
341     for (int i = 0; i < g.n_tensors; i++) {
342         Tensor *t = &g.tensors[i];
343         t->name = rd_str(&c);
344         uint32_t nd = rd_u32(&c);
345         if (nd > 4) die("gguf: tensor with %u dims", nd);
346         for (int d = 0; d < 4; d++) t->ne[d] = d < (int)nd ? (int64_t)
    rd_u64(&c) : 1;
347         t->type = rd_u32(&c);
348         offs[i] = rd_u64(&c);
349     }
350
351     size_t align = (size_t)kv_num(&g, "general.alignment", 32);
352     size_t data_start = ((size_t)(c.p - g.base) + align - 1) / align *
    align;
353     for (int i = 0; i < g.n_tensors; i++) {
354         g.tensors[i].data = g.base + data_start + offs[i];
355         if (data_start + offs[i] + tensor_nbytes(&g.tensors[i]) > g.size
    ) die("gguf: tensor data out of range");
356     }
357     free(offs);
358     return g;
359 }
```

Listing 1.3: `gemma4.c:320–359`

Why the pointer map is also a loader

The tensor directory is generic, but the model needs named roles such as token embeddings, attention projections, and output weights. The model loader matches those names and records pointers in a `Model` and its array of `Layer` structures (`gemma4.c:850--1097`). This separates two jobs: the GGUF reader answers "where are the bytes and what format are they?", while the model loader answers "which tensor performs this operation?" That separation also lets the same reader serve E2B and E4B, whose dimensions and layer metadata differ.

A tensor pointer is not necessarily ready for arithmetic. Quantized tensors need a row-byte calculation, and their elements must be decoded according to the recorded type. The numeric conversion and block decoders are grouped at `gemma4.c:368--613`; Chapter 4 follows a few concrete blocks. Keeping a pointer into the file until a row is needed avoids expanding every 4-bit weight into a large float array at startup. It also means memory traffic is determined by the compact stored representation rather than by a full-precision copy.

Which mapped bytes become GPU buffers?

The model file is not one uniform slab of GPU memory. The mapping routine walks the tensor records, skips parts the GPU does not need, and combines adjacent eligible payloads into runs. Runs must respect

Metal's page-alignment requirements. The host mapping code rounds the relevant address down to a page boundary, remembers the offset from that boundary to the tensor data, and exposes the requested byte range through a no-copy buffer. When several tensors are adjacent, one buffer can serve them all.

Why leave some bytes out? E4B stores a large per-layer token embedding table that the CPU reads as rows; it is not copied into the wired GPU set. The format also contains K/V weight matrices for layers whose cache is shared, although those matrices are not needed for the execution path. These are different reasons for exclusion: one is a CPU-side lookup, the other is a redundant stored path for this architecture. The 2.51 GB wired figure counts the selected model weights made resident for Metal, not the full GGUF and not all process memory. Activations, the KV cache, temporary staging areas, and the operating system's file cache are separate accounting questions.

This design is useful because a discrete GPU normally has its own memory and would need a bulk copy. On Apple silicon the GPU can address unified memory, so a no-copy view can point at the same file-backed physical pages. The operating system can still fault pages in as they are touched; "mapped" does not mean the full file has already been read from storage. "Wired" is the additional residency request for the chosen GPU working set. Chapters 3 and 6 return to the exact pointer and page-alignment mechanics.

From text pieces to model IDs

The tokenizer's vocabulary contains more than whole words. A common word may be one piece, but a less common word can be a sequence of frequent fragments; punctuation, leading spaces, and byte fallback pieces all have their own IDs. This is why counting spaces or words cannot predict the prompt length. The model sees integers indexing its learned embedding table, not characters.

On load, `tokenizer_load` builds a vocabulary lookup and a lookup for merge-pair ranks from the GGUF metadata (`gemma4.c:3481--3525`). For a text run, `bpe_run` starts with UTF-8 character pieces, puts mergeable neighbours into a min-heap by rank, and applies the best available merge repeatedly. When a merge changes the neighbours, new candidates are pushed; stale heap entries are skipped. A heap avoids rescanning the whole run for the best pair after every merge. The implementation falls back to the constituent byte tokens when a piece is not in the vocabulary.

A newline run can be treated specially if that entire run is a vocabulary piece. Otherwise it joins the usual BPE path. The tokenizer does not first split the text into dictionary words. This preserves the exact vocabulary rules expected by this model and is why Chapter 2 checks IDs, including punctuation and non-English text, rather than merely checking that decoding looks readable.

The template is part of the prompt

A chat is not sent as a sequence of alternating plain text strings. The rendered stream contains markers that identify speakers, delimit message bodies, and tell the model whether it should continue or finish a turn. The engine stores those markers in the model vocabulary, so the template renderer turns them into tokens alongside the user's text. The renderer is at `chat_render` (`gemma4.c:4143`); the conversation driver is `chat_run` (starting at line 4262).

For orientation, imagine that the cached token sequence ends with the full first user message and the new rendered sequence starts with that same message followed by a second user turn. The common prefix remains valid: the engine can set its past position to that matched prefix and prefill only the new suffix. In the actual code, entries can refer either to text tokens or to media rows, and image chunks that must run together are kept whole. The cache must also respect the sliding-window state: `state_keep` can shorten a candidate reuse point if the attention history needed there is no longer available. Prefix reuse is therefore a correctness decision as well as a speed trick.

This also explains why "same prompt" has two meanings. The same user-visible text can render differently if roles, turn endings, or system instructions differ. Conversely, two longer rendered prompts can share a substantial initial token sequence and reuse its KV state. Chapter 16 follows the chat protocol and cache-prefix logic in depth; Chapter 10 explains what the KV cache stores.

PREPRINT

What one forward call actually returns

During prefill, a call to `forward` can carry many token rows, denoted by `T` in the C code. During ordinary decode, it carries one row. The entry path fetches embedding rows from the token embedding tensor, or accepts already prepared rows for media, and then normalizes them. The main loop runs attention and feed-forward operations in layer order. GPU work is encoded while CPU code prepares later work; calls are submitted to the GPU as the loop advances.

The loop's boundary reflects a model detail. The first 24 layers write KV entries; later layers that share those entries do not need to retain fresh keys and values. In a multi-token prefill, once the shared-cache layers begin, some rows may no longer be needed for logits. The forward code can retain only requested rows at that boundary, rather than carrying all prompt rows through every later layer. The condition is visible near lines 2094–2108. This is a concrete example of model structure changing the schedule, not just the shape of a tensor.

PLE adds another dependency. Its embedding rows are projected, then each layer's per-layer contribution is combined at the corresponding point in the forward path. A fence lets the CPU wait for an earlier GPU result only when that value is needed. The point is not to make every line asynchronous: the CPU and GPU have a dependency, so the engine queues independent operations and places waits where data crosses back. Chapters 9–15 take these pieces apart, beginning with the CPU reference and then its Metal counterpart.

Which layers share state?

What do "42 layers" and "24 KV owners" mean for memory? The loader reads the block count from GGUF metadata at `gemma4.c:932`, then reads the count of shared KV layers and derives how many layers actually need to allocate cache entries at lines 944–946. Per-layer attention settings, including the sliding-window pattern, are read while the loader fills each `Layer` record around lines 981–998. This is why the engine can follow the model file's architecture rather than rely on a hand-maintained table of layer numbers.

A sliding-window layer attends only to a recent ring of positions; a global layer can attend to the full retained history. The layer's `swa` setting tells attention which rule applies. Some later layers point back to an earlier compatible KV source, so they can reuse that layer's keys and values instead of keeping a duplicate cache. For this E4B layout, 24 layers have cache state and 18 share. This saves cache memory and avoids computing/storing redundant K/V projections, while leaving each layer's own attention and feed-forward work in place.

The loader also reads the per-layer embedding width at line 943. If it is nonzero, the forward path builds the projected per-layer input and combines it through the 42-layer loop. These are three separate model-file choices: block count determines loop length, shared-KV metadata determines cache ownership, and PLE width enables the extra branch. Keeping them as metadata-driven fields is what lets the same engine source handle related model variants without guessing from a filename. The architecture fields and their use in cache allocation are unpacked in Chapters 10, 11, and 13.

Choosing one score without copying them all

The output head produces a logit for each vocabulary entry. For a vocabulary of 262,144 entries, a float32 array would be about one megabyte; copying the whole array to the CPU for every generated token would add traffic and synchronization. Greedy decoding needs only the highest-scoring ID. The Metal argmax kernel reduces the scores on the GPU and writes the selected integer into the token buffer, where the next step can consume it. This is why the C decode loop calls `metal_argmax` and places a CPU fallback behind the same `STEP` contract.

The token is not necessarily printed immediately by the CPU in the pipelined path. The GPU writes it, the next GPU step can use it, and the CPU waits at the fence when it needs the completed token to stream text, detect a stop marker, or update chat state. The README's measured decode numbers therefore

PREPRINT

include a coordinated pipeline rather than a single isolated kernel. The book's performance chapters will separate kernel time, end-to-end time, and synchronization effects.

The output head has a real cost

Why not stop computation as soon as the model has a good answer? The output head maps the final hidden row to one score per vocabulary item. The engine must consider the complete vocabulary to know which score is largest; the GPU argmax reduces those scores, but it cannot skip the matrix multiply that creates them. Historical analysis measured the Q4_K output head at about 377.5 MB of a 2,635 MB decode read, roughly 14 percent of the bytes for a token. That is a useful scale check: the last operation is visible in the timing budget even though it is only one matrix.

The analysis also explains why an optimization can look attractive in a small kernel and barely move end-to-end speed. If the other layer matrices already stream near the machine's practical bandwidth, saving a few microseconds in one attention kernel affects only a small fraction of the full step. Fusing operations can reduce dispatch overhead, but it does not erase the bytes that must be read. These are historical measurements and analysis from ENGINE-MATH.md §6 and §11, not a prediction for every quantization. Chapter 21 will measure the current snapshot and make its conditions explicit.

A route through the two speed regimes

The prefill measurement and decode measurement answer different questions. Prefill reports how fast a batch of prompt tokens is processed; its matrix multiplications reuse each weight tile across multiple rows. Decode reports how fast one continuation token follows another; it streams the layer weights for each step and has a dependency on the just-selected token. A benchmark should name its prompt length, decode length, model quantization, and context depth or it is hard to compare with another number.

The small table earlier is deliberately modest about what is comparable. Its 440 tok/s and 29–30 tok/s entries are the README's Q4_0 benchmark on the M3 Air, while 0.000012 KL is from the separate IQ4_XS comparison. The 2.51 GB wired count describes the E4B mapping. None is a new result from this writing session. Chapter 21 will rerun and date the measurements. That distinction protects the reader from treating a familiar-looking number as a guaranteed property of every quantization, context, or machine load.

Experiment

Can the snapshot build and explain its own command line without opening a model? This is a W0 check: no weights are needed. From the book project root, build into the snapshot's designated output directory, then invoke the usage path:

```
nice -n 10 sh code/snapshot/build.sh
code/build/gemma4 --help
```

Inspect the generate, benchmark, perplexity, and chat options. The program's usage routine exits with status 1 on this argument-free path; that is expected. The captured output below came from running these commands on 2026-10-22 on an M3 MacBook Air (16 GB). No model was loaded.

PREPRINT

Experiment W0

The build completed with exit status 0 and no compiler output. `code/build/gemma4 --help` printed 25 lines and exited 1 as expected. Its first lines were:

```
usage: gemma4 -m model.gguf [-f text.txt] [-c n_ctx] [--chunks n] [-t threads]
              [--kl-divergence-base file [--kl-divergence]]
  -f text                  compute perplexity over the text
```

The full capture is in `code/expected/ch01-w0-help.txt`. This experiment confirms only that the snapshot builds and reaches its usage path; it says nothing about generated text, accuracy, or speed.

Try it on your machine (W2). In a scratch copy with a local model, compare the same greedy prompt on the Metal and CPU paths:

```
code/build/gemma4 -m "$MODEL" -p "Once upon a time" -n 32
code/build/gemma4 -m "$MODEL" -p "Once upon a time" -n 32 -ngl 0
```

Try it on your machine (W2): run the commands above in a scratch copy with the same prompt and options. Inspect the wired-weight banner, generated text, throughput, and greedy-token agreement. The chapter predicts matching greedy output for this short prompt; treat throughput as a measured result only when you record the run conditions.

Pitfalls from the build history

What can go wrong while the output still looks plausible? These are documented incidents, not hypothetical advice.

Pitfall

A stale executable can tell a convincing lie. The Chapter 1 notes record a live-folder binary dated October 3 beside source edited October 6. Output from that binary did not represent the source being inspected. The correction was to build and run the frozen snapshot under `code/build/`. Source: OUTLINE.md, Chapter 1 pitfall; PLAN.md §4.2 distinguishes the book snapshot build from the live engine tree.

Pitfall

A GPU decline can silently become a CPU slowdown. In the E2B prefill attention path, the host guard rejected eight query heads for the 8-bit-cache kernel and returned "not handled." The engine correctly fell back to CPU attention, so answers remained correct, but the CPU spent 27.7 seconds for about 5 seconds of wall time. The Metal host and kernel were changed to split the heads across threadgroups; prefill rose from about 450 to 1,652 tok/s. Source: ENGINE-MATH.md §12, dated 2026-10-06; the original kernel is preserved at `backups/2026-10-06-e2b/gemma4_metal.m.orig`, with the before/after checks recorded there.

Pitfall

A correct-looking transcript missed a token-buffer race. The README's bug history records that the CPU could overwrite token storage before the GPU had finished reading it. Perplexity/KL then jumped to 0.83 instead of about 0.00001. The repair added the required GPU synchronization before reusing that buffer. Source: README.md, "Bugs fixed on the way" (token-buffer race).

Pitfall

A mathematically safe attention shortcut still shifted measured accuracy. A static maximum made one decode-attention kernel slightly faster, but end-to-end testing found mean KL divergence 0.000094 and a changed top token at 0.2% of positions. The cause remained unexplained, so the variant was reverted at the user's request. The session notes say to validate the complete model path, not stop at an exact-arithmetic argument or a kernel microbenchmark. Source: ENGINE-MATH.md §6.1; its dated session log is identified there as `ae10d6aa-…jsonl` (2026-10-03).

Exercises

Exercises

1. Estimate the number of model weights from the 4.72 GB file size and an average 4.25 bits per weight. State what the estimate leaves out.
2. Using 2.7 GB streamed per decode token and 100 GB/s, estimate the bandwidth ceiling in tokens per second. Compare it with the historical rate and give two reasons the measured result is lower.
3. Trace the file mapping: find where `gguf_open` sets up the file view and where `map_model` makes selected regions available to Metal. Why is the buffer no-copy?
4. Find `tokenizer_load`, `chat_render`, and the longest-prefix reuse in `chat_run`. In your own words, which work can a repeated chat turn avoid?
5. Follow one greedy decode step from the end of `forward` to GPU argmax and then to printed text. Which values cross back to the CPU?
6. Build the snapshot with the W0 commands and explain why the expected usage-path exit status is 1. Which experiment in this chapter would require weights and is deliberately not run?

Rebuild checkpoint

This is an orientation chapter, so it does not introduce a rebuild milestone. Chapter 2 starts checkpoint m1: implement a tokenizer and compare its IDs with the snapshot on a small sentence set. The later checkpoints build on the map shown here.

PREP

Chapter 2

Words to numbers: the tokenizer

The question

How does a line of text become the exact list of integer IDs that this model expects, and how can those IDs become readable text again?

The idea

Does the model read words?

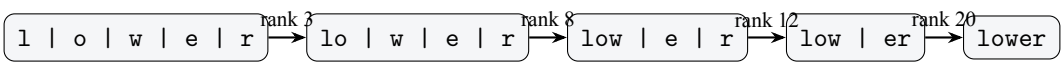
The model does not receive words, letters, or a UTF-8 string. It receives integer IDs. Each ID selects a row from the model's embedding table, so changing even one ID changes the input the network sees. The tokenizer is the small program at the boundary: it turns bytes from the prompt into those IDs, and later turns generated IDs back into bytes for the terminal.

This vocabulary is large: the E4B file has 262,144 entries. Many entries are familiar-looking words, but a token is simply a reusable piece. It may be a whole word, a word fragment, punctuation, a space-prefixed fragment, a control marker, or one byte of a UTF-8 sequence. There is no reliable conversion from word count to token count. In the captured example later, "Hello, world" is 12 UTF-8 bytes and three text pieces; a short Thai greeting is 27 UTF-8 bytes and two text pieces. BOS, when requested, adds one ID to either input. These are examples, not a language-wide rate.

The tokenizer's data comes from the GGUF metadata, alongside the tensor directory and before the large tensor payloads described in Chapter 3. Two tables make repeated lookups cheap: token spelling to ID, and an adjacent pair of token IDs to the merge rank and resulting ID. The vocab and ordered merge list define what the tokenizer is allowed to produce. The code does not train a tokenizer or consult a language library at run time.

What happens to a short word?

Start with five symbols: `l | o | w | e | r`. In this hand-sized example, the available ranks first join `l+o`, then `lo+w`, then `e+r`, and finally `low+er`. Each step picks the available adjacent pair with the lowest rank and stops when no pair has a rule. These ranks illustrate the procedure; they are not the E4B ranks for this word.



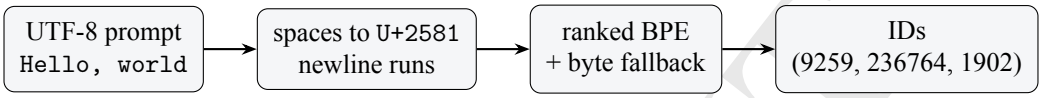
The arrows show one possible ordered merge table. In the real file, a merge's rank is its position in the metadata array; lower rank wins. BPE means byte-pair encoding: the starting symbols are Unicode characters represented as UTF-8 byte spans, and the permitted merges join adjacent symbols. A complete Unicode character is kept together when the run is initialized, even though it can occupy several bytes.

PREPRINT

Where do the spaces go?

The engine rewrites each ordinary ASCII space as the visible marker U+2581. The marker is not an extra space inserted for readability; it is the token vocabulary's spelling for a space. That lets the vocabulary distinguish a piece at the beginning of a word from the same letters after another piece. Three consecutive spaces become three markers, and the vocab may contain the whole run as one token, as the measured example shows.

Newlines follow a separate boundary rule. The text is divided into runs of newline bytes and runs of everything else. If a whole run of newlines is already a vocabulary entry, it is emitted at once. Otherwise that run goes through BPE like any other run. Ordinary non-newline text is not first split into words by punctuation or a regular expression. This matters: inserting a convenient word splitter would change which adjacent pairs are eligible and therefore change the IDs.



The picture is the plain-text path. Chat adds a second path: it writes role and turn markers such as <|turn> and <turn|> into the rendered prompt, and asks the tokenizer to recognise known special-token spellings. A literal string that looks like a control marker in an ordinary prompt is not automatically a control token. The caller chooses that behavior with a flag. The distinction prevents a user from accidentally ending a message merely by typing characters that resemble a protocol marker.

How does it choose a merge quickly?

For a run with n starting symbols, a simple tokenizer could scan every adjacent pair, pick the best rank, merge it, and scan again. That repeats work: most pairs are unchanged after each merge. The engine instead puts eligible pairs in a min-heap keyed by (rank, left position). It pops the best candidate, checks that the two symbols are still neighbours with the same IDs, and skips the candidate if an earlier merge made it stale. A merge changes only the two neighbouring pairs, so the engine adds those new candidates rather than rebuilding the heap.

The linked `prev` and `next` arrays describe the live sequence. A merged right-hand node is marked with length zero and bypassed. The left node now spans both original pieces. Keeping the original byte start and combined length means the code can look up the concatenated spelling without allocating a new string for every candidate. The heap's tie-break on left position makes equal-rank choices deterministic and leftmost-first.

What if a character is not a token?

If a starting character has no direct vocabulary entry, it receives ID -1. That makes it ineligible for a pair merge, since a merge table is indexed by two valid IDs. When the algorithm emits the remaining symbols, any one without a token ID is expanded into its original bytes and each byte is looked up as a special spelling such as <0xE2>. The decoder can therefore preserve an unfamiliar character without replacing it with a lossy question mark.

This fallback is byte-accurate, not character-accurate. A single displayed character can be represented by several byte tokens. A later generated sequence can also end partway through a multi-byte UTF-8 character. The output routine appends each token's bytes and writes them in order; it does not prove that every individual token is a valid UTF-8 string. Treating a token as a C string and printing it independently would be unsafe for display. The engine's `detokenize` returns a byte count, and its printer uses `fwrite`, which can emit those bytes without looking for a NUL terminator.

PREPRINT

What makes a control token different?

A GGUF token-type array marks control and user-defined entries, and `tokenizer_load` collects those IDs into a separate list. When the caller enables `TOK_SPECIAL`, `tokenize` scans for a matching spelling and chooses the longest match at that position. A match flushes the ordinary text before it through the normal path, appends the one control ID, then resumes after its spelling. With the flag off, the same characters are just ordinary text and go through whitespace handling and BPE.

The chat renderer in `chat_render` builds a protocol stream. In a plain user turn, it places the BOS ID if enabled, tokenizes `<|turn>user` as a marker plus ordinary text, appends the user's text, then adds `<turn|>` and the start of the model turn. The marker spellings are vocabulary entries: in this E4B file, BOS is 2, `<|turn>` is 105, and `<turn|>` is 106. The next model turn is not simply concatenated prose; its IDs identify who is speaking and where the turn ends.

The GGUF also contains `tokenizer.chat_template`. This engine reads that metadata to detect the template variant, while its `chat_render` function constructs the actual E4B stream in C. For example, thinking mode can add `<|think|>` in a system message; image messages use `<|image>` and `<image|>` around encoded image rows. The markers establish boundaries. The encoded image rows themselves are not text tokens from the tokenizer. Chapter 16 follows the whole chat protocol; here the important point is that the template's printed marker spellings become designated IDs only when the renderer requests special-token handling.

Can IDs turn back into text?

Detokenization looks up each ID's vocabulary spelling. For a normal piece, it copies the spelling's bytes while replacing every `U+2581` marker with one ASCII space. For a byte token like `<0xE2>`, it emits that one byte. For control tokens, it emits the visible spelling; generation code separately checks stop IDs before printing, so the end-of-turn marker is used for control flow rather than shown as reply text.

Tokenization and detokenization are not perfect inverses over every sequence of IDs. A user-provided string is converted to the canonical merges chosen by the rank table. The model, however, may generate a different sequence of token IDs whose pieces concatenate to the same bytes. Re-tokenizing those bytes can yield a different ID sequence. The visible text can match while the sequence of IDs differs; that is a key detail for chat-prefix reuse, not a contradiction in the tokenizer.

What does loading the metadata buy us?

The GGUF does not store one ready-made dictionary object. It stores an ordered vocabulary array and an ordered merge array as metadata values. `tokenizer_load` turns those arrays into two indexes that match the hot questions asked while tokenizing: "which ID has this byte spelling?" and "is this pair mergeable, and what is its rank and result?" The original arrays remain the source of truth; the indexes are a derived in-memory view.

The vocabulary map uses open addressing. A hash selects a starting slot, then linear probing moves forward until the spelling is found or an unused slot proves there is no match. The table is sized to a power of two at least twice the number of entries. That leaves room for probes and allows a mask to wrap the index without a division. The slot keeps a pointer and byte length into the vocabulary string, plus its ID. Equality checks both length and bytes, so `word` cannot match the first four bytes of `wording`.

The pair map packs the left and right token IDs into one 64-bit key. It too uses open addressing. Every merge string is divided at its separator into left spelling and right spelling. The loader finds each part's token ID, joins the spellings to find the result ID, and stores the rank from the array position. Some malformed or unsupported merge entries are skipped instead of being allowed to create an invalid pair key. In the normal GGUF, rank is the list position: there is no separate floating-point "score" to approximate.

Why retain both the result rank and result ID? The algorithm has to decide which pair goes first, but after selecting it must also know the token ID of the concatenated spelling. If that spelling is absent, the merge record can still establish an ordering while the code checks the concatenated span against the

PREPRINT

vocabulary. A missing result ID does not invent an ID; the final symbol then takes the byte fallback route. That guard keeps the ranking structure and the vocabulary structure separate.

What do the measured IDs tell us?

The W1 probe gives a concrete shape to the metadata. This E4B GGUF has 262,144 vocabulary strings and 514,906 merge strings. Its model tag is `gemma4`; its BOS ID is 2, and its EOS ID is 106. The same ID is the visible `<turn|>` marker, which is why the end of an assistant turn is both a control-flow event and a vocabulary item.

The two familiar English pieces in "Hello, world" are not one token apiece. The comma stands on its own, and the space is part of the last piece's spelling: `Hello` (9259), `,` (236764), `U+2581world` (1902). The probe reports 12 input bytes and three text pieces, plus the shared BOS ID. A three-space input happens to be one token, ID 139, whose spelling contains three space markers. The vocabulary has learned that entire repeated run as a piece. A program that guessed one token per word, punctuation mark, or whitespace character would get both examples wrong.

The Thai greeting occupies 27 UTF-8 bytes but only two vocabulary pieces in this file. The emoji occupies four bytes and is one vocabulary piece. These outcomes do not establish that every Thai sentence is short in tokens, or that every emoji is one token. They show why a tokenizer experiment must inspect the vocabulary's actual IDs rather than derive token count from character count or byte count.

The control example exposes a caller choice. In chat mode, `<|turn>` is found in the special-token list and becomes ID 105; the following letters `user` remain ordinary text and become ID 2364. In the `--tokens` inspection mode, the C code does not pass `TOK_SPECIAL`, so a typed marker spelling follows the regular BPE path instead. The W1 helper deliberately enables special handling for the control example to demonstrate the chat path; it is a header-and-vocabulary experiment, not a run of the engine's CLI.

Why keep a heap if the merge table already has ranks?

The table answers a local question for any pair of current neighbours. It does not tell the tokenizer which pair to process first across a long line. If a line has many candidate pairs, the heap keeps the best candidate available without rescanning every untouched pair after each merge.

Consider `a b c d` where the current candidates are `(a,b)` rank 14, `(b,c)` rank 3, and `(c,d)` rank 8. The rank-3 pair is merged first. That removes `b` and `c` as separate neighbours, so the old candidates touching either one cannot be trusted. The pair `(c,d)` entry still sits somewhere in the heap; deleting it by array position would require tracking and maintaining a second set of heap positions. Instead, the candidate records the left position and the IDs it expected. When it reaches the root, the code checks that those IDs are still present in that relationship. If not, the stale entry is discarded.

After that merge, only the pair to the left of the merged node and the pair to its right can be new. Those are the only candidates pushed. The arrays `prev` and `next` make those neighbours accessible even though the original nodes remain allocated. The `len` array marks consumed nodes as inactive. The original sequence order is never rearranged in memory; the links describe the current token sequence. This arrangement trades a little bookkeeping for avoiding repeated whole-line work.

Ties need a rule, too. The comparison uses rank first and original left position second. Therefore two equal-ranked pairs do not depend on heap insertion order or compiler behavior: the leftmost pair wins. This matters when reproducing another implementation's IDs. It is possible for two token sequences to decode into the same text while differing in which equal-ranked merge happened first; the visible answer alone would not reveal the mismatch.

What exactly counts as one UTF-8 symbol?

The run initializer looks at the first byte and infers a character width of one, two, three, or four bytes from the leading-bit pattern. If that width would extend beyond the end of the run, it is shortened to the

PREPRINT

bytes that remain. The source does not call a Unicode normalization library, combine accents, or rewrite punctuation. It preserves byte spelling and applies the model's own merge rules.

This is a practical boundary, not a full Unicode validator. A valid UTF-8 prompt is expected at the interface. For malformed input, a byte whose leading bits do not match a multi-byte pattern is treated as a one-byte symbol; an apparently multi-byte lead cut off at the end is clamped to the remaining bytes. If no resulting spelling exists, the normal fallback attempts to emit the available bytes individually. This is another reason to keep the byte count explicit and not assume every token's spelling is a well-formed standalone string.

Spaces and newlines are also byte-level cases. The space replacement recognizes byte `0x20`, not every Unicode whitespace character. A tab is not rewritten to the space marker. A carriage return is not a newline run by itself. The only special run boundary in `tokenize_text` is the line-feed byte, `0x0A`; all other bytes stay in the non-newline run and are subject to ordinary BPE. That precise rule is easy to overlook if one thinks in terms of what a terminal displays instead of what bytes the C loop compares.

The 3-byte spelling U+2581 must be copied as a unit when it is inserted. The code reserves enough additional space for two extra bytes for each ASCII space, then writes the three marker bytes. This capacity calculation is not a token-count estimate; it is the maximum byte length of the expanded input. After expansion, all offsets in `bpe_run` refer to the expanded buffer. When detokenizing, the reverse search recognizes exactly those same three marker bytes and writes one ASCII space.

Where does text end and protocol begin?

A template is ordinary C strings until the renderer asks the tokenizer to treat selected spellings as control IDs. The renderer does not pass one giant string and hope the tokenizer guesses which pieces are special. It appends template fragments separately and calls `push_text`, which turns on `TOK_SPECIAL` for text that may contain protocol markers. For a user's message body, the renderer trims its edges and tokenizes its content with the same option; the special-token list decides whether an embedded marker spelling has protocol meaning.

That makes the tokenizer a protocol boundary as well as a text boundary. BOS is prepended only when the model metadata says it is enabled and the caller passes `TOK_BOS`. EOS is not automatically appended to ordinary text. In generation, the stop IDs come from the model metadata and the chat code also treats `<turn|>` as a stop marker. When a generated stop arrives, it ends the reply rather than being printed as one more body piece.

The chat template has a useful visual grammar: a start marker names a speaker; newline separates the header from the body; an end marker closes the message; and the model turn begins with its own header. Additional markers can name a channel such as `<|channel>`thought`<channel|>` or bracket media rows. The word `thought` in that header is text between special markers, not itself necessarily a special token. In the actual E4B metadata the relevant channel markers have their own IDs (100 and 101), and the thinking marker has ID 98. Token IDs preserve this distinction even where their printed spellings sit next to one another.

Why can the same bytes produce different ID lists?

Suppose the model emits the two pieces `a` and `b`, but the vocabulary also has a single piece `ab` and the merge table prefers it. Detokenization emits the bytes `ab`. If the chat renderer later reconstructs the stored reply from text, re-tokenization chooses the canonical one-piece sequence. Both sequences spell the same bytes, yet their lengths and IDs differ. This is normal for a subword vocabulary: it assigns a preferred segmentation to new input but the model can generate any vocabulary ID at each step.

The chat cache compares token entries, not printed strings. Its longest unchanged prefix can therefore stop when it reaches the first raw-generated ID that differs from the canonical re-tokenization. The renderer is correct to prioritize text as stored in the conversation; the cache is correct to reuse only entries whose IDs match. A user may observe additional prefill on a later turn even though the transcript

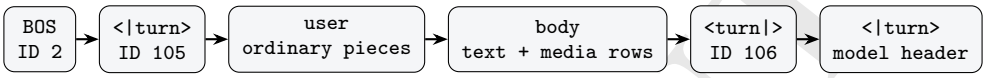
PREPRINT

appears unchanged. Chapter 16 follows the cache logic, but the tokenizer explains why byte equality is not sufficient evidence for token equality.

This also gives a useful test for a rebuilt tokenizer. Comparing detokenized output is a round-trip test for byte preservation, but it is not the compatibility test. Compatibility means matching IDs for the same input under the same flags, including BOS and special-token handling. The m1 checkpoint therefore compares token lists directly on a fixed set of samples. If a mismatch occurs, printing each ID beside its vocabulary spelling often narrows the cause to space marking, a merge rank, a tie, a newline run, or the caller's special-token flag.

Can we read a chat prefix as a sequence?

A one-message prompt makes the role of these IDs concrete. The renderer starts with BOS when enabled, then adds a user header, the body, the user-turn ending, and the open model header. In readable form, that shape is:



This is a structural sketch rather than a complete literal transcript. The newline after a role separates the header from the body. Some modes insert a system message first, and a thinking-enabled prompt can add a channel marker before the message ends. A media message can insert actual encoder rows between its begin and end markers. Those rows occupy prompt positions but are not produced by BPE. Chapter 16 follows these variations; the tokenizer's job is to preserve the token stream the renderer deliberately assembled.

Notice that the visible role word is not automatically a protocol ID. Only spellings whose token types were marked special are recognized as special. Here the header string is broken into a special marker and ordinary text. The same principle applies to channel names. A reader looking at decoded text might see `<|channel>thought<channel|>` and assume it is a single magic phrase. The model actually receives a sequence of IDs: marker ID, text pieces for the channel name, closing marker ID. This distinction lets one marker introduce many channel names without assigning each full phrase a separate control ID.

The first generated ID is another part of the contract. The user prompt ends at the open model header; the model supplies the next ID after that prefix. When it eventually emits the stop ID, the generation loop stops. If the tokenizer were to append EOS automatically to every input string, it would close the prompt before generation began. That is why the BOS flag and the stop handling live at different call sites instead of being hidden inside generic text conversion.

How can a no-weight experiment still be useful?

The W1 helper, `tokenizer_probe.py`, reads only the header and key/value metadata needed for tokenization. It parses the GGUF's typed metadata values, retaining tokenizer model name, vocabulary, merge list, token types, and BOS/EOS settings; values it does not need are skipped. Once the metadata table ends, it stops. The tensor count in its first line comes from the header, but it never reads tensor names, offsets, or payload bytes. It does not make a memory map, initialize Metal, or open a model through the engine.

This narrow reader is useful for checking claims about a particular local file before asking the engine to use it. It can report the vocabulary size and spellings, resolve a special marker to its ID, and run examples through a small reference implementation of the BPE rule. It cannot tell us whether generated text is good, how fast tokenization is on every input, whether the GPU path is correct, or whether the engine and an external implementation agree on all edge cases. Those require different checks. In particular, the script's example IDs are produced by its own implementation of the source algorithm; the project deliberately does not run the model's executable just to print IDs, because the book's W1 scope is limited to metadata access.

PREPRINT

When comparing a future m1 tokenizer with the engine, keep the input and flags in the test record. Does each path add BOS? Does it recognize special markers? Is the prompt sent as plain continuation text or rendered chat? Did both paths apply the same edge trimming? A mismatch without those details is hard to diagnose because two valid callers may intentionally produce different IDs from the same visible characters.

What does a useful mismatch report contain?

For each fixed sample, record the original bytes, the flags, the resulting IDs, and the vocabulary spelling for every ID. For a Unicode test, include the UTF-8 byte length as well as the visible input. If two tokenizers disagree, compare the first differing position rather than only the total count. An early mismatch can shift all later indices while leaving later spellings familiar. The five W1 examples cover distinct paths: an English word and punctuation, a non-Latin run, a multi-byte symbol, a potential control spelling, and repeated spaces.

If the first mismatch occurs before any merge, inspect the initial character spans and byte tokens. If it occurs only around a space, inspect the marker expansion and whether a leading-space piece exists. If it occurs after a newline, verify that the complete run is checked before BPE. If the same tokens are available but a different merge is selected, print each candidate pair's rank and left position. If only a marker spelling differs, check the token-type array and the caller's special flag. This turns "the ids are wrong" into a sequence of small, observable questions.

It is tempting to repair a mismatch by adding a special case for one test string. That hides the first incorrect assumption. The tokenizer is driven by metadata, so the durable check is the source of the candidate rule and its rank. A proper test corpus should include both familiar and boundary inputs: leading and trailing spaces, several spaces, one and several newlines, punctuation adjacent to words, mixed ASCII and multi-byte text, byte fallback, and strings that resemble control tokens. This is the role of the 20-sentence comparison in checkpoint m1.

What does detokenization promise?

The function accepts an ID, an output pointer, and a capacity. It returns how many bytes it wrote; it does not return a C string and does not add a terminating NUL byte. That interface is deliberate. A token spelling can contain embedded bytes that are not meaningful as a complete string by themselves, and a byte-fallback ID may contribute only one byte to a multi-byte character. The caller owns the larger output buffer and advances by the returned count.

For ordinary vocabulary pieces, the decoder scans by byte offset. When it sees the three-byte U+2581 sequence, it writes one byte, the ASCII space, and advances past all three marker bytes. Otherwise it copies one byte at a time until it reaches the token spelling's end or the caller's capacity. For a six-character spelling of the form `<0xXX>`, it parses the hex value and writes exactly one byte. These are different representation cases that share a byte-oriented result.

The generation printer uses a small per-token buffer, calls `detokenize`, then writes the exact returned byte count. It must not call `strlen` on that buffer: a generated token can contain a NUL byte, and the buffer need not have a terminator. It also must not assume that every token finishes a Unicode character. If a stream is cut off between the first and second byte of a character, a terminal may show a replacement glyph or nothing at that moment; a later token can complete the byte sequence. The byte stream, not each token in isolation, is what should be decoded as UTF-8.

There is a useful boundary between this conversion and the chat driver. Stop IDs are checked before they are printed as reply text. This keeps a control spelling like `<turn|>` out of the visible answer. Other control IDs can be displayed as their vocabulary spellings when the caller wants to inspect them. Detokenization itself does not know whether an ID means "stop", "image begins", or a normal word piece; it only converts the vocabulary spelling back to bytes. Protocol decisions belong to the code that consumes the ID.

PREPRINT

Why not tokenize special markers by default?

There are two reasonable jobs for a tokenizer call. A plain-text prompt or a scoring corpus needs a stable mapping from all of its input bytes through BPE. A chat renderer also needs to turn its own protocol strings into control IDs. Treating every marker-looking substring as special in every context would mix those jobs and make a text document containing `<|image>` act as if an image were attached.

The explicit flag puts that decision in the call site. `tokenize` can prepend BOS independently from its special scan. The chat helper combines `TOK_SPECIAL` with its text append operation; the ordinary `--tokens` path asks only for BOS. A caller that intentionally wants to inspect chat markers should therefore use the chat renderer or an equivalent call with special recognition enabled. A single test command is not enough unless its flags match the path being tested.

When special recognition is active, the scan is longest-match-first. If one marked spelling is a prefix of another, selecting the longest avoids emitting a shorter control token and leaving the rest of the longer spelling as ordinary text. The code checks the marked list at each input position, flushes the plain-text gap before the match through `tokenize_text`, inserts the control ID, and continues after the entire match. Thus the normal BPE path remains unchanged on either side of a marker.

This detail also explains why spaces and special markers are handled at different stages. The marker scan happens on the original input bytes before `tokenize_text` changes spaces to U+2581. The pieces on either side are then tokenized independently, so an ordinary merge cannot cross through a control token. This is the boundary the template needs: it preserves the protocol ID while allowing ordinary body text to use the full BPE vocabulary.

What can a tokenizer test prove?

An ID comparison on a fixed input proves compatibility for that input and those flags. A set of 20 examples gives a useful first checkpoint, but it is not a proof for every possible Unicode string or every token sequence. The vocabulary is large, and rare byte combinations may take fallback routes that a small English corpus never exercises. Keep the test list as a file so a change to the merge loop can be compared against the same bytes later.

Test both directions with different expectations. For text input, compare the exact ID list with the reference path. For output, concatenate the decoded bytes and compare them with the expected byte string. Do not require that re-tokenizing every generated ID sequence produce the same list: as discussed above, several ID sequences may spell the same bytes. A round trip that compares only rendered text can miss an ID mismatch; a round trip that demands identical IDs can reject a valid non-canonical generated sequence.

Finally, keep special tokens in a separate test group. A role marker recognized in chat mode and the same spelling treated as ordinary text are both valid results from different call sites. Record the mode next to the expected IDs. This small discipline avoids a confusing class of failures where the implementation is correct but the test accidentally asks it to tokenize a chat template as plain prose, or plain prose as protocol.

The real code

How does a spelling find its ID?

The vocabulary lookup is an open-addressed hash table. It hashes the bytes, checks the slot, and walks forward until it finds either an empty slot or an exact length-and-byte match. The length check matters because token spellings are byte spans, not necessarily NUL-terminated strings. Pair lookup uses the same pattern with a packed pair of integer IDs as its key. This excerpt shows those lookups and the UTF-8 leading-byte length helper that lets the BPE run start with whole characters.

```
3461 static int token_id(const Tokenizer *tk, const char *s, size_t n) {
```

PREPRINT

```
3462     for (uint64_t i = hash_bytes(s, n) & tk->str_mask;; i = (i + 1) & tk
         ->str_mask) {
3463         const StrSlot *e = &tk->str_map[i];
3464         if (!e->s) return -1;
3465         if (e->n == n && memcmp(e->s, s, n) == 0) return e->id;
3466     }
3467 }
3468
3469 static const PairSlot *find_pair(const Tokenizer *tk, int a, int b) {
3470     if (a < 0 || b < 0) return NULL;
3471     const uint64_t key = (uint64_t)a << 32 | (uint32_t)b;
3472     for (uint64_t i = hash_u64(key) & tk->pair_mask;; i = (i + 1) & tk->
        pair_mask) {
3473         const PairSlot *e = &tk->pair_map[i];
3474         if (e->rank < 0) return NULL;
3475         if (e->key == key) return e;
3476     }
3477 }
3478
3479 static int utf8_len(uint8_t c) { return c < 0x80 ? 1 : (c >> 5) == 6 ? 2
         : (c >> 4) == 14 ? 3 : (c >> 3) == 30 ? 4 : 1; }
```

Listing 2.1: gemma4.c:3461–3479

What does loading the vocabulary prepare?

The loader first checks that this GGUF declares the supported gemma4 tokenizer. It records BOS, reads token types to collect special IDs, then inserts every vocabulary spelling into the string map. Finally it prepares all 256 byte-token IDs. The excerpt ends there; the next part of the same function reads the ordered merge array and builds the pair-to-rank table. Together, those tables let tokenization avoid scanning the full 262,144-entry vocabulary for each character or pair.

```
3481 static Tokenizer tokenizer_load(const Gguf *g) {
3482     Tokenizer tk = {0};
3483     Str model = kv_str(g, "tokenizer.ggml.model");
3484     if (!str_eq(model, "gemma4")) die("tokenizer %.*s not supported (
         need gemma4)", (int)model.n, model.s);
3485     tk.bos = kv_num(g, "tokenizer.ggml.add_bos_token", 1) ? (int)kv_num(
         g, "tokenizer.ggml.bos_token_id", -1) : -1;
3486     const KV *types = find_kv(g, "tokenizer.ggml.token_type");
3487     if (types) {  // control (3) and user-defined (4) tokens
3488         Cursor c = {types->val, g->base + g->size};
3489         const uint32_t et = rd_u32(&c);
3490         const uint64_t n = rd_u64(&c);
3491         tk.special = xmalloc(n * sizeof(int));
3492         for (uint64_t i = 0; i < n && et == GGUF_I32; i++) {
3493             int32_t t;
3494             memcpy(&t, c.p + 4 * i, 4);
3495             if (t == 3 || t == 4) tk.special[tk.n_special++] = (int)i;
3496         }
3497     }
3498
3499     tk.vocab = kv_str_array(g, "tokenizer.ggml.tokens", &tk.n_vocab);
3500     const uint64_t ns = pow2_at_least(2 * (uint64_t)tk.n_vocab);
3501     tk.str_map = xcalloc(ns, sizeof(StrSlot));
3502     tk.str_mask = ns - 1;
3503     for (int id = 0; id < tk.n_vocab; id++) {
3504         const Str v = tk.vocab[id];
```

```
3505         uint64_t i = hash_bytes(v.s, v.n) & tk.str_mask;
3506         while (tk.str_map[i].s) {
3507             if (tk.str_map[i].n == v.n && memcmp(tk.str_map[i].s, v.s, v
     .n) == 0) break;  // keep the first id
3508             i = (i + 1) & tk.str_mask;
3509         }
3510         if (!tk.str_map[i].s) tk.str_map[i] = (StrSlot){v.s, (uint32_t)v
     .n, id};
3511     }
3512     for (int b = 0; b < 256; b++) {
3513         char buf[8];
3514         snprintf(buf, sizeof buf, "<0x%02X>", b);
3515         tk.byte_id[b] = token_id(&tk, buf, 6);
```

Listing 2.2: gemma4.c:3481–3515

Where does the heap change the sequence?

The BPE routine starts with UTF-8 character spans and one linked-list node per symbol. It seeds the heap with mergeable neighbours. The loop checks the candidate against the current IDs and links before applying it; the comment marks the stale-candidate test. Once merged, only the left and right neighbours need new heap entries. At the end, a valid ID is emitted directly; otherwise each original byte is emitted through the byte-ID table. This is the heart of the algorithm, and the excerpt is intentionally limited to 40 lines.

```
3595 static void bpe_run(const Tokenizer *tk, const char *s, int n, int **out
     , int *n_out, int *cap) {
3596     // symbols: [start, len) in s, token id (-1: not in the vocabulary),
      linked list
3597     int *start = xmalloc(n * sizeof(int)), *len = xmalloc(n * sizeof(int
     )), *id = xmalloc(n * sizeof(int));
3598     int *prev = xmalloc(n * sizeof(int)), *next = xmalloc(n * sizeof(int
     )), ns = 0;
3599     for (int i = 0; i < n; ns++) {
3600         int l = utf8_len((uint8_t)s[i]);
3601         if (i + l > n) l = n - i;
3602         start[ns] = i, len[ns] = l, id[ns] = token_id(tk, s + i, l),
     prev[ns] = ns - 1, next[ns] = ns + 1;
3603         i += l;
3604     }
3605     if (ns) next[ns - 1] = -1;
3606     Cand *heap = xmalloc((2 * ns + 1) * sizeof(Cand));
3607     int nh = 0;
3608     for (int i = 0; i + 1 < ns; i++) {
3609         const PairSlot *p = find_pair(tk, id[i], id[i + 1]);
3610         if (p) heap_push(heap, &nh, (Cand){p->rank, i, id[i + 1], id[i
     ]});
3611     }
3612     while (nh) {
3613         const Cand c = heap_pop(heap, &nh);
3614         const int l = c.left, r = next[l];
3615         if (len[l] == 0 || r < 0 || id[l] != c.left_id || id[r] != c.
     right_id) continue;  // stale
3616         const PairSlot *p = find_pair(tk, id[l], id[r]);
3617         len[l] += len[r];
3618         id[l] = p->id >= 0 ? p->id : token_id(tk, s + start[l], len[l]);
3619         len[r] = 0;
3620         next[l] = next[r];
```

```
3621         if (next[r] >= 0) prev[next[r]] = l;
3622         if (prev[l] >= 0 && (p = find_pair(tk, id[prev[l]], id[l])))
         heap_push(heap, &nh, (Cand){p->rank, prev[l], id[l], id[prev[l]]});
3623         if (next[l] >= 0 && (p = find_pair(tk, id[l], id[next[l]])))
         heap_push(heap, &nh, (Cand){p->rank, l, id[next[l]], id[l]});
3624     }
3625     for (int i = 0; i >= 0 && ns; i = next[i]) {
3626         if (id[i] >= 0) {
3627             push_token(out, n_out, cap, id[i]);
3628         } else {
3629             for (int k = 0; k < len[i]; k++) {
3630                 const int b = tk->byte_id[(uint8_t)s[start[i] + k]];
3631                 if (b >= 0) push_token(out, n_out, cap, b);
3632             }
3633         }
3634     }
```

Listing 2.3: gemma4.c:3595–3634

Where do spaces and newline runs enter?

Before BPE, this function expands each ASCII space into the three-byte UTF-8 encoding of U+2581. It then walks the resulting byte string, making runs of newline and non-newline bytes. An exact newline-run vocabulary entry is a fast path; every other run goes to bpe_run. Notice what is absent: a split on words. That choice preserves the merge table's intended input.

```
3639 static void tokenize_text(const Tokenizer *tk, const char *text, size_t
         len, int **out, int *n_out, int *cap) {
3640     size_t n_sp = 0;
3641     for (size_t i = 0; i < len; i++) n_sp += text[i] == ' ';
3642     char *s = xmalloc(len + 2 * n_sp + 1);
3643     size_t n = 0;
3644     for (size_t i = 0; i < len; i++) {
3645         if (text[i] == ' ') { memcpy(s + n, "\xe2\x96\x81", 3); n += 3;
         }
3646         else s[n++] = text[i];
3647     }
3648     for (size_t pos = 0, end; pos < n; pos = end) {
3649         const int nl = s[pos] == '\n';
3650         end = pos;
3651         while (end < n && (s[end] == '\n') == nl) end++;
3652         const int whole = nl ? token_id(tk, s + pos, end - pos) : -1;
3653         if (whole >= 0) push_token(out, n_out, cap, whole);
3654         else bpe_run(tk, s + pos, (int)(end - pos), out, n_out, cap);
3655     }
```

Listing 2.4: gemma4.c:3639–3655

How does the caller opt into control tokens?

The flags let one tokenization routine serve ordinary prompts, scoring text, and chat. TOK_BOS prepends the model's configured beginning ID; TOK_SPECIAL enables the longest-match scan over the special list. The chat caller passes this flag for template fragments, while the --tokens inspection path passes BOS only. The distinction is visible in this excerpt and is why a printed <|turn> is not always the control ID.

```
3663 static int *tokenize(const Tokenizer *tk, const char *text, size_t len,
         int *n_tokens, int flags) {
```

```
3664      int *out = NULL, n_out = 0, cap = 0;
3665      if ((flags & TOK_BOS) && tk->bos >= 0) push_token(&out, &n_out, &cap
          , tk->bos);
3666      size_t start = 0;
3667      for (size_t i = 0; i < len && (flags & TOK_SPECIAL); i++) {
3668          int best = -1;
3669          size_t best_n = 0;
3670          for (int j = 0; j < tk->n_special; j++) {
3671              const Str v = tk->vocab[tk->special[j]];
3672              if (v.n > best_n && v.n <= len - i && !memcmp(text + i, v.s,
          v.n)) best = tk->special[j], best_n = v.n;
3673          }
3674          if (best < 0) continue;
3675          tokenize_text(tk, text + start, i - start, &out, &n_out, &cap);
3676          push_token(&out, &n_out, &cap, best);
3677          i += best_n - 1;
3678          start = i + 1;
3679      }
3680      tokenize_text(tk, text + start, len - start, &out, &n_out, &cap);
3681      *n_tokens = n_out;
3682      return out;
```

Listing 2.5: `gemma4.c:3663–3682`

The reverse path is short enough to summarize precisely. `detokenize` at `gemma4.c:3546` emits a byte token as one byte, converts the three UTF-8 bytes of `U+2581` back to a single ASCII space, and otherwise copies bytes up to the supplied capacity. `print_token` begins at line 4017; its `fwrite` at line 4022 writes exactly the returned byte count. This count-based interface is what makes byte fallback workable across generated tokens.

Experiment

Can we inspect the real vocabulary and see the difference between bytes, pieces, and IDs without loading weights? This is W1. The following small Python 3 standard-library script reads the GGUF header and metadata only. It stops at the end of the key/value table; it does not visit the tensor directory or tensor data and does not start the engine or use the GPU. Its BPE probe follows the source's rank-first, leftmost-on-tie rule and prints the selected vocabulary spellings beside the IDs.

Run it from the book root, replacing the path with the local E4B GGUF:

```
python3 code/tokenizer_probe.py "$MODEL"
```

Inspect the vocabulary and merge counts, the control-token IDs, and how token count differs from UTF-8 byte count. The captured run below is from the local `gemma-4-E4B-it-IQ4_XS.gguf`; the script reports its GGUF metadata and example tokenizations. For the marker example, this probe explicitly enables special-token recognition, as chat does. The engine's `--tokens` path passes BOS only, so it treats a typed marker spelling as ordinary text. No tensor payload was read and no inference was run.

DRAFT

Experiment W1

```
GGUF v3: 720 tensors (not read); vocabulary 262,144; merges 514,906
tokenizer.ggml.model = gemma4
tokenizer.ggml.bos_token_id = 2
tokenizer.ggml.eos_token_id = 106
special markers: <|turn>=105 <turn|>=106 <|channel>=100 <channel|>=101 <|think|>=98 <|
    image>=255999 <image|>=258882
'Hello, world': 12 UTF-8 bytes -> 4 ids [2, 9259, 236764, 1902] pieces ["<bos>", "Hello
    ", ",", "\u2581world"]
'\u0e2a\u0e27\u0e31\u0e2a\u0e14\u0e35\u0e04\u0e48\u0e30': 27 UTF-8 bytes -> 3 ids [2,
    49366, 66719] pieces ["<bos>", "\u0e2a\u0e27\u0e31\u0e2a\u0e14\u0e35", "\u0e04\u0e48
    \u0e30"]
'\U0001f642': 4 UTF-8 bytes -> 2 ids [2, 240731] pieces ["<bos>", "\ud83d\ude42"]
'<|turn>user': 11 UTF-8 bytes -> 3 ids [2, 105, 2364] pieces ["<bos>", "<|turn>", "user
    "]
'   ': 3 UTF-8 bytes -> 2 ids [2, 139] pieces ["<bos>", "\u2581\u2581\u2581"]
```

This is actual output from reading the local model's tokenizer metadata; the 720 tensor count is read from the header, but no tensor records or weights are touched. The Thai sample has two text pieces and the English sample has three; each output list also begins with BOS, so byte count alone does not predict token count. The emoji is one vocabulary token in this file; byte fallback is the path for a character or merged piece the vocabulary cannot represent. The complete capture is in `code/expected/ch02-w1.txt`.

PREPRINT

Pitfalls from the project notes

Pitfall

The text can look unchanged while the cached prefix gets shorter. The documented chat-state analysis says that the cache holds the model's raw generated IDs, while the next rendered conversation detokenizes a stored reply and tokenizes that text again. If the generated IDs were a non-canonical BPE sequence, the new IDs can differ even when their displayed text is the same. Longest-prefix comparison then stops at the first different ID and the engine recomputes more of the conversation. The fix is to compare the rendered entries by token ID and report the actual reuse boundary; do not infer cache reuse from equal-looking text. This is a source-level behavior analysis, marked "reading; not measured," in `ENGINE-MATH.md` §2.7; the implementation is `chat_render` and `chat_run` in `gemma4.c:4143` and `4262`.

Exercises

Exercises

1. **Read.** For `U+2581theU+2581` and this toy rank table—`t + h` rank 1, `th + e` rank 2, `U+2581 + the` rank 3, `U+2581the + U+2581` rank 4—write the merge order and final piece. Hints are in Appendix F.
2. **Read.** The heap may contain a candidate for a pair that no longer exists after a neighbouring merge. Which fields in `Cand` let `bpe_run` reject it, and why is skipping safer than trying to remove arbitrary heap entries?
3. **Read.** Why does the tokenizer begin with UTF-8 character spans rather than one node per byte, yet still need a byte fallback at output time? Trace one unfamiliar three-byte character.
4. **Change.** Extend `--tokens` to print each ID beside its vocabulary spelling. Make sure byte tokens are shown as `<0xXX>` and explain what your printer does with `U+2581`.
5. **Measure.** Tokenize a 200 KB UTF-8 text file and report bytes, IDs, elapsed time, and tokens per second. Repeat it, keep the input fixed, and state whether file reading is included in the timing.
6. **Change.** Add an optional special-token flag to a small tokenizer driver. Show that `<|turn>` is one control ID when the flag is on and ordinary BPE text when it is off.

Rebuild checkpoint

Rebuild checkpoint

Milestone m1: a tokenizer, not yet a model loader. Implement `steps/m1_tokenizer.c` so it reads the vocabulary and merge list from the GGUF key/value table—or, while Chapter 3's GGUF reader is unfinished, from a text dump produced by a separate reader. It should implement space marking, newline-run handling, lowest-rank/leftmost merging, byte fallback, optional special-token matching, and BOS insertion. Test it against the snapshot's IDs on 20 fixed sentences, including punctuation, repeated spaces, newlines, Thai, an emoji, and a marker spelling with special handling both on and off. Exact ID agreement is the check; readable output alone is not enough.

PROOF

Chapter 3

The model file: GGUF and mmap

The question

How can the engine use a 4.72 GB model file without first copying all of its weights into another 4.72 GB allocation?

The idea

The surprising part is that "loading" does not have to mean copying. The engine asks the operating system to map the file into its address space, then reads the small directory at the front. Each tensor record becomes a name, shape, type, and pointer into that mapping. The large payload stays in the file-backed mapping; pages arrive when CPU code or the GPU touches them.

Here is the path, from the file through the directory and into the GPU. The tensor pointer is an address, not a duplicate tensor. The Metal buffer is a wrapper over selected mapped pages, not a second weights array.

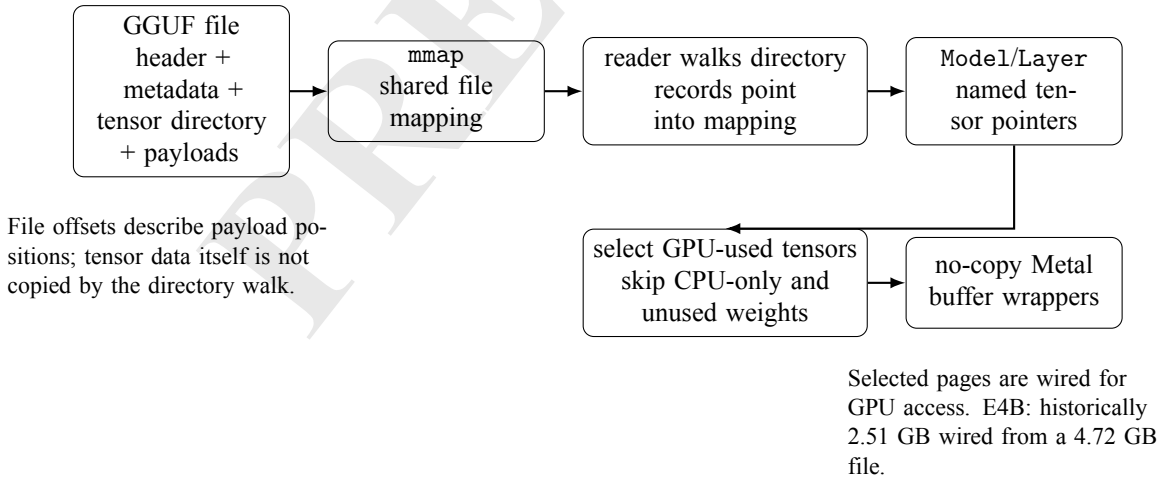


Figure 3.1: A GGUF load is first a map and a directory walk. Later, selected mapped regions are wrapped for Metal.

What does the directory describe? Start with a small made-up inventory: a file has a 4-byte magic, a version, counts for tensors and metadata entries, and then key/value records. After those records come tensor records. Each tensor record says, for example, "`blk.7.attn_q.weight`, two dimensions, this type, this offset." The payload bytes are stored later in one aligned data area. The directory is a map from useful names to locations, not a sequence of tensor contents.

PREPRINT

The file begins with a fixed-size header. In this engine's reader, the magic is read as a 32-bit little-endian integer and compared with `0x46554747`, which is the bytes `GGUF` in the file. The version follows. The next two 64-bit counts say how many tensor records and metadata entries to expect. A count is useful because it lets the reader walk records without scanning the entire file for a marker.

Metadata answers questions that are about the model as a whole: architecture name, layer count, embedding width, vocabulary, RoPE settings, attention window, and tokenizer arrays. The value is tagged with a type. A string has a length before its bytes; an array has an element type and a count. Some values are scalars, while others—such as a token vocabulary or per-layer pattern—are arrays. The engine keeps pointers to value bytes inside the map and decodes only the values it needs.

Then the directory lists tensors. A tensor has a length-prefixed name, a dimension count, up to four dimensions, a numeric type, and a 64-bit data offset. The shape and type together tell the engine how many bytes belong to that tensor. Offsets are relative to the beginning of the aligned tensor-data area, not to the beginning of the file. A file may store tensors in an order that differs from the order the model code wants, so the loader looks them up by name.

Why does alignment matter?

Suppose the tensor directory ends at byte 15,826,723 and the file's alignment is 32 bytes. The first payload starts at the next multiple of 32: byte 15,826,752. There are 29 padding bytes between the last record and that boundary. This is not wasted mystery data; it makes payload starts satisfy the layout's alignment rule. The offset in a tensor record is then added to that data-start address. For the first tensor, an offset of 0 means "start at the beginning of the aligned payload area."

Alignment is metadata too. The reader asks for `general.alignment`, defaulting to 32 if the key is absent. It rounds the current cursor position upward once, after the tensor records. The payload pointer is computed as `base + data_start + offset`. In our W1 capture, the records end at 15,826,723, the data region begins at 15,826,752, and the alignment value is 32. These values come from the local file's directory; the program did not touch its weight payloads.

The directory end and the data start are different because GGUF permits padding between them. The padding is not another tensor and is not included in any tensor's recorded size. A reader that forgets this distinction will point every tensor too early by the padding length. A reader that rounds every tensor's offset from the file start will be wrong too: each offset is already relative to the one aligned data start. Keep the three pieces separate in your notes: mapping base, aligned data start, and per-tensor offset.

The address calculation can be summarized in four named places:

Place	What it marks	How the reader uses it
Mapping base	Byte zero of the GGUF file	Starts the cursor and all mapped pointers.
Directory end	Byte after the last tensor record	Cursor position before padding.
Data start	Directory end rounded to GGUF alignment	Origin for every tensor offset.
Tensor start	Data start plus that record's offset	Pointer stored in `Tensor.data`.

Metal introduces another boundary after this table: page-rounded buffer start and end. That boundary exists for the GPU API; it does not replace the tensor's data start or its exact offset.

There are also two different reasons to care about alignment in this chapter. The GGUF alignment tells us where the tensor data area begins and constrains tensor offsets in that format. The operating system's page size is used later by the Metal wrapper. Those values need not be equal. The example file uses 32-byte GGUF alignment; Metal rounds buffer boundaries to system pages, which are much larger. Confusing these rules can produce a directory reader that works but a no-copy Metal buffer that the runtime rejects.

The dimensions have an orientation that is easy to misread. For a matrix with shape `[K, rows]`, `ne[0]` is the row length—the input-feature count—and `ne[1]` is the number of rows. A quantized row size depends on `ne[0]` and the stored block format; multiply it by the number of rows to get the matrix's byte count. The shape is not a promise that the first dimension is the number of rows. In the sample table,

PREPRINT

`blk.0.attn_q.weight` has dimensions `[2560, 2048]` and occupies 2,785,280 bytes as IQ4_XS. Its 2,560-value rows are the input width; there are 2,048 such rows.

How does the cursor know where a value ends?

Every metadata value starts with a type ID. A numeric scalar has a known width: for instance, a 32-bit integer occupies four bytes. A string begins with a 64-bit byte count, followed by exactly that many bytes. An array starts with an element type and an element count. If the elements are fixed-width numbers, the reader can advance by the count times that width; if they are strings, it must read each length-prefixed string in turn. That is why the C reader has both `scalar_size` and `skip_value`: the metadata table can be traversed without interpreting every value.

The cursor is a pair of addresses, a current pointer and an end pointer. Before a read, `need` checks that enough bytes remain. The small primitive readers advance only after that check. A length-prefixed string first reads its length and then checks that the stated number of bytes fit before returning a pointer to those bytes. The result is a byte span into the map; it need not be copied or NUL-terminated to compare it with a known key.

This arrangement separates two jobs. The cursor knows how to advance through the encoding and avoid running past the mapping. A metadata accessor knows what a particular key means. For example, `kv_num` looks up a key and converts its scalar representation; `kv_str` returns a string span; `kv_str_array` allocates only an array of spans for a caller that actually needs all the strings. The loader can skip a chat template or a large tokenizer array while still finding the tensor directory that follows it.

There are limits to what this proves. Bounds checks show that a value fits within the mapped file, and the tensor-end check catches payload ranges that extend past the file. The model loader then validates specific architecture and shape assumptions. This is not a general-purpose validator for every possible GGUF producer or hostile input. For the book's checkpoint, match the known file and source behavior first; robust overflow handling and a full conformance suite are separate engineering work.

What do the directory's byte totals include?

The W1 capture reports a file of 4,715,416,704 bytes. The data starts at byte 15,826,752, so roughly the first 15.8 MB contains the header, metadata, tensor directory, and alignment padding. Every tensor offset starts counting from the data area, not from the beginning of the file. The table of tensors occupies almost all the remaining file, but the sum of declared tensor sizes can still differ from file size because some payload space is padding and the metadata area is not a tensor.

The totals grouped by type answer a different question from the file size. They say how many bytes the known tensor shapes require under the encoded block sizes. They do not describe how many bytes the process has resident, which bytes Metal wires, or how much data one decode step reads. The last quantity depends on which tensors the forward pass visits; the wired total depends on the mapping policy; the resident amount depends on page use and operating-system state. All three can differ from the on-disk size.

The W1 type inventory is also a check on what this particular file actually contains. The captured table has BF16, F32, IQ4_XS, Q4_K, and Q5_K entries. A format supported by the engine's decoder need not appear in every GGUF, and a type mentioned for a different quantization variant should not be inferred from the model name alone. When a new type appears, first record its numeric type ID and shape; only add a byte-size calculation after confirming that type's block layout.

Why bother to calculate declared byte sizes if we are not reading the payload? It lets us see the scale of a tensor without touching its contents. The 1,937,768,448-byte token table and the 2,785,280-byte attention matrix differ by nearly three orders of magnitude. That contrast makes the mapping policy concrete: a giant table can be intentionally left for CPU row reads, while a smaller dense matrix is wrapped for GPU access. Size arithmetic is a directory operation; dequantization and inference are later operations.

Think of the directory as a set of address labels rather than a packed C structure. The on-disk fields have fixed encodings and variable-length strings, so the C reader advances a cursor and builds its own

PREPRINT

arrays of `KV` and `Tensor` records. The in-memory record layout is free to include convenient pointers and four dimension slots even when the file stored fewer dimensions. Missing dimensions are filled with one. That makes later size code simpler: it can multiply the dimensions without a special case for a vector or a matrix.

The payload itself is not converted into a C array of floats here. The `Tensor` struct stores the data pointer plus the shape and type needed by later code. For a quantized matrix, that pointer still addresses packed blocks in the GGUF. A CPU decoder or a Metal kernel interprets the blocks when it needs values. The file reader should not confuse "I found the bytes" with "I decoded the weights." Those are different milestones: m2 lists tensors, while m3 implements type-specific row decoding.

One pointer can therefore carry several kinds of context. The address says where the stored bytes begin. The dimensions say how those bytes are arranged into rows and any higher axes. The type ID says how many bytes represent each element group and how a compute path should decode it. A bare pointer without shape and type is not enough to safely address a quantized matrix. The loader keeps these fields together so a later matvec can pass the tensor record, select a type-specific path, and calculate each row's storage stride.

How does a pointer avoid a copy?

The reader calls `mmap` with `PROT_READ` and `MAP_SHARED`. The returned base address makes the file appear as a contiguous byte range in the process. Parsing a name or metadata value reads those bytes in place; assigning `t->data` calculates an address within the range. No routine reads a whole tensor into an equally large heap allocation just to establish its address.

That does not mean all bytes are instantly resident in physical memory. A mapping establishes virtual addresses and a relationship to a file. When code first reads a page, the operating system supplies it, often from the page cache or by reading storage. Later reads can reuse resident pages. The map is a way to address the file, not a guarantee that every page has been read. Loading the directory touches the first part of the file; inference later touches tensor regions as needed.

This is why a program can reserve a virtual range larger than the machine's free memory without immediately copying that range into anonymous memory. Virtual address space is an address map. Resident memory is the subset of pages currently held in physical memory. The distinction is useful when reading memory reports: a process can have a very large mapped file and a much smaller resident set, while GPU residency accounting may report file-backed pages that a process-only tool misses. The project's README says its memory measurement includes wired, anonymous, and compressed system memory for this reason.

The mapping also has a lifetime. Every tensor pointer in `Gguf` is valid only while its mapping stays open. The model loader keeps the `Gguf` alive during inference; it does not close the mapping after filling `Model`, because the model's tensor fields still point into it. A small standalone reader can close its mapping after printing the table because it has no later consumer. This is the same ownership rule as any pointer into a buffer: freeing the buffer invalidates every interior pointer even if the pointer values themselves remain unchanged.

Mapping a read-only file is not the same as making tensor memory immutable by magic. The process requested read access and the engine's tensor pointers are const-qualified, which express the intended use in C. The OS mapping permissions provide the corresponding protection. This is useful for both correctness and memory: the engine can give the GPU read-only model weights without first making a writable heap copy.

`MAP_SHARED` matters when those file-backed pages are wrapped for GPU access. The source comment explains the reason: wiring a private mapping for the GPU must not create copy-on-write pages. A private mapping may initially look shared, but making its pages writable can require private copies. For this reader the mapping is read-only, and the shared mapping lets the GPU wrapper refer to the same backing pages. It avoids turning the model into a second private allocation.

PREPRINT

How do names become model fields?

The raw table is general-purpose; the forward pass wants named fields. `load_model` first reads architectural settings from metadata: the E4B has 42 layers and a 2,560-wide embedding, and its metadata says 18 layers share key/value state. Those are not hard-coded tensor addresses. The loader uses the layer count to form names such as `blk.7.attn_q.weight`, looks them up, checks their dimensions, and stores pointers in the matching `Layer` fields.

That separation is why one loader can support related models with different dimensions. The GGUF metadata supplies the model's layer count, widths, attention settings, and per-layer embedding width. The code still checks that values fit its supported limits and that related tensor shapes agree. Metadata-driven does not mean "accept every model"; it means the loader obtains dimensions from the file and verifies its assumptions instead of baking one file's offsets into the executable.

The names are the stable interface between the generic file table and the model-specific code. The GGUF table does not say "this is layer seven's query projection" as a C pointer; it gives that role a string name. The helper `layer_tensor` formats the expected layer name and passes it to `find_tensor`. Required weights fail loudly when missing. Optional weights return a null pointer, which lets the loader support tied embeddings or optional tensors without guessing from position. A table-order dependency would be fragile: inserting one tensor near the beginning could shift every later index.

After lookup, the loader checks structural relationships. The query matrix's row count must match the head count times the layer's head width; the key matrix must match the number of KV heads times that width. For shared-KV layers, it locates a prior layer with compatible attention settings and points the layer at that cache source. This work is not part of the byte parser. The parser answers "where is tensor X?"; the model loader answers "does this set of named tensors describe a supported Gemma 4?" Keeping those checks separate makes malformed-file errors easier to locate.

E4B's per-layer embeddings show the distinction between finding a tensor and choosing where it is used. Its `per_layer_token_embd.weight` has shape `[10752, 262144]` in the directory and occupies 1,937,768,448 bytes as Q5_K. The file contains this large table, and the model needs one token row at each layer. But the code keeps this table on the CPU side: it marks the model file with `F_NOCACHE`, and advises random access for the row lookups. A tensor can be part of the model without becoming a Metal weight buffer.

We can turn the directory's relative offset into a real file position without reading the tensor. The token table's offset is 55,061,504 bytes. Add the aligned data start, 15,826,752, and its first byte is at file position 70,888,256. The query matrix in block zero has an offset of 2,373,811,200, which puts its first byte at 2,389,637,952. These are addresses inside the map; neither calculation allocates a matrix or copies its contents.

That arithmetic is also a useful debugging step. If a tensor name and shape look right but its calculated range is implausible, check whether the parser added the data start exactly once. Adding it twice shifts every tensor. Forgetting it shifts every tensor back into the directory. Treating the offset as an absolute file position can sometimes appear to work for a tensor whose offset happens to land within the file, but it will read the wrong bytes. A directory dump that prints the data start, relative offset, and computed file position makes such errors visible.

Some small F32 per-layer matrices take another path. The loader reads the original F32 bytes using `pread` with no-cache behavior, converts them to F16 allocations, and gives those copies to the GPU path. It does not need to leave the original F32 pages in the file cache. This is an explicit conversion and allocation; the rest of the large mapped tensors are not transformed as part of discovering their addresses. Chapter 4 returns to why their stored numeric types matter.

This is why "no copies" in the chapter title does not mean the program never copies model-related data. It avoids a full second allocation of the multi-gigabyte file just to load the model. A few selected small matrices are deliberately converted; the tokenizer and metadata use their own runtime tables; and temporary CPU buffers may exist during loading. The important distinction is scale and purpose. The dense quantized weights stay file-backed and are wrapped for Metal, while small data structures or intentionally transformed tensors can have independent storage.

PREPRINT

Which mapped tensors does Metal see?

The file has 720 tensor records, but the GPU does not need every one as a dense Metal buffer. `map_model` sorts tensor records by their data addresses and identifies ranges it should skip. The source comment names the exclusions: expert weights that use a separate cache, the per-layer token table, F32 originals with F16 copies, and key/value tensors for layers that share an earlier layer's cache. It walks the sorted list, groups consecutive included tensors into address ranges, and asks `metal_map` to wrap each range.

Why group neighboring tensors? A single no-copy buffer can span several adjacent payloads; the kernels still receive the individual tensor pointers. If an excluded tensor lies between included runs, it breaks the range. The range selection is an engine policy layered on top of the GGUF directory: GGUF says where all tensors are, while `map_model` says which regions the GPU will use as mapped dense weights.

In the Metal host, `metal_map` rounds the start down and the end up to system page boundaries, then calls `newBufferWithBytesNoCopy`. The buffer covers the same address range. Rounding is necessary because Metal's no-copy buffer requires page-aligned memory and length. The buffer is configured for shared storage, appropriate for Apple silicon's unified memory. A discrete GPU would normally need a separate transfer or allocation; here CPU and GPU can refer to the same physical memory.

Rounding outward means a Metal buffer can cover a few bytes before the first tensor or after the last tensor in a run. The buffer is a page-granular wrapper, but tensor pointers still identify their exact starts within it. Kernels are passed a buffer reference and tensor-specific offsets or pointers according to the host interface; the page boundary is an allocation constraint, not a change to the matrix's logical shape. Two adjacent tensors can share one wrapper while remaining distinct entries in the engine's table.

The Metal host also stores each wrapper in a region list. Later calls can find the buffer containing a mapped address instead of recreating a buffer every time a kernel needs a weight. The map's lifetime and the wrapper's lifetime must overlap all GPU commands that can read it. That is why unmapping has a "GPU must be done with it" comment next to the Metal cleanup path. A no-copy buffer makes ownership more important: releasing the wrapper does not mean a private copy remains behind.

"Wired" describes residency for GPU use. It does not mean an extra copy has been made, nor does it mean the complete GGUF must stay resident. The project's recorded E4B figure is 2.51 GB wired from a 4.72 GB file (README, historical configuration; ENGINE-MATH §11.1). The large CPU-read token table is about 1.94 GB by itself. Shared-layer K/V tensors and other excluded data account for more of the gap. This chapter's W1 script reads the directory only, so it does not reproduce the engine's wired-memory banner. That figure needs an engine load and belongs to a W2 capture; it remains a documented historical value here.

It is tempting to subtract 2.51 from 4.72 and label the entire remainder "unused." That would be too simple. The selected Metal buffers cover groups of neighboring payloads, and their page-rounded ends can include bytes beyond a tensor boundary. Conversely, some records describe data the GPU path does not use. The exact wired total comes from the engine's mapping policy and the ranges it hands to Metal, not just a count of tensor types. The 2.51 GB value is the documented total for this model/configuration, not an estimate from this chapter's directory-only script.

The token table illustrates why residency policy and file layout are different questions. The table is in the GGUF payload area like the other tensors; it has a valid offset and is available through the mapping. It is simply accessed by the CPU, one token row at a time, and omitted from the dense Metal buffer ranges. The code advises random access because adjacent token rows are not necessarily useful next. Leaving it out saves roughly 1.94 GB of GPU-wired pages while preserving access to the same model file.

The unused K/V matrices in shared layers offer a second example. Their names and bytes remain in the GGUF. The loader determines that those layers read an earlier layer's cache instead, and the mapping policy excludes their stored key and value weights. This is not deleting data or editing the file. It is a runtime choice about which recorded tensors participate in this execution path. A different model architecture or a different engine might use them.

Finally, "no-copy" does not mean "no setup." The host creates a Metal buffer object for each chosen region, registers it for lookups, and keeps the buffers alive while commands may use them. It also rounds

PREPRINT

the ranges to page boundaries. Those operations allocate bookkeeping and establish residency, but they do not duplicate the multi-gigabyte tensor bytes. Chapter 6 returns to the buffer lifetime and command submission side; this chapter follows only the route from the file pointer to the wrapper.

Why not read the whole file into one allocation?

A conventional loader can open a file, allocate a buffer as large as the file, read every byte into it, and then copy or upload the needed tensors. That is easy to picture, but it temporarily needs space for both the file-backed pages and the new allocation, and it makes startup do work on bytes that may never be used. For this 4.72 GB file, an additional full-size buffer would be another several gigabytes before model state, activations, or the KV cache are counted.

Mapping lets the operating system manage file pages and gives the parser stable addresses without copying all payload bytes at startup. Selecting Metal ranges then limits which weight regions the GPU must keep wired. These choices solve different parts of the problem: mmap avoids an eager whole-file copy; tensor selection avoids wiring every stored tensor; page rounding satisfies Metal's buffer rules. A reader can use mmap and still make a poor GPU mapping choice, just as a selective Metal map could still sit on top of a separately copied file.

There are tradeoffs. The first access to a cold page may wait for storage, and a mapped pointer is only valid while the mapping remains alive. The operating system may reclaim ordinary file-backed pages when memory is needed; wired pages are treated differently because the GPU has been promised access. The engine's design therefore does not rely on "the entire file is always in RAM." It relies on an addressable file, explicit GPU residency for chosen ranges, and a separate policy for data such as the token table and experts.

For a reader building m2, the important result is modest: produce correct names, shapes, types, byte counts, and offsets. The checkpoint does not yet need to decide which ranges to wire. That separation mirrors the engine and makes failures diagnosable. If the listing differs, inspect cursor movement, type decoding, and alignment. If the listing matches but the GPU cannot access a tensor, inspect selection, page rounding, and Metal lifetime in the next stage.

What does the directory-only probe actually touch?

The W1 helper is deliberately narrower than the engine. It maps the file read-only, checks the four magic bytes, reads the version and counts, then walks metadata entries so it can find the aligned data start. It does not need to build a Python object for the vocabulary's hundreds of thousands of strings. For uninteresting values, it advances over the encoded value; it retains only a short list of architecture and alignment fields. It then reads each tensor record and records its name, dimensions, type ID, and offset.

The distinction between walking metadata and loading weights matters here. GGUF stores tokenizer metadata in arrays before the tensor directory, so the cursor must step over those bytes to reach the records. This can touch several megabytes near the front of the file. It still does not visit the tensor payload regions that begin at the aligned data start. The output's "payload bytes read: 0" describes the script's own operations: its only payload-related work is arithmetic on shapes and offsets.

The helper computes sizes only for the tensor types present in this E4B file and listed in its small table. For quantized types the block size and encoded bytes per block determine the result; for F16, BF16, and F32 the element width does. The script does not decode a single number. It also does not hash payloads, since hashing would read every byte and turn this inspection into a multi-gigabyte file scan. If a new GGUF has a type the probe does not know, it prints that type ID and leaves the size unknown rather than silently making up a size.

Why does a directory-only pass take time if the tensor payload is 4.7 GB? The metadata itself includes tokenizer information, including large arrays of token spellings and merge entries. The cursor must step across those variable-length values to arrive at the tensor records. The Python helper decodes strings briefly while skipping them, then discards them; it does not build a long-lived vocabulary table. The measured work, if a reader times the command, is therefore a parse of the front matter plus ordinary

PREPRINT

file-open and process-start costs. It is not a benchmark of weight loading, tensor decoding, or GPU bandwidth.

This probe is a teaching aid, not a replacement for the engine's production reader. The C code checks that the computed tensor end stays within the file and aborts on a truncated file. The small Python script is used on the known local file for W1 inspection, not as a validator for untrusted files. Its value is that the directory structures are visible in a short program with only Python's standard library.

The script is intentionally conservative about what a successful run means. It proves that this file begins with the expected GGUF marker, that its header and directory can be traversed by this reader, and that the selected metadata and type/shape records can be summarized. It does not prove every payload contains valid quantized values. It does not prove that each tensor name is expected by the engine, that two tensors are numerically compatible, or that a forward pass would succeed. Those checks belong to later stages: the model loader validates names and shapes, a decoder interprets stored blocks, and inference exercises the complete path.

The same boundary helps keep W1 safe and repeatable. The command has one positional argument and prints a directory report. It has no option that initializes Metal or asks for a prompt. It never opens the companion projection file, never allocates the vocabulary as a Python object, and never hashes the weights. To compare a second quantization, pass its path to a second invocation and compare its directory report; the weights still remain unread. To measure output quality or the engine's wired total, switch to the appropriate later experiment and its stricter conditions instead of stretching this probe beyond its purpose.

When adding fields to the probe, keep a boundary between parsing and reporting. Parsing should advance correctly over every metadata type, even when the script does not care about the value. Reporting should select only a few values or summarize counts. This is the same pattern used by the C engine: generic cursor movement first, model-specific lookups afterward. If a field is an array, ask whether the report needs every element. A per-layer setting may need a count or a few examples; the 262,144 token strings do not need to become Python strings just to print the tensor table.

The real code

Where does the mapping become a directory?

This excerpt starts at the actual file mapping. The `MAP_SHARED` comment records why the engine chooses this mapping mode. The cursor reads the fixed header, walks the metadata, then parses each tensor record: name, dimensions, type, and offset. The next excerpt rounds to the aligned data start and fixes up the payload pointers.

```
313 static Gguf gguf_open(const char *path) {
314     Gguf g = {.path = path};
315     int fd = open(path, O_RDONLY);
316     if (fd < 0) die("cannot open %s", path);
317     struct stat st;
318     fstat(fd, &st);
319     g.size = st.st_size;
320     g.base = mmap(NULL, g.size, PROT_READ, MAP_SHARED, fd, 0);  //
      shared: wiring pages for the GPU must not copy them
321     close(fd);
322     if (g.base == MAP_FAILED) die("mmap failed: %s", path);
323
324     Cursor c = {g.base, g.base + g.size};
325     if (rd_u32(&c) != 0x46554747) die("%s: not a GGUF file", path);  //
      "GGUF"
326     uint32_t version = rd_u32(&c);
327     if (version < 2) die("%s: GGUF v%u not supported", path, version);
328     g.n_tensors = (int)rd_u64(&c);
329     g.n_kv = (int)rd_u64(&c);
```

PREPRINT

```
330
331     g.kv = xmalloc(g.n_kv * sizeof(KV));
332     for (int i = 0; i < g.n_kv; i++) {
333         g.kv[i].key = rd_str(&c);
334         g.kv[i].type = rd_u32(&c);
335         g.kv[i].val = c.p;
336         skip_value(&c, g.kv[i].type);
337     }
338
339     g.tensors = xmalloc(g.n_tensors * sizeof(Tensor));
340     uint64_t *offs = xmalloc(g.n_tensors * sizeof(uint64_t));
341     for (int i = 0; i < g.n_tensors; i++) {
342         Tensor *t = &g.tensors[i];
343         t->name = rd_str(&c);
344         uint32_t nd = rd_u32(&c);
345         if (nd > 4) die("gguf: tensor with %u dims", nd);
346         for (int d = 0; d < 4; d++) t->ne[d] = d < (int)nd ? (int64_t)
    rd_u64(&c) : 1;
347         t->type = rd_u32(&c);
348         offs[i] = rd_u64(&c);
349     }
350
351     size_t align = (size_t)kv_num(&g, "general.alignment", 32);
```

Listing 3.1: gemma4.c:313–351

The tensor records use a temporary offset array because data starts only after all records have been read and aligned. Once the cursor reaches the end of the table, the loader rounds up using `general.alignment`. The second excerpt shows the final pointer calculation and range check, followed by lookup by name. Together, these two pieces explain why offsets are relative to the aligned data area and why the model code asks for names rather than positions.

```
352     size_t data_start = ((size_t)(c.p - g.base) + align - 1) / align *
    align;
353     for (int i = 0; i < g.n_tensors; i++) {
354         g.tensors[i].data = g.base + data_start + offs[i];
355         if (data_start + offs[i] + tensor_nbytes(&g.tensors[i]) > g.size
    ) die("gguf: tensor data out of range");
356     }
357     free(offs);
358     return g;
359 }
360
361 static const Tensor *find_tensor(const Gguf *g, const char *name, int
    required) {
362     for (int i = 0; i < g->n_tensors; i++)
363         if (str_eq(g->tensors[i].name, name)) return &g->tensors[i];
364     if (required) die("missing tensor %s", name);
365     return NULL;
366 }
```

Listing 3.2: gemma4.c:352–367

How are architecture settings and layer pointers connected?

The loader reads model dimensions and the shared-KV count from metadata before it starts filling layer fields. It also finds the per-layer token table, disables normal caching on the file descriptor used for auxiliary reads, and marks the large table for random access. This excerpt is the beginning of `load_model`;

it does not show the whole 42-layer loop, but it shows that dimensions and policy come from metadata and named tensor lookup.

```
927 static Model load_model(const Gguf *g) {
928     Model m = {0};
929     Str arch = kv_str(g, "general.architecture");
930     if (!str_eq(arch, "gemma4")) die("architecture '%.*s' not supported
    (need gemma4)", (int)arch.n, arch.s);
931 
932     m.n_layer = (int)kv_num(g, "gemma4.block_count", 0);
933     m.n_embd = (int)kv_num(g, "gemma4.embedding_length", 0);
934     m.n_head = (int)kv_num(g, "gemma4.attention.head_count", 0);
935     m.n_expert = (int)kv_num(g, "gemma4.expert_count", 0);
936     m.n_expert_used = (int)kv_num(g, "gemma4.expert_used_count", 0);
937     m.n_swa = (int)kv_num(g, "gemma4.attention.sliding_window", 0);
938     m.eps = (float)kv_num(g, "gemma4.attention.layer_norm_rms_epsilon",
    1e-6);
939     m.softcap = (float)kv_num(g, "gemma4.final_logit_softcapping", 0);
940     m.embd_scale = sqrtf((float)m.n_embd);
941     if (m.n_layer > MAX_LAYERS) die("too many layers");
942     if (m.n_expert_used > MAX_EXPERT_USED) die("too many experts per
    token");
943     m.n_pl = (int)kv_num(g, "gemma4.embedding_length_per_layer_input",
    0);
944     const int n_kv_layers = m.n_layer - (int)kv_num(g, "gemma4.attention
    .shared_kv_layers", 0);  // layers with a cache
945     if (n_kv_layers < 1) die("no layers with a KV cache");
946     m.n_kv = n_kv_layers;
947     int fd = -1;
948     if (m.n_pl) {
949         if ((fd = open(g->path, O_RDONLY)) < 0) die("cannot open %s", g
    ->path);
950         fcntl(fd, F_NOCACHE, 1);
951         m.pl_tok = find_tensor(g, "per_layer_token_embd.weight", 1);
952         m.pl_model_proj = find_tensor(g, "per_layer_model_proj.weight",
    1);
953         m.pl_proj_norm = f32_data(find_tensor(g, "per_layer_proj_norm.
    weight", 1));
954         if (m.pl_tok->ne[0] != (int64_t)m.n_layer * m.n_pl || m.
    pl_model_proj->ne[1] != (int64_t)m.n_layer * m.n_pl ||
955             m.n_pl % 32)
956             die("unexpected per-layer embedding shapes");
957         // the token table is read a row (a token) at a time: no read-
    ahead of its neighbours
958         const size_t page = (size_t)getpagesize(), a = (uintptr_t)m.
    pl_tok->data / page * page;
959         madvise((void *)a, (uintptr_t)m.pl_tok->data + tensor_nbytes(m.
    pl_tok) - a, MADV_RANDOM);
960     }
```

Listing 3.3: gemma4.c:927–960

How does the GPU get only the needed ranges?

The comment is the compact policy statement: some tensors remain CPU-only or unused, even though their records are in the file. The function sorts data pointers, builds the exclusion set, groups neighboring included tensors, and reports the total bytes mapped. This is the book's CPU-side selection before the Metal wrapper. The Metal function then rounds each selected range to pages and wraps it without copying.

```
4518 static void map_model(const Gguf *g, const Model *m) {
4519     const Tensor **ts = xmalloc(g->n_tensors * sizeof(Tensor *));
4520     for (int i = 0; i < g->n_tensors; i++) ts[i] = &g->tensors[i];
4521     qsort(ts, g->n_tensors, sizeof(Tensor *), cmp_tensor_data);
4522     const Tensor *unused[4 * MAX_LAYERS + 1];
4523     int n_unused = 0;
4524     unused[n_unused++] = m->pl_tok;
4525     for (int l = 0; l < m->n_layer; l++) {
4526         const Layer *L = &m->layers[l];
4527         const Tensor *t[4] = {L->kv_src != l ? layer_tensor(g, l, "
    attn_k.weight", 0) : NULL,
4528                               L->kv_src != l ? layer_tensor(g, l, "
    attn_v.weight", 0) : NULL,
4529                               m->n_pl ? layer_tensor(g, l, "inp_gate.
    weight", 0) : NULL,
4530                               m->n_pl ? layer_tensor(g, l, "proj.weight"
    , 0) : NULL};
4531         if (t[2] == L->pl_gate) t[2] = NULL;  // (used as it is)
4532         if (t[3] == L->pl_proj) t[3] = NULL;
4533         for (int j = 0; j < 4; j++)
4534             if (t[j]) unused[n_unused++] = t[j];
4535     }
4536     char *expert = xcalloc(g->n_tensors, 1);  // (or unused)
4537     for (int i = 0; i < g->n_tensors; i++) {
4538         for (int l = 0; l < m->n_layer; l++) {
4539             const Layer *L = &m->layers[l];
4540             if (L->router && (ts[i] == L->exp_gate || ts[i] == L->exp_up
     || ts[i] == L->exp_down)) expert[i] = 1;
4541         }
4542         for (int j = 0; j < n_unused; j++) expert[i] |= ts[i] == unused[
    j];
4543     }
4544     size_t dense = 0;
4545     for (int i = 0; i < g->n_tensors;) {
4546         if (expert[i]) { i++; continue; }
4547         const uint8_t *start = ts[i]->data, *end = start;
4548         for (; i < g->n_tensors && !expert[i]; i++)
4549             if (ts[i]->data + tensor_nbytes(ts[i]) > end) end = ts[i]->
    data + tensor_nbytes(ts[i]);
4550         if (!metal_map(start, end - start)) die("metal: cannot map the
    weights");
4551         dense += end - start;
4552     }
4553     fprintf(stderr, "metal: %.2f GB of weights, wired\n", dense / 1e9);
4554     free(ts);
4555     free(expert);
4556 }
4557 #endif
```

Listing 3.4: gemma4.c:4518–4557

Here is the small Metal-side operation that the last paragraph described. The pointer and size supplied by C are rounded outward to page boundaries, then passed directly to Metal's no-copy constructor. The buffer record is retained for later lookup and cleanup. The wrapper does not copy tensor bytes into a new buffer.

```
1726 int metal_map(const void *p, size_t size) {
1727     const size_t page = (size_t)getpagesize();
```

```
1728      const char *start = (const char *)((uintptr_t)p / page * page);
1729      const size_t len = ((uintptr_t)p + size + page - 1) / page * page -
          (uintptr_t)start;
1730      id<MTLBuffer> b = [device newBufferWithBytesNoCopy:(void *)start
          length:len
1731                                                  options:
          MTLResourceStorageModeShared deallocator:nil];
1732      if (!b) return 0;
1733      add_region(start, len, b);
1734      return 1;
1735  }
```

Listing 3.5: `gemma4_metal.m`:1726–1735

Experiment

Can we inspect the real GGUF layout without loading a model into the engine? This W1 probe uses Python's standard library and a read-only memory map. It reads the header, the metadata keys and values needed to step through the metadata table, and all tensor records. It computes byte counts from each recorded shape and type. It does not hash, decode, copy, or otherwise read tensor payload bytes; it never invokes the engine or Metal.

Run it with the local E4B file:

```
python3 code/gguf_probe.py "$MODEL"
```

The capture below used the local E4B file on 2026-10-22. Inspect the file size, count and placement of header/directory data, alignment, architecture fields, type totals, and the 1.94 GB per-layer token table. The values are declared layout sizes, not a record of pages resident in memory. The complete command and output are saved in `code/expected/ch03-w1.txt`.

Experiment W1

```
GGUF v3: 720 tensors, 56 metadata entries
file size: 4,715,416,704 bytes; directory ends at 15,826,723; data starts at 15,826,752;
     alignment: 32
general.architecture = gemma4
general.name = Gemma-4-E4B-It
gemma4.block_count = 42
gemma4.embedding_length = 2560
gemma4.attention.shared_kv_layers = 18
gemma4.embedding_length_per_layer_input = 256
tensor types: BF16=1 (55,050,240 declared bytes), F32=423 (222,464,168 declared bytes),
    IQ4_XS=234 (2,047,180,800 declared bytes), Q4_K=19 (392,970,240 declared bytes),
    Q5_K=43 (1,981,923,328 declared bytes)
per_layer_token_embd.weight: Q5_K shape=[10752, 262144] offset=55,061,504 bytes
    =1,937,768,448
blk.0.attn_q.weight: IQ4_XS shape=[2560, 2048] offset=2,373,811,200 bytes=2,785,280
payload bytes read: 0 (header, metadata, and tensor directory only)
```

The type totals are a useful inventory, not all the file's bytes: they do not include the GGUF header and metadata, alignment padding, or any type the probe does not know how to size. The per-layer table is a notable example: it occupies 1.94 GB on disk but is intentionally not made into a Metal weight buffer. To inspect the wired total, the engine must load the model. That W2 check is a reader experiment: in a scratch copy, load the model and inspect the wired total beside the W1 file-size inventory. The source mapping predicts that the wired total excludes the per-layer token table and unused shared-layer K/V data; the documented historical reference is 2.51 GB wired from a 4.72 GB file (README, "The model file" / "What the E4B needs"; ENGINE-MATH §11.1).

Pitfalls from the project notes

Pitfall

A mapped model can quietly become a second in-memory model. In the project's documented 26B mapping work, private file mappings caused copy-on-write pages when the GPU wired them; the result was 7.8 GB of compressed anonymous memory and swapping. The fix recorded in the notes was to use `MAP_SHARED` for the dense mapped weights and to keep the mapping as a wrapper over those pages. This history is described in `README-26B.md`, "How it works" (dense weights), and the reader's source comment at `gemma4.c:320`. The symptom was memory pressure; the cause was a private copy of mapped pages; the fix preserved shared backing.

Exercises

Exercises

1. **Read.** Draw the bytes of one tensor record in order: name, dimension count, dimensions, type, offset. Explain why the offset alone is not a file address.
2. **Read.** The probe reports `blk.0.attn_q.weight` as `[2560, 2048]`. Which value is the input-feature row length? Which is the number of output rows?
3. **Change.** Add one metadata key to `code/gguf_probe.py` and print it without retaining large vocabulary arrays in memory. Check the updated probe against the same file.
4. **Change.** Extend the probe to print each tensor's absolute file position as `data_start + offset`. Verify that the first data address is aligned and that the final tensor end does not exceed the file size.
5. **Measure.** Compare a cold and warm run of the directory-only probe. Record elapsed time and explain why this measures header and metadata access, not model loading or weight bandwidth.
6. **Read.** Follow the exclusion list in `map_model`. For each excluded group, say whether it is unused, CPU-read, replaced by a converted copy, or supplied by a different cache.

Rebuild checkpoint

Rebuild checkpoint

Milestone m2: list tensors from the GGUF directory. Implement `steps/m2_gguf.c` so it reads the header, metadata, aligned data start, and tensor records, then prints each tensor's name, type, shape, and declared byte count. Compare its table with `code/snapshot/tools/gguf_cmp.py` on the same file. The check concerns directory parsing and size calculations; it does not require reading tensor payloads or loading the model.

Chapter 4

Small numbers: fp16 and quantised blocks

The question

How can a weight stored in only a few bits still be useful in a matrix calculation?

The idea

What does a half-precision number keep?

The same value can be written as decimal text, as a 32-bit float, or in a compact 16-bit pattern. In binary16 (usually called fp16), the 16 bits are a sign, a 5-bit exponent, and a 10-bit fraction. For example, 1.0 is `0 01111 0000000000`, 0.5 is `0 01110 0000000000`, and -2.0 is `1 10000 0000000000`. The exponent has a bias of 15: the stored exponent 15 means an actual exponent of zero. A normal value has an implicit leading one before the fraction. Zeros, tiny subnormals, infinities, and NaNs have special exponent patterns, so conversion code needs more than chopping a float in half.

BF16 also uses 16 bits, but keeps the sign and 8-bit exponent of fp32 and only 7 fraction bits. It has about the same range as fp32, with less precision than fp16. In this model file the per-layer projection is BF16, while many normalisation values are F32. The C loader converts selected F32 matrices to F16 copies. On the GPU, the engine's matmul inputs and weight tiles are fp16; the dot-product accumulators are wider so many products can be added without rounding every partial sum back to fp16. The stored format, the multiply format, and the accumulator format are separate choices.

These three fp16 patterns make the fields visible. The value 1.0 is `0x3c00`, 0.5 is `0x3800`, and -2.0 is `0xc000`. In each, the exponent field is respectively 15, 14, and 16. A BF16 encoding of 1.0 is `0x3f80`; its exponent field has eight bits, so it can represent much larger and smaller magnitudes than fp16. For ordinary values fp16 has more fraction bits, which gives finer spacing near one. "16-bit" alone therefore does not say what range or precision the stored values have.

What does the C conversion have to preserve? A normal fp16 number's exponent is rebased from 15 to fp32's 127, and its ten fraction bits are shifted into the high part of fp32's 23-bit fraction. The sign moves to bit 31. Subnormals have no leading one, so the converter shifts their fraction until its leading one appears and adjusts the exponent. The reverse conversion rounds discarded bits to nearest-even. That rounding rule matters at halfway cases: repeatedly converting through fp16 should not systematically round every tie upward.

The engine precomputes all 65,536 possible half patterns into a lookup table when it initialises its numeric tables. The CPU row decoder can then convert a stored half with one indexed lookup. BF16 needs no table: its 16 bits are already the high half of the fp32 pattern. On Metal the language has a native `half` type, so kernels load half scales and place decoded weights into half tiles directly.

The half-to-float helper handles the cases the bit picture leaves out. It copies the sign, rebiases normal exponents, preserves infinity and NaN payload bits, returns signed zero, and normalises subnormals. Chapter 4's code section shows this conversion beside the row decoders that use the resulting values.

PREPRINT

Which plain floating types does this GGUF use?

The E4B tensor table has both F32 and BF16 entries. F32 stores one IEEE-style 32-bit value per weight: one sign bit, eight exponent bits, and 23 fraction bits. Its byte map is simple because every row value occupies four bytes and no block scale is shared.

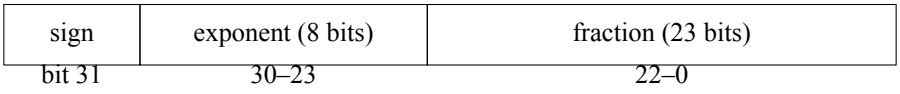


Figure 4.1: F32 has no per-block metadata: every weight is a complete 4-byte floating-point value.

The W1 sample uses the first value of `output_norm.weight`. Its four bytes are `00 00 fd 40`; as a little-endian F32 value they decode to 7.90625. The tensor has 2,560 values and 10,240 declared bytes, exactly four bytes per value. This is a vector of normalization weights, not one of the large quantized matrices.

BF16 stores two bytes per value. It keeps F32's sign and eight exponent bits, then keeps the top seven fraction bits. The missing low 16 bits of the F32 representation are not stored, so converting BF16 to F32 simply places its bits in the upper half of a 32-bit word.

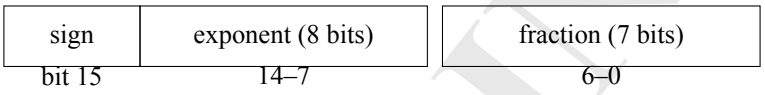


Figure 4.2: BF16 uses the upper half of an F32 bit pattern, retaining its range and fewer fraction bits.

The first W1 value in `per_layer_model_proj.weight` is stored as bytes `0b ba`, which form the 16-bit pattern `0xba0b`. Its decoded F32 value is -0.00053024292. The tensor contains 27,525,120 BF16 values and 55,050,240 bytes. Unlike IQ4_XS or the K formats, there is no extra scale to unpack: each stored BF16 value directly supplies one weight.

How do 32 weights fit in a few bytes?

Imagine 32 weights that are all near zero. Storing each as F16 takes 64 bytes. Q8_0 stores one two-byte fp16 scale and 32 signed bytes: 34 bytes total. Q4_0 stores the two-byte scale and 32 four-bit codes packed into 16 bytes: 18 bytes. Each code represents a value from -8 through 7 after subtracting 8, then multiplying by the shared scale. Q4_1 instead uses a scale and minimum, allowing its 16 codes to cover a range that need not be centered on zero; it uses 20 bytes.

For Q4_0, decode each code with the block scale using the allowed rule:

$$w = d\,(n - 8), \qquad n \in \{0, \ldots, 15\}.$$

For a four-value example, let the scale be 0.5 and the codes be 8, 9, 7, and 10. The decoded weights are 0, 0.5, -0.5, and 1. In a dot product, the shared scale can be applied once after accumulating the products with the shifted integer codes.

Here is the byte map for the beginning of that block. The scale occupies two bytes in little-endian order; byte 2 holds codes for positions 0 and 16, byte 3 for positions 1 and 17, and so on. The low nibble is the earlier position. If byte 2 is `0x98`, its low nibble is 8 and its high nibble is 9: the first weight is zero and the weight at position 16 is half the scale.

Q8_0 is more direct: after its two-byte fp16 scale come 32 signed bytes, one for each successive position. Its scale is usually chosen from the largest magnitude in the block, and each code is a rounded signed integer. Q4_0 uses fewer bits per code, with a symmetric zero point at code 8. Q4_1 adds a minimum so the encoded range can be offset instead; that helps when a block's values are not balanced around zero, at the cost of storing another fp16 number.

PREPRINT

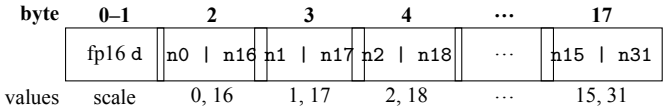


Figure 4.3: Q4_0 uses 18 bytes for 32 values. Each packed byte supplies two codes sixteen positions apart.

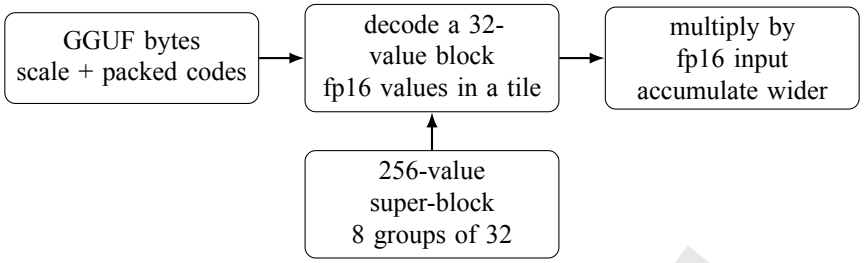


Figure 4.4: A quantised matrix stays packed in the file until a compute path needs a row or tile.

Why add super-blocks?

K formats share a larger scale across 256 values but retain eight local groups of 32. This reduces overhead and adapts to local variation. Q4_K uses 144 bytes per 256 weights: two fp16 super-scales, 12 packed bytes for eight 6-bit local scales and minima, and 128 nibble bytes. Q5_K adds one high bit per code and uses 176 bytes. IQ4_XS also uses 136 bytes per 256 values: a base fp16 scale, packed local 6-bit scales, and 128 code bytes. Its 4-bit codes index a nonuniform 16-value table, with more finely spaced values near zero.

The storage cost includes metadata, so “4 bits” does not mean exactly half a byte per weight. Here are the actual byte counts derived from the block layouts:

Format	Weights	Bytes	Bits/weight
F16 / BF16	256	512	16
Q8_0	256	272	8.5
Q4_0	256	144	4.5
Q4_1	256	160	5
Q4_K	256	144	4.5
Q5_K	256	176	5.5
Q6_K	256	210	6.5625
IQ4_XS	256	136	4.25

The table compares storage, not accuracy. A smaller block can save memory traffic but quantisation error changes the values. The model’s output depends on many layers of calculations, so visual inspection of a generated answer is not an accuracy measure. The project notes compare perplexity, KL divergence, and top-token agreement when they test weight variants.

The row width turns these rates into a concrete byte count. The first attention projection in the captured E4B file has 2,560 input values per row and 2,048 output rows. IQ4_XS groups each row into ten 256-value super-blocks, so each row occupies 1,360 bytes. The same row in F16 would need 5,120 bytes. The GGUF directory reports 2,785,280 bytes for the full matrix, exactly 2,048 rows of 1,360 bytes. This arithmetic needs only its shape and type; it does not read the payload.

The entire file gives another scale comparison. Chapter 3’s directory capture counted 2,047,180,800 bytes of IQ4_XS tensors, 392,970,240 bytes of Q4_K, and 1,981,923,328 bytes of Q5_K. Those declared totals do not include metadata and padding. The token table alone is Q5_K and uses 1,937,768,448 bytes, which is why a model-wide average is not the same as the format of its main dense matrices. The quantisation name describes a collection of tensors; it does not mean every tensor uses one block type.

Here is the complete type mix reported by the W1 probe. The percentages use the full 4,715,416,704-byte file as the denominator, so they do not add to 100

PREPRINT

Type	Bytes/weight	Tensor MB	File share
F32	4	222.464	4.72%
BF16	2	55.050	1.17%
IQ4_XS	0.53125	2,047.181	43.41%
Q4_K	0.5625	392.970	8.33%
Q5_K	0.6875	1,981.923	42.03%

The block-format bytes per weight include scales and code metadata averaged over complete blocks: 136/256, 144/256, and 176/256 bytes respectively. F32 and BF16 have no block overhead, so their entries are exactly four and two bytes per value. The probe derives these totals from tensor shapes and types; tensor MB uses decimal megabytes and the share denominator is the 4.72 GB file size.

What does a 256-value Q4_K super-block contain?

Q4_K starts with two fp16 values, called the base scale and base minimum. Twelve bytes follow, encoding eight local scale/minimum pairs in 6 bits each. The last 128 bytes carry four groups of packed nibbles. Each pair of neighbouring 32-value groups shares the same 32-byte run: the low nibbles belong to the even group and the high nibbles to the next group. The offsets are therefore deliberate: block 0's codes begin at byte 16, block 1 also uses those same bytes' other nibbles, and block 2 starts at byte 48.

The local parameters are not stored as eight conventional 16-bit pairs; that would cost 32 bytes. Instead, six-bit values are packed densely into the 12 scale bytes. For groups 0 through 3, each scale is the low six bits of one byte and each minimum is the low six bits of a byte four places later. For groups 4 through 7, the low four bits come from a later byte and the top two bits are borrowed from the first eight bytes. The decoder's shifts and masks reconstruct those 6-bit values before it touches the code nibbles.

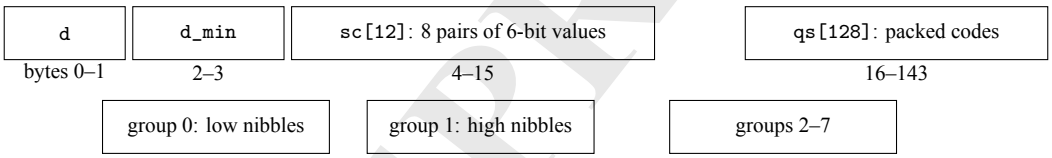


Figure 4.5: Q4_K stores 256 values in 144 bytes. Q5_K inserts 32 high-bit bytes before the code nibbles.

Why bother with a minimum? With only a scale and unsigned codes, a block's levels start at zero. Q4_K's per-group minimum shifts them, so one group can represent a range whose low end is below zero. This spends four bytes on the two super-block fp16 values plus the packed local metadata and lets each group adapt both its step and offset. Q5_K reuses the same scales and minima but adds one high bit for each of the 256 codes; its 32 extra bytes make 5-bit levels.

The first Q4_K super-block in the tied embedding begins with the header bytes `c5 06 66 12 ad bf bd a1 ...`. The first two little-endian fp16 fields decode to a base scale of 0.000103294849 and a base minimum of 0.000781059265. The first group's local scale and minimum are both 45, reconstructed from the six-bit fields in the 12-byte metadata area. Its first four codes are 0, 9, 14, and 6. Applying that shared scale and minimum gives -0.0351476669, 0.00668674707, 0.0299280882, and -0.00725805759. The W1 decoder reads 144 payload bytes for this super-block and reports the same values.

The Q5_K high-bit array is arranged by value position rather than by a separate byte run for each group. For value position `j`, bit `ib` of `qh[j]` is the extra bit for group `ib`. The decoder takes the selected four-bit nibble and ORs in that bit shifted above it. This means the four-bit code data is laid out exactly as in Q4_K, while one extra plane supplies the fifth bit. Q5_K pays 32 bytes for that plane, averaging one extra bit per weight.

The first Q5_K super-block in `per_layer_token_embd.weight` has base scale 0.000242114067 and base minimum 0.00651550293. Group zero has local scale 63 and minimum 63. Its first four reconstructed integer codes, after the high-bit plane is combined with the nibble, are 26, 30, 27, and 28. The decoded values are -0.0138938427, 0.0471189022, 0.00135934353, and 0.0166125298. The W1 script reads 176 bytes for this one super-block; the full tensor remains untouched.

UNEDITED

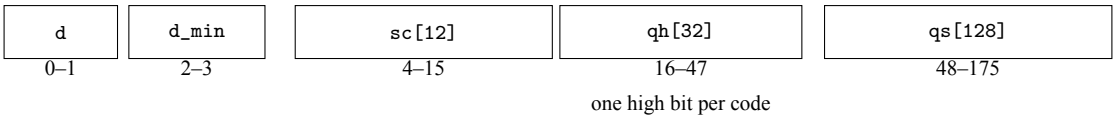


Figure 4.6: Q5_K adds a 32-byte high-bit plane to Q4_K's super-block, before the 128 nibble bytes.

What happens to a Q4_1 block?

Q4_1 is the smaller 32-value format that stores both a scale and a minimum. Its 20-byte map has a two-byte scale, a two-byte minimum, then 16 packed nibble bytes. The low and high nibbles again map to positions separated by sixteen. Each code becomes a nonnegative level times the scale, then the minimum is added. Unlike Q4_0's center code, Q4_1's zero value is generally not one of the representable levels; the minimum offsets the full range.

That extra two-byte field can help a block whose values cluster around a nonzero centre. The trade-off is visible without timing anything: the block holds 32 values in 20 rather than 18 bytes. Q4_1 is present in the engine's decoder but not in this particular E4B file's type totals. It appears here because the CPU and Metal source support it and because it is a useful contrast to the symmetric Q4_0 layout.

Why is IQ4_XS not ordinary Q4?

IQ4_XS also holds 256 values in a 136-byte super-block, but its 16 code levels are not evenly spaced. The shared table is `-127, -104, -83, -65, -49, -35, -22, -10, 1, 13, 25, 38, 53, 69, 89, 113`. The small gaps near zero give a quantiser more choices where many weights cluster, while larger magnitudes have wider spacing. A 4-bit code indexes this table; it is not interpreted as a signed four-bit integer.

For a scale-neutral comparison, place 16 uniform levels across the same endpoints, -127 and 113. Near zero those levels step by about 16; the IQ4_XS values have smaller local gaps there. That is the useful difference: if most weights are close to zero, a nearby table entry can represent them with less absolute error while the same four-bit index still reaches the large values. The following values are the two grids, not a claim about a particular tensor's measured error:

Grid	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Uniform	-127	-111	-95	-79	-63	-47	-31	-15
IQ4_XS	-127	-104	-83	-65	-49	-35	-22	-10
Grid	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Uniform	1	17	33	49	65	81	97	113
IQ4_XS	1	13	25	38	53	69	89	113

The spacing is intentionally nonuniform; the codebook has more choices around zero and wider steps toward the extremes. A lookup table turns each nibble into one signed level, after which the group's scale supplies its magnitude. This gives 4.25 bits per stored value including metadata, with the accuracy trade-off assessed by model-level comparisons rather than by the grid alone.

Its super-block header is 8 bytes: a two-byte fp16 base scale, a two-byte packed high-scale word, and four bytes of low-scale nibbles. Each of the eight 32-value groups has a six-bit local scale. Four bits are in the corresponding nibble of the four-byte low array and two bits come from the high word. Subtracting 32 makes this local scale signed, so a group may reverse the sign of the base scale. Then come eight separate 16-byte code groups, totaling 128 bytes.

Every code byte has two lookups. Its low nibble selects the level for one of the group's first 16 values; its high nibble selects the level for the corresponding value in the second half. So the first two bytes `0x89`, `0x76` contain codes 9, 8, 6, and 7 when read as low/high pairs. The first four weights in row order instead use the low nibbles of code bytes 0 through 3. This distinction is easy to miss when looking at a hexadecimal dump because byte order and value order are not the same thing.

PREPRINT

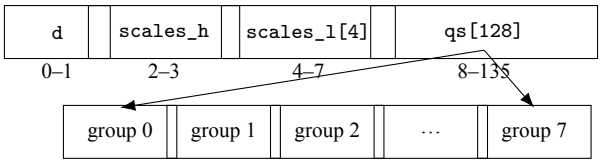


Figure 4.7: IQ4_XS keeps each group's 16 code bytes together, while a compact header supplies its base and local scales.

For the captured first group, the first two bytes `b9 02` are little-endian fp16 bits `0x02b9`, approximately $4.15444374 \times 10^{-5}$. Its six-bit local scale is zero, so subtracting 32 gives -32 and the effective multiplier is approximately -0.001329422. The first four weights use the low nibbles of `89 76 9a c6`: indices 9, 6, 10, and 6 select levels 13, -22, 25, and -22 from `kvalues_iq4nl` (`gemma4.c:424, 539--540`). Multiplying by the effective scale gives the first four decoded values in row order: -0.017282486, 0.029247284, -0.033235550, and 0.029247284. The high nibbles supply the corresponding values in the second half of the 32-value group.

This is a useful example of why format names alone do not decode data. The same nibble has a different meaning in Q4_0, Q4_K, and IQ4_XS. A decoder must know not only the number of code bits, but also where its scale is stored, how local scales are packed, how nibbles are assigned to positions, and whether a code is a signed integer or a table index.

What does smaller storage buy, and what can it cost?

At a fixed memory bandwidth, reading fewer bytes can make a matrix operation faster. It also makes a model file smaller and can leave more room for the operating system, activations, and cache. But a quantiser approximates the original weights. Whether that approximation changes the answer depends on where the affected matrix sits, the input values it receives, and how downstream layers respond. A local weight reconstruction error is not itself a user-visible quality score.

The project began with a Q4_0 file and later used an IQ4_XS variant. The README says the latter file was only 2.5% smaller, largely because the 1.94 GB token table remained Q5_K in both. The documented decode-step bytes fell from 2.77 GB to 2.61 GB. Historical kernel captures showed the IQ4_XS gate-and-up matvec reading 27.8 MB in 306 microseconds, while the Q4_0 version read 29.5 MB in 330–350 microseconds. Small IQ4_XS projections were slower per byte, and end-to-end noise hid the difference. These are README results from the recorded sessions, not new measurements in this chapter (snapshot `README.md`, "The model file" and "Results").

Measured, see Chapter 20. The project README reports mean KL divergence 0.000012 against llama.cpp with the fp16 cache, at context 512 over four chunks, and 99.9% same top token for the IQ4_XS E4B file. Its one-token-at-a-time KL was 0.000010. These are historical measured results from the README's evaluation text, not measurements made for this chapter. The README also cautions that the Q4_0 perplexity was slightly lower on that text, but within the stated uncertainty; a fair ranking would need a higher-precision reference.

Quantisation can also be selective. The 84 small per-layer matrices are read every token, so the engine stores F16 working copies instead of keeping their F32 originals in the GPU mapping. The project tried Q8_0 copies: half the F16 storage again, with a reported 2% speed gain, but mean KL divergence rose about 100-fold and the top token changed 0.7% of the time. That is why the pitfall box says the smaller representation was rejected. The best format is a measurement decision about a tensor's role, not a contest to minimise every byte.

Why are the codes grouped into blocks?

A code by itself has no physical weight value. It becomes a value only after the decoder knows its block's scale and, for the K formats, its local minimum or extra code bits. Grouping values lets that metadata be shared. With one scale for 32 values, Q4_0 spends two bytes on the scale and 16 on the codes. That is

PREPRINT

18 bytes rather than 64 bytes for 32 half values. A K super-block shares two base half scales across 256 values, then assigns local parameters to eight 32-value groups. The metadata cost is larger in total but smaller when averaged over 256 codes.

Smaller groups adapt more closely to local variation, but require more metadata reads and more scale work. Larger groups reduce metadata overhead, but one outlier can set a scale that is poor for the rest of that group. K formats compromise: a 256-value container reduces repeated base values, while eight local 32-value settings keep the range flexible. IQ4_XS uses a related 256-value container, with a compact six-bit signed scale for each 32-value group and a shared codebook. The block size is therefore a design choice that balances metadata, local fit, and decoding work.

The same arrangement suits matrix kernels. Weights are consumed in contiguous runs along the input-feature dimension. A threadgroup can load a block's header once, decode its codes into a short-lived tile, and reuse that tile while processing several activation rows. The kernel need not materialise a row's complete floating representation before it starts calculating. It can move through one block at a time, preserving the compact file layout and limiting scratch space.

This is also why block boundaries must line up with the row stride. The W1 attention projection has 2,560 values per row, exactly ten 256-value super-blocks. Each IQ4_XS row occupies 1,360 bytes, including all ten headers and code arrays. The tensor directory's 2,785,280 bytes equal 2,048 such rows. A shape alone does not give the byte count; the block format contributes the per-row storage rule, and every row must end on a complete super-block boundary for this tensor.

Why is accuracy evaluated separately from storage?

The code table describes possible reconstructed values, not how close a particular model's logits will be to a reference. A local error in a weight is multiplied by the activation that reaches that weight. The resulting partial products are summed with contributions from other weights, then passed through later projections, nonlinearities, normalisation, and attention. A small change can vanish in later calculations or alter a close competition between two output tokens. The storage table cannot predict which outcome occurs.

The project therefore records output comparisons separately from byte counts. Its README reports the fp16-cache run's mean KL divergence of 0.000012 against llama.cpp at context 512 over four chunks, with 99.9

A useful way to read such a report is to ask what changed between the two compared paths. Was the GGUF identical? Was the KV cache precision the same? Were prompts, chunk boundaries, and token positions matched? Did both sides evaluate the same distribution? Those questions matter because an apparent weight-format difference can come from cache rounding or a data race instead. One historical token-buffer race produced KL 0.83 in a decode-path run, despite the correct result being around 0.00001. The pitfall later in the chapter shows why a numerical result needs a validated execution path behind it.

Where does the engine unpack the weights?

The file reader from Chapter 3 records each tensor's type and pointer; it does not turn all payloads into arrays of floats. CPU reference code decodes a row through `dequant_row`. For a large matrix multiply, Metal decodes small weight tiles into fp16 threadgroup memory, reuses them across activation rows, and accumulates products. Matvec kernels also decode blocks inside the kernel. The packed GGUF remains the stored source; temporary decoded tiles are working data. Expanding every quantised tensor to F16 would multiply its payload footprint by almost four relative to `IQ4_XS`, require several extra gigabytes, and make each kernel read the expanded copy. Instead, a 32-weight block is decoded only as the active tile is built, then reused and overwritten. The exception is deliberate: the 84 small F32 per-layer matrices have F16 working copies because they are consumed every token; that measured choice is discussed below.

The engine snapshot supports a wide range of formats: F16, BF16, Q8_0, Q4_0, Q4_1, Q4_K, Q5_K, Q6_K, IQ4_NL, IQ4_XS, and IQ3_S. The E4B file inspected in Chapter 3 contains BF16, F32, IQ4_XS,

PREPRINT

Q4_K, and Q5_K. The remaining quantised types are decoder support, not evidence that this particular file uses them.

Reading a K-format scale table

The 12 metadata bytes in a K-format super-block hold eight local scale/minimum pairs, not twelve independent numbers. Each local field is six bits. For groups zero through three, the scale comes from the low six bits of byte `sc[ib]`, and the minimum from the low six bits four bytes later. For groups four through seven, the low four bits are in a later byte and the remaining two bits are borrowed from another byte. The C decoder's masks and shifts are the unpacking recipe; the fields are laid out this way to use exactly 96 bits for sixteen six-bit values.

In the first Q4_K sample, the metadata begins `ad bf bd a1 ad bf bf f7 49 0d 85 7d`. Group zero's scale is the low six bits of `ad`, or 45. Its minimum comes from the low six bits four bytes later, also 45. Group one reads `bf` in those two positions, giving 63 and 63. Those small integers multiply the two fp16 super-scales to make the group's effective scale and minimum. Treating `ad` as a complete byte value would incorrectly give 173; masking to six bits is essential.

The Q4_K code values are unsigned nibbles. The group's minimum shifts the range, so code zero can reconstruct to a negative weight and code nine can reconstruct to a positive one. In the W1 sample, the first four codes are 0, 9, 14, and 6; applying the base fields and local parameters gives -0.0351476669, 0.00668674707, 0.0299280882, and -0.00725805759. This is a complete small decode: two half values, two six-bit local fields, and four codes. It also explains why the metadata is not decorative overhead. Without the minimum, the low code could only represent zero or a positive level under this unsigned interpretation.

Q5_K keeps the same scale/minimum table and nibble run, then adds 32 bytes of high bits. The decoder finds the nibble for a logical position, extracts the corresponding bit for that group from the high-bit plane, and joins them into a five-bit code. The extra plane is one bit per weight, which is why the format adds 32 bytes to the 256-value block. The first W1 group reconstructs codes 26, 30, 27, and 28; after the same local scale/minimum operation, these give -0.0138938427, 0.0471189022, 0.00135934353, and 0.0166125298. When checking a decoder, compare reconstructed integer codes before comparing floats: this separates a nibble/high-bit indexing error from an fp16 scale or rounding difference.

Why the IQ4_XS table bends toward zero

A uniform four-bit quantiser assigns the same spacing to every neighbouring code. That spends as many choices on a large interval near the extremes as it does near zero. Neural network weight distributions often put many values close to zero, so a codebook can reduce error for common small magnitudes by putting more representable values there and widening steps where values are less common. IQ4_XS uses a shared sixteen-entry non-linear table, then multiplies its selected signed level by a group scale. The lookup is a table access, not a signed-nibble conversion.

The table comparison earlier in this chapter is a picture of the code locations, not a claim about the model's distribution. For example, around zero the IQ4_XS levels include -10, 1, and 13, while an evenly spaced grid across the same endpoints has nearby values around -15 and 1, then 17. That gives the nonlinear grid closer choices on both sides of zero. Near the extremes, the IQ table uses larger gaps. This design can lower average error when small weights are common, while preserving reach to larger magnitudes with the same four code bits.

There is a second level of adaptation in IQ4_XS: each 32-value group gets a signed six-bit local multiplier. Four low bits are packed in the four-byte low-scale array, and two high bits per group are packed in a 16-bit field. The stored six-bit number is biased by 32 before it multiplies the super-block fp16 scale. A negative local multiplier is valid and flips all sixteen table values in each half-group. This lets the quantiser reuse one table while fitting groups with different magnitudes and signs. It is also why the first sample's local scale code zero yields a negative effective multiplier.

PREPRINT

The code bytes are arranged as sixteen pairs: the low nibble selects the first half of a 32-value group and the high nibble selects the matching position in the second half. Thus byte zero contains the code for positions zero and sixteen, byte one for positions one and seventeen. A row decoder that emits low then high for every byte will produce the wrong order. The C code writes the two halves separately, and the W1 hand calculation uses low nibbles from consecutive bytes for the first four values.

What the plain floating formats trade

F32 and BF16 appear beside the quantised formats in the same tensor table, but they have no block scale or code index. F32 keeps every 32-bit pattern, including its exponent and fraction; BF16 keeps the high sixteen bits of that pattern. This makes BF16's conversion to F32 simple and preserves roughly the full F32 magnitude range, but the shorter fraction makes adjacent representable values farther apart. The captured BF16 value is a small negative number, so the retained exponent is important: changing it into a narrow-range integer format would require a separately selected scale.

The snapshot also understands F16, but the E4B file's primary tensor table contains BF16 and F32 rather than F16 weights. F16 still matters on the compute side. The loader converts selected F32 matrices into F16 working copies, while Metal activations and decoded weight tiles are half values. The multiply accumulates in a wider type. Those boundaries are explicit: storage can be F32, BF16, or packed integer codes; the tile can be F16; the accumulator can be F32. A format label on disk does not by itself describe all later arithmetic.

The first F32 sample, `output_norm.weight`, begins with bytes `00 00 fd 40`. The least significant byte comes first in the file; interpreted as one little-endian F32 word, these bytes are 7.90625. The BF16 sample begins `0b ba`, forming bits `0xba0b`. Putting these bits in the upper half of an F32 word produces -0.00053024292. Both are scalar decodes: no block boundary, nibble order, scale, or codebook is involved. Their purpose in a mixed-format GGUF is to keep selected tensors in a format that matches their numerical role and loading path.

What the size table says about the whole file

The per-block table is useful for a kernel, while the whole-file table shows what dominates storage. The W1 probe counts tensor payload bytes by type from their shapes and GGUF type IDs. It finds 2,047,180,800 IQ4_XS bytes and 1,981,923,328 Q5_K bytes: together they account for most of the tensor payload. The 392,970,240 Q4_K bytes are mainly the token embedding, and the F32/BF16 tensors occupy much smaller shares. The percentages use the complete file size, so the small remainder consists of metadata, alignment, and other non-payload bytes.

A type's share of bytes is not the same as its share of values or compute time. A large embedding may be read for output scoring but not used in the same way as every attention projection; some per-layer matrices are read on each token and copied to F16; some listed K/V tensors are never read by shared-KV layers. Therefore the type table describes disk payload, not wired memory or bytes transferred in one inference step. Chapter 3 traces which tensors the engine maps, and the later kernel chapters measure the paths that actually read them.

The real code

How are the 16-bit formats interpreted?

How does a half become a C float? The CPU helper handles normal values, signed zero, subnormals, infinities, and NaNs by rebuilding the F32 bit pattern. This is the point where the two-byte fp16 scale in each quantised block becomes an ordinary value the decoder can use.

```
372 static float half_to_float(uint16_t h) {
373     uint32_t sign = (uint32_t)(h >> 15) << 31, exp = (h >> 10) & 0x1f,
      man = h & 0x3ff, bits;
```

PREPRINT

```
    if (exp == 0x1f) {
        bits = sign | 0x7f800000 | (man << 13);  // inf / nan
    } else if (exp != 0) {
        bits = sign | ((exp + 112) << 23) | (man << 13);  // normal: rebias 15 -> 127
    } else if (man == 0) {
        bits = sign;  // zero
    } else {  // subnormal: normalise the mantissa
        int e = -1;
        do { e++; man <<= 1; } while (!(man & 0x400));
        bits = sign | ((uint32_t)(112 - e) << 23) | ((man & 0x3ff) << 13);
    }
    float f;
    memcpy(&f, &bits, 4);
    return f;
}
```

Listing 4.1: gemma4.c:372–388

How does a plain F32 or BF16 row decode?

The reference row decoder copies F32 bytes directly and uses the BF16 conversion helper for its two-byte elements. The same switch also shows where the row is divided into encoded blocks and routed to the quantised helper. This is the CPU path that all the type-specific examples follow.

```
static void dequant_row(const Tensor *t, int64_t r, float *out) {
    const int64_t K = t->ne[0];
    const uint8_t *p = t->data + r * row_bytes(t);
    switch (t->type) {
        case T_F32:
            memcpy(out, p, K * 4);
            break;
        case T_F16:
            for (int64_t k = 0; k < K; k++) { uint16_t h; memcpy(&h, p + 2 * k, 2); out[k] = f16_table[h]; }
            break;
        case T_BF16:
            for (int64_t k = 0; k < K; k++) { uint16_t h; memcpy(&h, p + 2 * k, 2); out[k] = bf16_to_float(h); }
            break;
        case T_Q8_0:  // blocks of 32: fp16 scale d, then 32 int8 q; w = d * q
            for (int64_t b = 0; b < K / 32; b++, p += 34) {
                uint16_t h;
                memcpy(&h, p, 2);
                const float d = f16_table[h];
                const int8_t *q = (const int8_t *)(p + 2);
                for (int j = 0; j < 32; j++) out[b * 32 + j] = d * q[j];
            }
            break;
        case T_Q4_0:  // blocks of 32: fp16 scale d, then 16 bytes; w[j] = d·(lo nibble - 8), w[j+16] = d·(hi - 8)
            for (int64_t b = 0; b < K / 32; b++, p += 18) {
                uint16_t h;
                memcpy(&h, p, 2);
                const float d = f16_table[h];
                for (int j = 0; j < 16; j++) {
```

```
587                     out[b * 32 + j] = d * ((p[2 + j] & 0xF) - 8);
588                     out[b * 32 + j + 16] = d * ((p[2 + j] >> 4) - 8);
589                 }
590             }
591             break;
592         case T_Q4_1: case T_Q4_K: case T_Q5_K: case T_Q6_K: case
    T_IQ4_NL: case T_IQ4_XS: case T_IQ3_S:
593             for (int64_t b = 0; b < K / 32; b++) dequant_blk32(p, t->
    type, b, out + b * 32);
594             break;
```

Listing 4.2: gemma4.c:559–594

How do Q4_K and Q5_K decode one group?

This branch in the CPU helper chooses the 144- or 176-byte super-block, reconstructs the local scale and minimum, and extracts one group's codes. Q5_K then adds its high bit. The output is 32 decoded F32 values; the exact same block bytes remain in the GGUF.

```
502     } else if (type == T_Q4_K || type == T_Q5_K) {  // d, dmin, scales
    [12], (Q5_K: qh[32]), qs[128]
503         const uint8_t *p = row + b / 8 * (type == T_Q4_K ? 144 : 176), *
    sc = p + 4;
504         const int ib = (int)(b % 8);
505         uint16_t hm;
506         memcpy(&h, p, 2), memcpy(&hm, p + 2, 2);
507         const int s = ib < 4 ? sc[ib] & 63 : (sc[ib + 4] & 15) | (sc[ib
    - 4] >> 6) << 4;
508         const int mn = ib < 4 ? sc[ib + 4] & 63 : (sc[ib + 4] >> 4) | (
    sc[ib] >> 6) << 4;
509         const float d = f16_table[h] * s, m = f16_table[hm] * mn;
510         const uint8_t *qs = p + (type == T_Q4_K ? 16 : 48) + 32 * (ib /
    2), *qh = p + 16;
511         for (int j = 0; j < 32; j++) {
512             int q = (qs[j] >> 4 * (ib % 2)) & 15;
513             if (type == T_Q5_K) q |= ((qh[j] >> ib) & 1) << 4;
514             out[j] = d * q - m;
515         }
```

Listing 4.3: gemma4.c:502–515

How does IQ4_XS use its codebook?

The IQ4_XS case locates the selected 32-value group inside a 256-value super-block, combines the low and high scale bits, subtracts the signed-scale bias, and maps each nibble through the shared nonuniform level table. These lines are the CPU decoder used for the real block captured in W1.

```
532     } else if (type == T_IQ4_XS) {  // 136-byte super-blocks: d,
    scales_h (u16), scales_l[4], qs[128]
533         const uint8_t *p = row + b / 8 * 136;
534         const int ib = (int)(b % 8);
535         uint16_t sh;
536         memcpy(&h, p, 2), memcpy(&sh, p + 2, 2);
537         const int ls = ((p[4 + ib / 2] >> 4 * (ib % 2)) & 15) | (((sh >>
     2 * ib) & 3) << 4);
538         const float dl = f16_table[h] * (ls - 32);
539         const uint8_t *q = p + 8 + ib * 16;
```

```
540         for (int j = 0; j < 16; j++) out[j] = dl * kvalues_iq4nl[q[j] &
    15], out[j + 16] = dl * kvalues_iq4nl[q[j] >> 4];
```

Listing 4.4: `gemma4.c:532–540`

Where does Metal decode while multiplying?

This complete helper reads a row's packed bytes, calls the block decoder for K and IQ formats, and converts only 32 values into a half tile. The full function makes the memory boundary visible: the source pointer remains in the mapped GGUF, the local array is only 32 values, and the destination is a bounded tile that the matmul reuses. The shared device decoder includes the IQ4_XS table branch; here it is reached through `dequant_blk32`.

```
263 static void dequant32(device const uchar *row, int wtype, int k,
    threadgroup half *dst) {
264     half w[32];
265     if (is_blk(wtype)) {
266         float f[32];
267         dequant_blk32(row, wtype, k / 32, f);
268         for (int i = 0; i < 32; i++) w[i] = half(f[i]);
269     } else if (wtype == 2) {  // Q4_0: low nibbles are k .. k+15, high
    nibbles k+16 .. k+31
270         device const uchar *blk = row + (k / 32) * 18;
271         const half d = *(device const half *)blk;
272         device const ushort *qs = (device const ushort *)(blk + 2);
273         for (int i = 0; i < 8; i++) {
274             const ushort v = qs[i];  // bytes 2i, 2i + 1
275             w[2 * i] = d * half(int(v & 15) - 8);
276             w[2 * i + 1] = d * half(int((v >> 8) & 15) - 8);
277             w[16 + 2 * i] = d * half(int((v >> 4) & 15) - 8);
278             w[16 + 2 * i + 1] = d * half(int(v >> 12) - 8);
279         }
280     } else if (wtype == 8) {  // Q8_0
281         device const uchar *blk = row + (k / 32) * 34;
282         const half d = *(device const half *)blk;
283         device const char *q = (device const char *)(blk + 2);
284         for (int i = 0; i < 32; i++) w[i] = d * half(q[i]);
285     } else if (wtype == 1) {  // F16
286         for (int i = 0; i < 32; i++) w[i] = ((device const half *)row)[k
     + i];
287     } else {
288         for (int i = 0; i < 32; i++) w[i] = half(weight(row, wtype, k +
    i));
289     }
290     for (int g = 0; g < 4; g++) {
291         threadgroup half4 *o = (threadgroup half4 *)(dst + g * 64);
292         o[0] = half4(w[8 * g], w[8 * g + 1], w[8 * g + 2], w[8 * g + 3])
    ;
293         o[1] = half4(w[8 * g + 4], w[8 * g + 5], w[8 * g + 6], w[8 * g +
     7]);
294     }
295 }
```

Listing 4.5: `gemma4_metal.m:263–295`

Experiment

What does the storage saving cost on a tiny, controlled input? This W0 program builds one deterministic 256-value vector and stores it as each type present in the E4B file: F32, BF16, Q4_K, Q5_K, and IQ4_XS. It reconstructs the values and reports mean and maximum absolute error. The quantisers use simple fixed scales and real block layouts for teaching; they are not the model publisher's calibration or quantisation process. The run needs no model file.

Compile and run:

```
clang code/examples/ch04_blocks.c -lm -o code/build/ch04_blocks
code/build/ch04_blocks
```

The captured output is stored in `code/expected/ch04-w0.txt`. Inspect the bytes per 256 values and the error under each representation. F32 reconstructs exactly; BF16 rounds the low fraction bits; the three block formats trade additional approximation for fewer bytes. Their errors are specific to this synthetic vector and do not predict perplexity or answer quality.

Experiment W0

```
F32     bytes=1024 mean_abs_error=0 max_abs_error=0
BF16    bytes=512 mean_abs_error=0.00064384408 max_abs_error=0.0018749833
Q4_K    bytes=144 mean_abs_error=0.033320323 max_abs_error=0.125
Q5_K    bytes=176 mean_abs_error=0.016152341 max_abs_error=0.0625
IQ4_XS  bytes=136 mean_abs_error=0.034725338 max_abs_error=0.1171875
IQ4_XS codebook levels: -127 -104 -83 -65 -49 -35 -22 -10 1 13 25 38 53 69 89 113
```

The scales and local fields are deliberately simple and exactly representable in the block headers. These errors compare reconstructions of one synthetic vector, not model outputs or the publisher's quantiser.

F32 needs 1,024 bytes for this vector and reconstructs it exactly; BF16 needs half as many. The three quantised encodings are smaller again: Q4_K uses 144 bytes, Q5_K uses 176, and IQ4_XS uses 136. In this particular run Q5_K's mean and maximum errors are lower than Q4_K's, at the cost of 32 bytes. IQ4_XS is the smallest block here and its error is close to Q4_K's, but a few bytes of difference do not establish a general quality ordering.

These values are absolute errors over the vector, so their scale depends on the vector's values. The program uses deterministic values spanning a modest range and simple fixed quantiser parameters, which makes repeated runs comparable and the byte maps easy to inspect. A production quantiser typically chooses scales to fit each real block and may use a calibration objective; it would be misleading to treat this demonstration as a proxy for that process. It answers a narrower question: how much do the five encodings change these 256 known values under one transparent set of rules?

The point of a W0 exercise is repeatability without a model. A reader can change the vector, observe which formats respond to outliers, or change a local scale and see the error shift. For example, a single large outlier can force a uniform scale to spend a large part of its code range on that value, leaving fewer useful levels for the smaller values. A locally scaled block can adapt groups independently; a nonuniform codebook can devote more levels near zero. Each feature addresses a different source of error, and none guarantees a better end-to-end answer without a measured evaluation.

Can we decode real packed bytes without asking the engine to load the model? This W1 script reuses Chapter 3's GGUF directory reader, selects one tensor for each type, and reads only four bytes for F32, two for BF16, and the first encoded super-block for each quantised type. The total payload read is 462 bytes. It reports the first values from each and hand-checks the IQ4_XS sample against its scale and lookup table. The full tensor directory is used to calculate the type totals, but the weight payload is not scanned.

Run:

```
python3 code/gguf_block.py "$MODEL"
```

The W1 capture was made from the local E4B GGUF on 2026-10-22. Inspect the type totals, file fractions, tensor names, offsets, raw scale/code bytes, and first decoded values. The script maps the file read-only to reuse the directory parser, but slices only those selected payload bytes; it does not scan or hash the rest of the payload, load the model, or run inference. Output is saved in `code/expected/ch04-w1.txt`.

Experiment W1

```
F32 output_norm: 222.464 MB; 4.72% of file; first value 7.90625
BF16 per_layer_model_proj: 55.050 MB; 1.17%; bytes 0b ba -> -0.00053024292
IQ4_XS blk.0.attn_q: 2047.181 MB; 43.41%; first values -0.017282486 0.0292472839
    -0.0332355499 0.0292472839; hand check True
Q4_K token_embd: 392.970 MB; 8.33%; codes 0 9 14 6; values -0.0351476669 0.00668674707
    0.0299280882 -0.00725805759
Q5_K per_layer_token_embd: 1981.923 MB; 42.03%; codes 26 30 27 28; values -0.0138938427
    0.0471189022 0.00135934353 0.0166125298
payload bytes read total: 462
```

The W1 script prints each selected tensor's exact offset and block bytes in the saved output. These are real model bytes; decoding them is not an accuracy benchmark and includes no inference.

The CPU and Metal decoders should agree because they implement the same packed layouts. The W1 samples exercise one real block or scalar per format. The project notes report a separate bit-for-bit check across six real tensors for the engine's decoder and an independent reference, with zero values differing (ENGINE-MATH §9.6); that broader check is documented history, not run by this chapter's script.

How does the weight tile meet the activation tile?

Prefill has several input rows available together. In the snapshot's Metal matmul, the input rows are fp16 and the accumulator fragments are fp32. A threadgroup decodes blocks covering 32 output rows and 64 input positions at a time into fp16 threadgroup memory. Simdgroups share those decoded weights across token rows, multiply fp16 activation tiles by fp16 weight tiles, and accumulate into fp32 fragments. The tiles are temporary, bounded workspace; the original tensor remains packed in its mapped GGUF pages.

For a one-token decode step, a matrix-matrix tile would leave most of its token lanes empty. The matvec kernel instead assigns output rows to simdgroups and walks one row's 32-value blocks, so a block scale is read once for the lane group that uses it. Some paths fuse work around the multiply, such as a norm of the input or the gate/up activation. Those fusions do not change the weight layout: they change which surrounding work shares a dispatch.

The CPU reference has a different purpose. For a tile of output rows it calls `dequant_row` once per row, stores a transposed float tile in per-thread scratch, then reuses that tile for a small batch of input rows. This makes the CPU implementation a readable numerical reference and avoids repeating dequantisation for every token row. It also makes the memory distinction concrete: CPU scratch is a working tile sized for a bounded number of rows, while the multi-gigabyte mapped tensors remain in their stored representation.

The output of a multiply is not always stored in the same precision as its accumulator. The Metal kernel's accumulator is fp32, but it writes fp16 for some fused gate/up results and fp32 for other matrix outputs. Later producers may deliberately cast values to fp16 because the next GPU operation expects that representation. These choices put rounding boundaries at known handoffs rather than after every individual multiply-add.

This explains why "the weights are quantised" is only one part of the arithmetic path. A stored IQ4_XS code is converted through a table and scale into a temporary half value; an activation already arrives as half; the product contributes to a wider accumulator; and the result may be rounded to half at a later boundary. A CPU reference may decode to float and accumulate in float. A small difference between CPU and GPU is therefore possible even when both read exactly the same bytes and use the same codebook.

PREPRINT

Pitfalls from the project notes

Pitfall

A smaller per-layer matrix did not earn its accuracy cost. The 84 per-layer `inp_gate` and `proj` matrices are stored as F32 in the file and converted to F16 copies during loading. The notes say Q8_0 copies were 2% faster, but mean KL divergence against llama.cpp rose from 0.00001 to 0.001 and the top token changed 0.7% of the time. The project kept F16. Source: snapshot `README.md`, "F16 copies of the per-layer matrices" (around line 216). This is a reminder that storage savings and output agreement must be measured together.

Pitfall

More rows per IQ4_XS simdgroup left the GPU with too little parallel work. The eight-row experiment took 324 microseconds for the gate-and-up matvec, compared with 306 microseconds at four rows. The symptom was a slower kernel despite processing more rows together; the cause recorded in the notes was too few threadgroups to occupy the GPU. The four-row setting was kept. Source: snapshot `README.md`, "The model file" (around line 162). The format's compact bytes do not remove the need to size the compute grid for the matrix.

Pitfall

A corrupted KL result looked like a catastrophic accuracy regression. During decode-path perplexity, the dense engine sometimes overwrote prompt tokens while the GPU was still reading them; KL divergence jumped to 0.83 instead of about 0.00001. The cause was a CPU/GPU race on the token buffer, not a faulty block decoder or quantised weight. The fix made those writes wait until the GPU had finished reading. Source: snapshot `README.md`, "Bugs fixed on the way" (around line 283). Accuracy checks must validate their data flow before blaming the stored format.

Pitfall

A wider tile spilled its accumulators and ran much slower. Changing the Metal matmul from 32 to 64 weight rows per simdgroup was meant to halve activation loads, but throughput fell from 260 to 44 tokens per second because the larger live accumulator set spilled out of registers. The narrower tile was retained. Source: snapshot `README.md`, "Things that did not work" (around line 288). A more ambitious tile can lose because register pressure is part of the storage trade-off too.

Exercises

Exercises

1. **Read.** Work out the bytes per weight for F32, BF16, IQ4_XS, Q4_K, and Q5_K from the E4B type table. Explain why the block types are not exactly 0.5, 0.5625, and 0.6875 bytes per value before metadata is counted.
2. **Read.** Given a Q4_0 block with scale 0.25 and first byte `0x9E`, decode the weights at positions 0 and 16.
3. **Change.** Extend `code/gguf_block.py` with a tensor selector and block index. Print the chosen bytes and a hand-checkable decode, while retaining the strict one-block read bound.
4. **Measure.** Compare the size of one F16 matrix with its Q4_K or IQ4_XS tensor from the directory. State the exact format and account for block metadata.
5. **Read.** Trace one IQ4_XS group from its 6-bit local scale and nibble through the lookup table. Use the W1 bytes to calculate the first reconstructed value by hand.
6. **Read.** Trace where the CPU row decoder and the Metal tile decoder handle the same type. Explain why temporary decoded tiles do not imply a full F16 copy of the model, and why the per-layer F32 matrices are an exception.

Rebuild checkpoint

Rebuild checkpoint

Milestone m3: decode quantised rows. Implement `steps/m3_dequant.c` for Q8_0, Q4_0, Q4_1, Q4_K, Q5_K, IQ4_XS, BF16, and F16, then add Q6_K if following the full snapshot decoder. Compare the decoded values with `tools/check_math dequant` on its six listed tensors. The check is bit equality against those reference numbers; merely matching a few hand calculations is not enough.

PRE

Chapter 5

The plain-C reference

The question

What is the slow, obviously right version of every step, and how can it tell us whether the fast GPU path is doing the same work?

The idea

Why keep a CPU path if the GPU is the point?

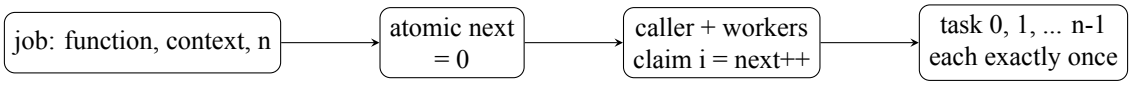
A GPU kernel is an optimized implementation of a small operation. It may read weights in a different order, use half precision in an intermediate, combine steps, or decline a tensor format altogether. A CPU implementation gives the engine another route through the same model: it can run on a machine without Metal, take over when a GPU operation declines an input, and provide a reference output to compare with the GPU. That is a practical definition of a reference, not a claim that floating-point answers must be bit-for-bit identical.

The CPU path is not a separate toy model. It starts from the same mapped tensor data and implements the same layer operations: select an embedding row; normalize; multiply by quantised matrices; prepare queries, keys, and values; rotate query and key coordinates by RoPE; attend over the cache; run the feed-forward block; and repeat for every layer. The same forward routine calls either the GPU implementation or the CPU task depending on whether a GPU operation accepts the arguments.

A decode token is especially demanding: each layer reads a large share of its weights to produce one next activation row. The reference may take seconds for a model token, but it gives us a route whose loops and intermediate arrays can be inspected. When an optimized path disagrees, the two paths let us ask where the first difference appears. Chapter 20 develops the numeric comparison in detail; here we follow the CPU route so that "compare the outputs" has a concrete meaning.

How do worker threads divide the work?

The thread pool is persistent. It starts worker threads once, then each job publishes a function, context, and task count. The calling thread also participates. All participants claim the next task number by incrementing one atomic counter, so tasks with uneven amounts of work do not leave one fixed chunk stranded on a slow worker. A generation counter and condition variable wake the workers for the next job; a second condition variable lets the caller wait until every worker has finished.



The small task interface is worth noticing. A task receives a context pointer, a task index, and a thread ID. In the matrix multiply, the task index identifies one tile of output rows, while the thread ID selects

PREPRINT

private scratch memory. The atomic counter distributes independent tiles; the scratch pointer prevents two workers from overwriting the same temporary tile.

What does the pool promise at the boundary?

`parallel_for(n, fn, ctx)` returns only after all indices in `[0,n)` have run. The implementation resets the atomic counter, advances the generation, wakes the workers, and then lets thread zero claim work too. When the workers finish, the last one signals the waiting caller. There is no per-task allocation or thread creation in this path.

The worker loop is intentionally visible in the excerpt. It locks only while observing the generation and accounting for completion. It releases the mutex while doing tasks, so a long tile does not hold up the thread that needs to claim another index. The pool is a general scheduling tool, but it does not make dependent steps parallel: attention preparation must write cache rows before attention reads them, and the feed-forward block consumes the attention residual. Those boundaries remain sequential.

What happens when the work is not divisible?

The final worker can claim an index equal to `n`; the loop condition rejects it. If there are fewer tasks than workers, some workers do no task but still acknowledge the generation, allowing the caller to finish. This is why the completion count tracks workers, while the task counter tracks actual work. The calling thread does not decrement `busy`; it waits only for the workers.

For a tiled matrix with 161 output rows and tiles of 16, there are 11 tasks: ten full tiles and one tile with a single valid row. Each tile task computes its own valid row count. A fixed split into equal row ranges would need separate tail logic in the scheduler; the current scheme leaves the tail local to the tile and keeps the scheduler simple.

The real code

Where does a task get claimed?

The excerpt shows the worker loop and the public parallel-for boundary. A generation change means “there is a new job”; the atomic counter means “claim an unused index.” The calling thread executes `run_tasks(0)` after waking the workers, so it is not idle while the pool runs. The excerpt shows the worker side of the synchronization boundary; the caller-side reset and wait complete the job before the next operation reads its results.

```
110 static void run_tasks(int tid) {
111     for (int i; (i = atomic_fetch_add(&pool.next, 1)) < pool.n;) pool.fn
        (pool.ctx, i, tid);
112 }
113
114 static void *pool_worker(void *arg) {
115     const int tid = (int)(intptr_t)arg;
116     int seen = 0;
117     pthread_mutex_lock(&pool.mu);
118     for (;;) {
119         while (pool.gen == seen) pthread_cond_wait(&pool.wake, &pool.mu)
        ;
120         seen = pool.gen;
121         pthread_mutex_unlock(&pool.mu);
122         run_tasks(tid);
123         pthread_mutex_lock(&pool.mu);
124         if (--pool.busy == 0) pthread_cond_signal(&pool.done);
125     }
126     return NULL;
```

PREPRINT

```
127 }
```

Listing 5.1: `gemma4.c:110–127`

The separate caller side resets the shared counter, advances the generation, broadcasts to workers, runs tasks on the caller, and waits for worker completion (`gemma4.c:138--153`).

How does the CPU multiply quantised rows?

The CPU matrix multiply accepts activations as floating-point values and weight rows in the model's stored format. Each task takes 16 neighbouring weight rows, dequantises each row once into a k-major tile, and reuses the tile while it processes up to four activation rows. The layout `wt[k * NB + j]` puts the 16 output weights for one input coordinate next to each other. The compiler's four-float vector type then updates four output accumulators together.

The accumulator array has room for four token rows and four vectors of four output columns. The loops deliberately handle a partial last token group and a partial final output tile. The expensive format-specific work is in `dequant_row`, from Chapter 4; this routine can therefore share the same matmul loop among Q4, Q5, IQ4, and other supported row formats.

```
705 static void matmul_tile(void *ctx, int tile, int tid) {
706     const MatmulJob *job = ctx;
707     const int K = job->K, N = job->N, n0 = tile * NB;
708     const int nn = N - n0 < NB ? N - n0 : NB;
709     float *row = job->scratch[tid], *wt = row + K;
710 
711     for (int j = 0; j < NB; j++) {
712         if (j < nn) dequant_row(job->w, n0 + j, row);
713         else memset(row, 0, K * sizeof(float));
714         for (int k = 0; k < K; k++) wt[k * NB + j] = row[k];
715     }
716 
717     for (int t0 = 0; t0 < job->T; t0 += TB) {
718         const int nt = job->T - t0 < TB ? job->T - t0 : TB;
719         const float *x[TB];
720         for (int i = 0; i < TB; i++) x[i] = job->x + (size_t)(t0 + (i <
    nt ? i : 0)) * K;
721 
722         vec acc[TB][NB / VW] = {{0}};
723         for (int k = 0; k < K; k++) {
724             const vec *w = (const vec *)(wt + k * NB);
725             for (int i = 0; i < TB; i++) {
726                 const float xv = x[i][k];
727                 for (int j = 0; j < NB / VW; j++) acc[i][j] += xv * w[j
    ];
728             }
729         }
730         for (int i = 0; i < nt; i++)
731             for (int j = 0; j < nn; j++) {
732                 float *y = job->y + (size_t)(t0 + i) * N + n0 + j;
733                 *y = acc[i][j / VW][j % VW] + (job->accumulate ? *y : 0)
    ;
734             }
```

Listing 5.2: `gemma4.c:705–734`

Draft book, October 2026. Not peer reviewed.

Why does a tile help if the CPU is the reference?

The straightforward mathematical operation is one dot product per output row. Doing that literally would repeatedly decode a row for each input token during prefill. The tile instead decodes a small group of rows once, then reuses them across four input rows. This is still a transparent CPU calculation, but it avoids an obviously wasteful loop order.

The scratch allocation is per worker and grows only when a larger input width is encountered. With `NB=16`, a worker holds one temporary row plus 16 rows arranged by input coordinate. With `TB=4`, it keeps the partial sums for four token rows in registers. These constants are engineering choices to try and measure, not properties of the model. The tiled path also makes a good baseline for later GPU work: same weights and accumulation precision, different parallel hardware and implementation details.

What does one tiny matvec do by hand?

Take two output rows and a three-value input, with ordinary (already decoded) weights. Let the input be [1, 2, -1]. Row zero has weights [2, 0, 1], so its dot product is 2 + 0 - 1 = 1. Row one has weights [-1, 1, 3], so its result is -1 + 2 - 3 = -2. The output is [1, -2]. A quantised row changes where the weights come from, not the outer calculation: the row decoder expands the stored codes and scales into three float values, then the same multiply-and-accumulate loop computes the row result.

Now imagine two input rows, [1,2,-1] and [0,1,1]. The same weights produce [1,-2] for the first row and [1,4] for the second. The scalar approach asks for each output row, decodes its weights, and loops over every input row. The tiled approach still performs those products, but it arranges the decoded weights so one input coordinate can update several output columns together. With a larger matrix, this changes how often packed data is decoded and how much work the compiler can vectorize; it does not change which mathematical dot products the matrix represents.

A useful first check when a matvec differs is to separate the stages. Print the decoded weight row and compare it with the reference decoder. If those values agree, compare the input after its half conversion. If that agrees too, compare one row's accumulated output. This order avoids blaming the vector loop for a mismatch already present in the decoder or activation conversion. For a real model matrix the rows have thousands of values, so a few selected indices and summary differences are easier to inspect than dumping the entire tensor.

Where can rounding enter the reference?

The CPU route has explicit precision boundaries. Packed weights are decoded to float for the CPU matmul; normalized layer inputs are converted to half before they become matmul inputs; attention queries and the cache may also be stored as half or as quantised 8-bit values with half scales. Accumulation in the CPU tile is float. These are architectural choices shared with the engine's intended GPU contract, though the exact order of additions can differ from a GPU kernel.

Suppose a calculation adds four small products to a large partial sum. Reordering those additions can round away a different small contribution. The CPU vector loop groups four lanes and combines its partial vectors in a particular order; a GPU reduction can group terms differently. Therefore a tiny discrepancy after one dot product may be ordinary rounding, while a large jump or a first mismatch in a decoded value suggests a different bug. Compare at boundaries and keep the tolerance tied to the operation under test rather than one arbitrary threshold for the whole model.

The same distinction applies to attention. The CPU path computes all visible scores, finds their maximum, exponentiates the shifted scores, sums them, then applies the normalized weights to values. A GPU kernel may use an online reduction that does not materialize every score at once. Both can implement the same softmax result within floating-point tolerance while taking different routes through the arithmetic. The later attention chapters show the memory and speed reason for that GPU design; for this chapter, the CPU's explicit arrays make a useful inspection point.

PREPRINT

What does normalization preserve?

RMS normalization scales a vector using its root mean square, then multiplies each coordinate by a learned weight. For a four-value hand example, take $x = [1, -1, 2, 0]$, $w = [1, 2, 1, 1]$, and ignore the tiny epsilon. The mean square is $(1+1+4+0)/4 = 1.5$, so the common scale is $1/\sqrt{1.5} \approx 0.8165$. The weighted result is approximately $[0.8165, -1.6330, 1.6330, 0]$. The operation changes the overall scale to a predictable range while retaining coordinate-specific learned gains.

The CPU computes the sum in double precision before converting the mean square back to float. For a layer boundary, `rmsnorm_rows` writes the normalized values in half precision because the matmul input follows the GPU's activation convention. The separate scalar `rmsnorm` helper stays in float for operations such as attention preparation and the router. This difference is a reminder that "CPU path" does not mean "every intermediate is full precision"; it means the CPU executes the designated fallback operations.

```
789 static float rms_scale(const float *x, int n, float eps) {
790     double ss = 0;
791     for (int i = 0; i < n; i++) ss += (double)x[i] * x[i];
792     return 1.0f / sqrtf((float)(ss / n) + eps);
793 }
794
795 static void rmsnorm(float *out, const float *x, const float *w, int n,
        float eps) {  // w may be NULL
796     const float scale = rms_scale(x, n, eps);
797     for (int i = 0; i < n; i++) out[i] = x[i] * scale * (w ? w[i] : 1.0f
        );
```

Listing 5.3: `gemma4.c:789–797`

How do RoPE and attention use the cached rows?

The query and key vectors are normalized per head. RoPE then rotates paired coordinates by an angle that depends on the token position. Values are normalized without a learned weight and written to the cache; keys are normalized with their learned weight, rotated, and cached. A shared-KV layer skips writing its own key and value rows, while its query still attends to the earlier layer's cache.

For a tiny rotation, let the first pair be $(x_0, x_1) = (1, 0)$ and the position-dependent angle be 90 degrees. The result is $(0, 1)$. At position zero the angle is zero and the pair stays $(1, 0)$. The actual angle shrinks for later coordinate pairs according to the RoPE base; the code steps through those pairs and applies the same two-coordinate rotation. The simple example shows the purpose: position changes the query/key coordinates before their dot products, so attention scores can depend on order.

After preparation, the CPU attention task selects the head and its grouped key/value head. For each query it computes dot products against the visible keys, subtracts the largest score before exponentiating, normalizes those weights, and forms a weighted sum of values. Causal mode ends at the query's own position; a sliding-window layer starts no earlier than its window boundary. The code in this section is intentionally the readable baseline; later chapters examine the GPU kernels and online softmax.

```
1245 // NeoX-style RoPE on the first n_rot dims of a head: pair (i, i + n_rot
        /2) turns by pos·base^(-2i/n_rot)/ff[i].
1246 // The angle is built as ggml's CPU rope does (theta *= base^(-2/n_rot)
        per pair).
1247 static void rope(float *x, int pos, int n_rot, float base, const float *
        ff) {
1248     const int half = n_rot / 2;
1249     const float theta_scale = powf(base, -2.0f / n_rot);
1250     float theta = (float)pos;
1251     for (int i = 0; i < half; i++, theta *= theta_scale) {
1252         const float th = theta / (ff ? ff[i] : 1.0f);
```

```
1253          const float c = cosf(th), s = sinf(th), x0 = x[i], x1 = x[i +
      half];
1254          x[i] = x0 * c - x1 * s;
1255          x[i + half] = x0 * s + x1 * c;
1256      }
1257  }
```

Listing 5.4: gemma4.c:1245–1257

Where does the attention task become a layer result?

The complete attention wrapper first projects queries, keys, and values, then runs the preparation and attention tasks, and finally projects the attended heads back into the embedding width. Each call is a dispatch point: when a Metal function accepts the data, the CPU task is skipped; otherwise the wrapper synchronizes before the CPU reads inputs and writes results. The order is essential: the cache must contain this call's keys and values before the query can read them.

The pooling unit for attention is a head and a group of up to 32 query rows. That gives independent tasks without splitting a query's score and value reductions across workers. The CPU implementation allocates temporary arrays for one task's scores, query, key/value row, and output. This costs memory and setup work, but keeps ownership obvious: one task writes one head slice for each assigned query.

```
1313  enum { ATT_QB = 32 };  // queries per task
1314
1315  // first key a query at `pos` sees: all before it, or the last n_swa in
      sliding-window layers
1316  static int first_key(const Model *m, const Layer *L, int pos) { return L
      ->swa && pos >= m->n_swa ? pos - m->n_swa + 1 : 0; }
1317
1318  // softmax attention over the cache (scale 1: the queries and keys are
      RMS-normed), causal unless bidir.
1319  // task i: head i % n_head, queries [ATT_QB * (i / n_head), +ATT_QB)
1320  static void attn_task(void *ctx, int task, int tid) {
1321      (void)tid;
1322      const AttnJob *job = ctx;
1323      const Layer *L = job->L;
1324      const int hd = L->hd, nq = job->m->n_head * hd, nkv = L->n_head_kv *
       hd;
1325      const int h = task % job->m->n_head, kvh = h / (job->m->n_head / L->
      n_head_kv);
1326      const int t0 = task / job->m->n_head * ATT_QB, t1 = t0 + ATT_QB <
      job->T ? t0 + ATT_QB : job->T;
1327      float *p = xmalloc((size_t)(job->n_past + job->T) * sizeof(float)),
      *o = xmalloc(hd * sizeof(float));
1328      float *q = xmalloc(hd * sizeof(float)), *kv = xmalloc(hd * sizeof(
      float));
1329
1330      for (int t = t0; t < t1; t++) {
1331          const int pos = job->n_past + t, s0 = first_key(job->m, L, pos),
       s1 = job->bidir ? job->n_past + job->T - 1 : pos;
1332          for (int i = 0; i < hd; i++) q[i] = f16_table[job->q[(size_t)t *
       nq + h * hd + i]];
1333          float mx = -INFINITY, sum = 0;
1334          for (int s = s0; s <= s1; s++) {
1335              kv_get(job->k, job->ks, s % job->ring, nkv, kvh * hd, kv, hd
      );
1336              p[s] = dot(q, kv, hd);
1337              if (p[s] > mx) mx = p[s];
```

```
1338            }
1339            for (int s = s0; s <= s1; s++) { p[s] = expf(p[s] - mx); sum +=
        p[s]; }
1340            for (int i = 0; i < hd; i++) o[i] = 0;
1341            for (int s = s0; s <= s1; s++) {
1342                kv_get(job->v, job->vs, s % job->ring, nkv, kvh * hd, kv, hd
        );
1343                const float w = p[s] / sum;
1344                for (int i = 0; i < hd; i++) o[i] += w * kv[i];
1345            }
1346            for (int i = 0; i < hd; i++) job->o[(size_t)t * nq + h * hd + i]
         = float_to_half(o[i]);
1347        }
1348        free(p);
1349        free(o);
1350        free(q);
1351        free(kv);
1352    }
```

Listing 5.5: `gemma4.c:1313–1352`

What follows attention in a layer?

The attention output is projected and added to the residual through a normalized, learned post-attention scale. The feed-forward input is normalized, projected through gate and up matrices, and combined with GeGLU: the gate's GELU output is multiplied coordinatewise by the up projection. A down projection returns to embedding width. In dense layers that is the shared MLP output; in expert layers, routed expert outputs are also combined. A final normalized residual add produces the next layer's input.

A useful four-coordinate picture is enough to understand the gate: if GELU(g) is $[0.2, 0.8, 0.5, 0]$ and up is $[2, 1, -2, 3]$, the product is $[0.4, 0.8, -1, 0]$. The down matrix then mixes that hidden vector back into the residual width. The real hidden vector is much larger and quantised weights are decoded by the shared matmul routine, but the order of operations is the same.

In this model, some layers also have per-layer embeddings after the feed-forward stage. Before the layer loop, the engine projects the token embedding into the per-layer inputs. For each layer, `pl_block` projects that layer's input through `pl_gate`, applies GELU, and multiplies elementwise by that layer's projected per-layer input. It then applies `pl_proj` and adds the result through `pl_post_norm`, including the layer's output scale, to produce the next normalized input (`gemma4.c:2012--2031`).

Where does the first activation row come from?

For a text token, embedding lookup selects the corresponding row from the token embedding matrix and applies the model's embedding scale. The CPU callback decodes that row from its stored quantised format into the activation buffer. If the caller provides media embeddings instead, the same dispatch boundary copies the supplied row rather than looking up a vocabulary row. This is the first place the token ID becomes a vector of numbers.

```
2040 typedef struct { const Tensor *w; const int *rows; float *out; float
        scale; } EmbedJob;
2041
2042 static void embed_row(void *ctx, int t, int tid) {
2043     (void)tid;
2044     const EmbedJob *job = ctx;
2045     const int K = (int)job->w->ne[0];
2046     float *out = job->out + (size_t)t * K;
2047     dequant_row(job->w, job->rows[t], out);
2048     for (int i = 0; i < K; i++) out[i] *= job->scale;
```

PREPRINT

```
2049 }
```

Listing 5.6: gemma4.c:2040–2049

How does one call cover the whole model?

The forward pass starts by getting token embedding rows (or accepting already-projected media rows), and makes the first normalized input. It then visits layers in order. Each layer consumes the previous activation, runs attention, adds and normalizes the attention residual, runs feed-forward, and optionally applies the per-layer embedding block. The last layer leaves a normalized representation for the output matrix, whose rows are the vocabulary scores, or logits.

The CPU build uses this same forward routine with GPU calls disabled at compile time. A mixed CPU/GPU build can take different branches operation by operation. The `STEP` macro makes the fallback rule concrete: ask the GPU function to accept the work; if it returns false, synchronize and run the CPU task across the thread pool. The synchronization prevents the CPU from reading an input or output that the GPU still owns. Matmul has a related wrapper because it accepts a tensor and can choose a specialized fused norm-matvec before the ordinary GPU matmul or CPU path.

```
664 // One step: encoded for the GPU if `gpu_call` accepts it, else run on
        the CPU threads.
665 #define STEP(gpu_call, n, cpu_task, job)
            \
666     do {
            \
667         if (!GPU(gpu_call)) { gpu_sync(); parallel_for((n), (cpu_task),
        (job)); } \
668     } while (0)
```

Listing 5.7: gemma4.c:664–668

What are the loop's deliberate shortcuts?

The layer loop can discard rows whose logits will not be returned after the model's layers that write a KV cache. During a prefill, only the final row usually needs a next-token score. This is a model-level shortcut: it preserves the rows needed by the cache and output while avoiding work for rows the caller will never score. The loop also submits completed GPU work between layers so a mixed path can overlap CPU work with queued GPU work where dependencies permit.

The CPU-only route does not use those GPU overlap opportunities, but it still follows the same state bookkeeping. The state records how many positions have been processed; the cache stores keys and values by position, using a ring for sliding-window attention when appropriate. Once all layers finish, the output projection produces logits for the requested rows. A next-token selector can take the largest score, but comparing only that winner hides small numeric differences. A useful reference comparison records the full vector or a summary of its differences, and identifies the first layer where the paths diverge.

```
2051 // Runs a->tokens[0..T) (or the rows a->embd) at positions st->n_past..
        (continuing the sequence in st); writes
2052 // the logits of this call's tokens [first, first + n_out), before the
        final soft-capping (score() applies it;
2053 // argmax doesn't need it).
2054 static void forward(const Model *m, Acts *a, State *st, int T, int first
        , int n_out, float *logits) {
2055     if (T > a->T_max || st->n_past + T > st->n_ctx) die("forward: %d
        tokens don't fit", T);
2056     if (st->ring < st->n_ctx && (T > st->ring - m->n_swa - 8 ||
        state_keep(m, st, st->n_past) != st->n_past))
```

```
2057         die("forward: the sliding-window cache can't hold this call");
2058     if (st->n_past == 0) st->n_high = 0;
2059     const int E = m->n_embd;
2060     void *prev = NULL;  // per-layer embeddings: all GPU work before
     this call (see pl_inputs)
2061     if (m->n_pl) {
2062         gpu_submit();
2063         prev = gpu_fence();
2064     }
2065     if (xcache && T >= PREFETCH_T) {  // a prefill: start reading the
     first MoE layer's missing experts
2066         gpu_sync();                   // (the staging slots may still be
      in use by the previous call)
2067         for (int l = 0; l < m->n_layer; l++)
2068             if (m->layers[l].router) { cache_prefetch(xcache, l); break;
     }
2069     }
2070     if (a->embd) {  // input rows given (an image): row t of a->embd,
     unscaled; a->tokens are the row indices
2071         Tensor rows = {.type = T_F32, .ne = {E, T, 1, 1}, .data = (const
      uint8_t *)a->embd};
2072         gpu_sync();  // (the GPU may not have read the previous call's
     tokens yet)
2073         for (int t = 0; t < T; t++) a->tokens[t] = t;
2074         EmbedJob ej = {&rows, a->tokens, a->x, 1.0f};
2075         STEP(metal_get_rows(a->x, a->embd, E * sizeof(float), T_F32, a->
     tokens, T, E, 1.0f), T, embed_row, &ej);
2076     } else {
2077         const Tensor *emb = m->tok_embd;
2078         EmbedJob ej = {emb, a->tokens, a->x, m->embd_scale};
2079         STEP(metal_get_rows(a->x, emb->data, row_bytes(emb), emb->type,
     a->tokens, T, E, m->embd_scale), T, embed_row, &ej);
2080     }
2081     if (m->n_pl) {  // the projection part of the per-layer inputs, from
      the embeddings
2082         CastJob cj = {a->xh, a->x, E};
2083         STEP(metal_cast_half(a->xh, a->x, T * E), T, cast_row, &cj);
2084         matmul(a->pl, a->xh, m->pl_model_proj, T, 0);
2085     }
2086
2087     // a->xn: the normed input of the next step. Each layer's last step
     makes the next layer's (and the last
2088     // layer's the output norm's, of all T rows), and its attention's
     add_norm makes the FFN's.
2089     rmsnorm_rows(a->xn, a->x, m->layers[0].attn_norm, T, E, m->eps, 0);
```

Listing 5.8: gemma4.c:2051–2089

The excerpt begins with bounds and cache checks, then chooses between caller-provided rows and token lookup. It projects optional per-layer inputs and creates the first normalized input. The next excerpt is the model's central repeated sequence: attention, residual normalization, feed-forward, optional per-layer embeddings, and a submit point. The final matrix multiply after the loop turns the last activation into vocabulary logits.

```
2091     for (int l = 0; l < m->n_layer; l++) {
2092         const Layer *L = &m->layers[l];
2093         const float *next_norm = l + 1 < m->n_layer ? m->layers[l + 1].
     attn_norm : m->output_norm;
2094         if (l == m->n_kv && Tl > n_out && first >= n_out && !a->bidir) {
```

```
2095             // The layers from here on write no KV cache, so the rows
     whose logits aren't wanted affect nothing
2096             // any more: only rows [first, first + n_out) go on (in a
     prefill, just the last). (Rows are moved
2097             // down, which needs first >= n_out so that no row is
     overwritten before it is read.)
2098             for (int t = 0; t < n_out; t++) a->rows[t] = first + t;
2099             const int P = m->n_layer * m->n_pl;
2100             Tensor xr = {.type = T_F32, .ne = {E, T, 1, 1}, .data = (
     const uint8_t *)a->x};
2101             Tensor pr = {.type = T_F32, .ne = {P, T, 1, 1}, .data = (
     const uint8_t *)a->pl};
2102             EmbedJob xj = {&xr, a->rows, a->x, 1.0f}, pj = {&pr, a->rows
     , a->pl, 1.0f};
2103             GatherJob gj = {a->xn, a->xn, a->rows, E};
2104             STEP(metal_get_rows(a->x, a->x, E * sizeof(float), T_F32, a
     ->rows, n_out, E, 1.0f), n_out, embed_row, &xj);
2105             STEP(metal_gather(a->xn, a->xn, a->rows, n_out, E), n_out,
     gather_row, &gj);
2106             if (P) STEP(metal_get_rows(a->pl, a->pl, P * sizeof(float),
     T_F32, a->rows, n_out, P, 1.0f), n_out, embed_row, &pj);
2107             Tl = n_out, pos += first, first = 0;
2108         }
2109         attention(m, L, l, a, st, Tl, pos);
2110         add_norm(a->x, a->a, L->post_attn_norm, Tl, E, m->eps, 1.0f, a->
     xn, L->ffn_norm);
2111         ffn(m, L, a, Tl, next_norm);
2112         if (m->n_pl) {
2113             if (l == 0) pl_inputs(m, a, Tl, prev);  // (the GPU runs
     layer 0 meanwhile)
2114             pl_block(m, L, l, a, Tl, next_norm);
2115         }
2116         gpu_submit();  // start the GPU on this layer while the next is
     encoded
2117     }
2118
2119     matmul(logits, a->xn + (size_t)first * E, m->output, n_out, 0);
2120     st->n_past += T;
2121     if (st->n_past > st->n_high) st->n_high = st->n_past;
2122 }
```

Listing 5.9: gemma4.c:2091–2122

When should two outputs count as agreement?

Exact equality is a good check for code that should copy bytes or use the same arithmetic order. It is too strict for two floating-point implementations that accumulate in a different order or round activations differently. Compare intermediate vectors with a maximum absolute difference, a relative difference where the reference value is not near zero, and the first index that exceeds a stated tolerance. At the model output, also compare the top token and the distribution of log probabilities; those give useful context but do not replace tensor-level checks.

A matching greedy token is only a coarse check. Two logits vectors can select the same winner while differing everywhere else; conversely, a tiny numeric difference near a tie can change the winner while the vectors remain close. The robust comparison method is to feed both paths the same model, token sequence, cache state, and precision settings; compare after each operation or layer; and report both numerical error and discrete decisions. The dedicated correctness chapter uses KLD and known

reference outputs for this broader job.

Experiment

Can we build and inspect the CPU-only executable without touching any weights? This is W0. From the book root, run the snapshot build script at low priority:

Experiment W0

```
nice -n 10 sh code/snapshot/build.sh
code/build/gemma4_cpu --help
```

The build compiles the snapshot source into `code/build/`; the help invocation parses no model and reads no weight file. Inspect the usage text and confirm that the executable is the CPU target produced without Metal flags. This checks the no-weight route and its command-line entry point; it says nothing about inference speed or numeric agreement.
Try it on your machine (W2): the CPU inference command opens the model and runs inference:

```
code/build/gemma4_cpu -m "$MODEL" -p "Once upon a time" -n 8
```

Compare generated tokens and per-token timing on CPU and GPU with the same prompt and flags. The CPU path should be slower; the README perplexity is a historical accuracy reference, not a speed result.

PRINT

Pitfalls from the project notes

Pitfall

The same top token does not prove the same numbers. The documented CPU comparison reports perplexity 9.292 for the engine and 9.289 for llama.cpp on one chunk, with mean KLD 0.000009 and the same top token on every scored position. The numbers are close, but the table itself records distinct perplexities. A top-token-only test would hide that difference. The practical fix is to compare vectors or log probabilities with a stated metric as well as checking the winning token. Source: snapshot `README.md`, “Accuracy” table, CPU build row; the experiment conditions are recorded there as `c=512, 1 chunk`.

Exercises

Exercises

1. **Read.** Trace one tile task from the atomic task number to the output address. For `NB=16`, `TB=4`, $K = 8$, and $N = 19$, identify which tile owns output row 18 and how many valid rows it writes.
2. **Change.** Replace the shared atomic counter with static contiguous chunks. Compare both schedulers on uniform tile work and on a callback whose task duration increases with its index.
3. **Measure.** In `matmul_tile`, change `NB` or `TB` while preserving the scratch layout. Measure a fixed synthetic matrix and record the compiler flags, repetitions, and median time.
4. **Read.** For the four-value RMSNorm example in this chapter, explain why multiplying by the learned weights after computing the common scale can change the direction of the result.
5. **Change.** Write a small Q8_0 row matvec that decodes one weight at a time and computes one output row. Compare it with a version that decodes a whole row before taking the dot product.
6. **Read.** Design a comparison report for CPU and GPU after one layer: which inputs must match, which vectors would you save, and why can matching top IDs alone miss an error?

Rebuild checkpoint

Rebuild checkpoint

Milestone m4: one CPU layer's essential arithmetic. Build `steps/m4_cpu_layer.c` from the m3 row decoders. Implement RMSNorm and a Q-projection matvec for layer 0, then feed a fixed activation vector through both the small program and the engine's CPU path. Compare the complete output vector and report the maximum absolute difference and first differing index. The snapshot does not currently expose a documented vector-dump environment variable, so the small program's comparison hook must be designed in the book copy before this check can be run. Do not claim a captured m4 result until that hook and its reference vector exist.

Part I recap

Part I followed the work from bytes to activations: Chapter 1 mapped the engine, Chapter 2 turned text into IDs, Chapter 3 found tensors in GGUF, and Chapter 4 decoded the compact numbers stored in their blocks. This chapter assembled those pieces into a readable CPU route. Part II now asks how to keep the same steps while moving their heavy arithmetic to Metal.

Part II

METAL

PREPRINT

Chapter 6

Metal for C programmers

Part II: Metal

Part I built a map of the model, its numbers, and a CPU path that can explain each operation. Part II asks what changes when the same steps run on the GPU. We will start with the small host interface that makes that possible, then follow the data and commands into the decode and prefill kernels. The later chapters will read those kernels; this one gives us the vocabulary to understand how the engine hands them work.

The question

How do we make the GPU run our code with no Xcode, and keep the CPU out of its way?

Reading map: 16 pages

The chapter is planned to fill the complete 16-page estimate in `OUTLINE.md`. These page counts include listings, figures, and exercise space.

Subsection	Pages
Part II opener	0.5
Device, queue, buffers, and compute pipeline	2
Shader source compiled at run time	2
Unified memory and no-copy model buffers	2
Threadgroups and SIMD groups	2
Ordering, command buffers, groups, and fences	2
The engine's C boundary and fallback contract	2
W0 vector-add experiment	1
Pitfalls from the project history	1.5
Exercises and checkpoint m7a	1
Total	**16**

The idea

Four ordinary values are enough to picture a compute job: `a = [1,2,3,4]`, `b = [10,20,30,40]`, and the answer `[11,22,33,44]`. A CPU might run one loop that visits those four positions. A GPU asks us to describe a kernel for one position and launch many copies of it. The host still prepares memory and decides what work can run; the GPU executes the launched grid.

In Metal, a *device* represents the GPU, a *command queue* accepts ordered command buffers, and a *compute pipeline* combines a compiled kernel function with the device's execution state. A command

PREPRINT

buffer is a batch of recorded work. A compute command encoder records which pipeline, buffers, parameters, and grid to use. Recording is not execution: the host can build a sequence first, submit it, and wait only when a CPU decision or read actually depends on its result.

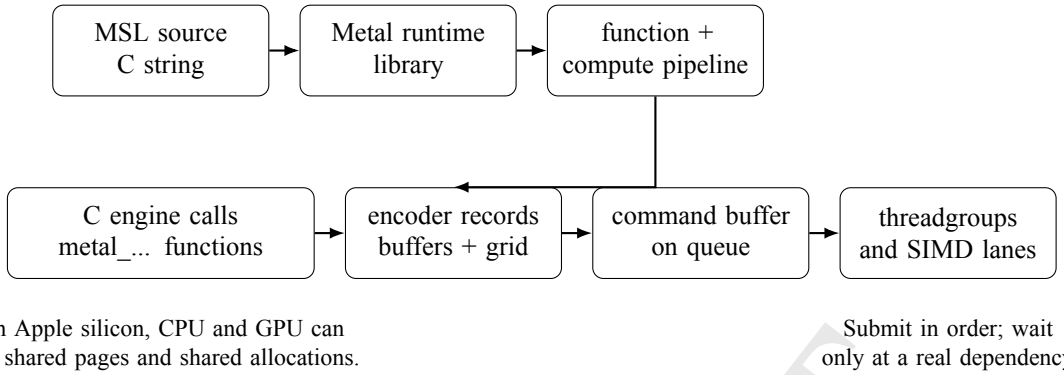


Figure 6.1: The host compiles a kernel, records a dispatch, and submits it to the device. The buffer data stays in GPU-visible memory.

The shape of a dispatch is also a shape of cooperation. Picture 128 threads as four groups of 32 lanes. A *SIMD group* is those 32 lanes executing together; lanes can exchange partial results with SIMD operations. Several SIMD groups can share a threadgroup's scratch memory and meet at a threadgroup barrier. A grid contains one or more threadgroups. The kernel chooses which output each lane owns, and a reduction must be explicit when several lanes contribute to one answer.

Grid	Threadgroup 0	Threadgroup 1	Threadgroup 2	Threadgroup 3
128 threads	SIMD 0: 0–31	SIMD 1: 32–63	SIMD 2: 64–95	SIMD 3: 96–127

The engine's code uses exactly this division in its small reductions. In `block_sum`, each lane first adds within its 32-lane SIMD group using `simd_sum`. Lane zero of each group stores one subtotal into threadgroup memory. A threadgroup barrier makes those subtotals visible, then the threads add the few group totals. The same pattern appears in the matvec kernel: each lane decodes part of a quantized row and contributes to an accumulated dot product.

The real code

What promises does the C header make?

The header is the seam between the C engine and the Objective-C host. Read it before implementation details: it says the GPU may decline a step, buffers must be exposed to Metal, and queue submission is separate from waiting. The caller can use this backend without putting Objective-C types in the model code. These opening comments are the contract, not decoration.

```
1 // Optional Metal backend for gemma4.c (build with -DGEMMA4_METAL, link
      -framework Metal -framework Foundation).
2 //
3 // The metal_* steps don't run immediately: they are encoded, in order,
      into a command buffer, which
4 // metal_commit() submits and metal_sync() waits for. Each step waits
      for the ones before it, except
5 // inside a metal_group(1) ... metal_group(0) bracket, whose steps must
      be independent of each other.
6 // Pointers must lie in GPU-visible memory (metal_map / metal_alloc).
      Each step returns 0 without encoding
7 // anything if it can't handle its arguments, so the caller can sync and
       run its CPU code instead.
```

PREPRINT

Listing 6.1: gemma4_metal.h:1–7

Where does a no-copy model buffer begin?

The engine's tensor pointer can start partway through a page. Metal's no-copy initializer needs a page-aligned address and a page-sized length. So `metal_map` rounds the beginning down and the end up, then creates a shared-storage buffer over that same address range. The tensor's exact address remains available as an offset when `encode` binds it. The rounded page range is the wrapper; it is not a copy of the tensor.

```
1725 // [p, p + size) rounded out to pages, as a buffer over the same memory
1726 int metal_map(const void *p, size_t size) {
1727     const size_t page = (size_t)getpagesize();
1728     const char *start = (const char *)((uintptr_t)p / page * page);
1729     const size_t len = ((uintptr_t)p + size + page - 1) / page * page -
     (uintptr_t)start;
1730     id<MTLBuffer> b = [device newBufferWithBytesNoCopy:(void *)start
     length:len
1731                                                 options:
     MTLResourceStorageModeShared deallocator:nil];
1732     if (!b) return 0;
1733     add_region(start, len, b);
1734     return 1;
```

Listing 6.2: gemma4_metal.m:1725–1734

This is why two adjacent tensor ranges can share a page and why the region lookup is by address range rather than by tensor name. It also explains the lifetime rule: unmapping releases the wrapper, not a separate set of weight bytes. The backing mapping must remain valid, and the GPU must be finished before the wrapper is removed. The storage option is `MTLResourceStorageModeShared`; Apple silicon's unified memory means the CPU and GPU address the same physical memory, though they still need ordering when one side writes data the other side reads.

What happens to the shader string?

The source file begins with the Metal Shading Language program embedded as a C string. `MSL(...)` stringifies its contents, and the prefix adds the standard Metal header. `metal_init` asks the system's Metal runtime to compile that source into a library. It then looks up ordinary kernel functions and creates their compute pipeline states. There is no offline `metal` compiler step in the book's build command: the target Mac compiles the source at startup.

```
1767 int metal_init(void) {
1768     @autoreleasepool {
1769         device = MTLCreateSystemDefaultDevice();
1770         if (!device) return 0;
1771         NSError *err = nil;
1772         library = [device newLibraryWithSource:@(kernel_src) options:nil
      error:&err];
1773         if (!library) { fprintf(stderr, "metal: %s\n", err.
     localizedDescription.UTF8String); return 0; }
1774         for (int i = 0; i < N_KERNELS; i++) {
1775             if (i == K_MATMUL || i == K_MATVEC || i == K_MATVEC_Q4 || i
     == K_MATVEC_Q4K || i == K_MATVEC_IQ3S ||
1776                 i == K_MATVEC_IQ4NL || i == K_ATTN_PREP || i ==
     K_ATTENTION || i == K_ATTN_DECODE || i == K_ATTENTION_FA ||
1777                 i == K_ATTENTION_FA8)
1778                 continue;
```

PREPRINT

```
1779            id<MTLFunction> fn = [library newFunctionWithName:@(
    kernel_names[i])];
1780            kernels[i] = fn ? [device
    newComputePipelineStateWithFunction:fn error:&err] : nil;
1781            if (!kernels[i]) { fprintf(stderr, "metal: kernel %s failed\
    n", kernel_names[i]); return 0; }
1782        }
1783        queue = [device newCommandQueue];
1784        if (@available(macOS 15.0, *)) {
1785            MTLResidencySetDescriptor *desc = [[
    MTLResidencySetDescriptor alloc] init];
1786            resident = [device newResidencySetWithDescriptor:desc error
    :&err];
1787            if (resident) [queue addResidencySet:resident];
1788        }
1789        fprintf(stderr, "metal: %s\n", device.name.UTF8String);
1790        const char *e = getenv("GEMMA4_PROFILE");
1791        if ((profiling = e && *e && *e != '0')) atexit(prof_report);
1792        if ((e = getenv("GEMMA4_SKIP")))
1793            for (int i = 0; i < N_KERNELS; i++) {
1794                const char *f = strstr(e, kernel_names[i]);
1795                const size_t n = strlen(kernel_names[i]);
1796                skip[i] = f && (f == e || f[-1] == ',') && (f[n] == 0 ||
     f[n] == ',');
1797            }
1798        return 1;
1799    }
```

Listing 6.3: `gemma4_metal.m`:1767–1799

Runtime compilation changes the failure point, not the need to handle failure. The compiler has the actual MSL text and the current device at startup, so a misspelled kernel name, unsupported language feature, or invalid declaration is discovered when the library is made. Metal returns an error object with a description; the host reports it and returns failure from initialization. The C program sets its GPU-enabled flag from that result, so it can continue with its CPU implementation when Metal is unavailable or the shader library does not compile. This path is useful during development because the error refers to the source the book describes, rather than to a stale prebuilt metallib.

The trade is startup latency. The embedded source contains helper functions, lookup tables, and kernels for several model operations. Compilation happens once for the library, and pipeline creation happens for each ordinary kernel the initialization loop requests. Specialized variants are deferred until their first use. A fast inference benchmark started after initialization excludes some of that cost; a command that measures process startup includes it. When comparing launch times, state whether initialization and specialization were inside the timer.

Some kernels need compile-time choices. The `variant` helper caches specialized pipelines keyed by weight types, fused operations, head dimension, and other flags. It supplies function constants when it requests a specialized function from the library, then stores the resulting pipeline so the same combination is not rebuilt for every matrix. The source remains one kernel family, while specialization can remove branches and expose fixed bounds to the compiler. The cache has a fixed capacity of 64 variants; the function exits with an error if that bound is reached. Runtime compilation moves work to startup or first use, so it is not free. The trade is a simpler build and pipelines specialized for the actual device and model.

The key has nine slots, so “the same kernel” is not enough to identify a pipeline. A Q4 matvec with fused input normalization is a different specialized function from one that expects a half-precision input. A head-dimension choice can change how attention maps heads to SIMD groups. On first request, the helper walks the existing keys and reuses the pipeline if every constant matches. Otherwise it constructs a Metal function-constant object, requests a specialized function, creates the device pipeline state, and records the key beside it. This is a small software cache around the runtime compiler.

```
1654 static id<MTLComputePipelineState> variant(int k, int wtype, int wtype2,
         int glu, int norm, int tok_sg, int hd, int quick,
1655                                           int plg, int kv8) {
1656     const int c[N_CONST] = {wtype, wtype2, glu, norm, tok_sg, hd, quick,
         plg, kv8};
1657     for (int i = 0; i < n_variants; i++)
1658         if (variants[i].k == k && !memcmp(variants[i].c, c, sizeof c))
        return variants[i].ps;
1659     if (n_variants == MAX_VARIANTS) { fprintf(stderr, "metal: too many
        kernel variants\n"); exit(1); }
1660     @autoreleasepool {
1661         MTLFunctionConstantValues *fc = [[MTLFunctionConstantValues
        alloc] init];
1662         const bool g = glu, nm = norm, qk = quick, pg = plg, k8 = kv8;
1663         [fc setConstantValue:&wtype type:MTLDataTypeInt atIndex:0];
1664         [fc setConstantValue:&wtype2 type:MTLDataTypeInt atIndex:1];
1665         [fc setConstantValue:&g type:MTLDataTypeBool atIndex:2];
1666         [fc setConstantValue:&nm type:MTLDataTypeBool atIndex:3];
1667         [fc setConstantValue:&tok_sg type:MTLDataTypeInt atIndex:4];
1668         [fc setConstantValue:&hd type:MTLDataTypeInt atIndex:5];
1669         [fc setConstantValue:&qk type:MTLDataTypeBool atIndex:6];
1670         [fc setConstantValue:&pg type:MTLDataTypeBool atIndex:7];
1671         [fc setConstantValue:&k8 type:MTLDataTypeBool atIndex:8];
1672         NSError *err = nil;
1673         id<MTLFunction> fn = [library newFunctionWithName:@(kernel_names
        [k]) constantValues:fc error:&err];
1674         id<MTLComputePipelineState> ps = fn ? [device
        newComputePipelineStateWithFunction:fn error:&err] : nil;
1675         if (!ps) { fprintf(stderr, "metal: kernel %s: %s\n",
        kernel_names[k], err.localizedDescription.UTF8String); exit(1); }
1676         [fn release];
1677         [fc release];
1678         variants[n_variants].k = k;
1679         memcpy(variants[n_variants].c, c, sizeof c);
1680         variants[n_variants].ps = ps;
1681         return variants[n_variants++].ps;
1682     }
1683 }
```

Listing 6.4: `gemma4_metal.m`:1654–1683

What does this mean for someone rebuilding only one path? It is useful to start with a single pipeline and a single format. Add another specialization only when a real call needs it. The function constants should have names or documented indices because an index is a contract between the host and the kernel declaration. If the order changes on one side but not the other, the compiler can still produce a valid pipeline for the wrong combination. Compare the host's indices with the kernel's declarations whenever a specialization field changes.

How does one threadgroup finish a reduction?

In the source, `block_sum` reduces each SIMD group's partial values first and stores only one subtotal per group. The explicit barrier is necessary because the next phase reads values written by other groups of lanes in the same threadgroup. A second barrier prevents a fast lane from returning while others still need the shared scratch array. The excerpt's `n` is the threadgroup width and the implementation assumes a multiple of 32.

```
// ---------------------------------------------------------------
    helpers

// sum / min over a threadgroup of n threads (n a multiple of 32, at
    most 1024)
static float block_sum(float v, threadgroup float *tmp, uint tid, uint n
    ) {
    v = simd_sum(v);
    if (tid % 32 == 0) tmp[tid / 32] = v;
    threadgroup_barrier(mem_flags::mem_threadgroup);
    float s = 0;
    for (uint i = 0; i < n / 32; i++) s += tmp[i];
    threadgroup_barrier(mem_flags::mem_threadgroup);
    return s;
}
```

Listing 6.5: `gemma4_metal.m`:12–23

That helper is not a claim that all Apple GPUs expose the same scheduling details. It is a concrete communication pattern in this engine's Metal source. SIMD operations exchange values within one SIMD group; threadgroup memory and a barrier join those groups. A barrier cannot join separate threadgroups. If a whole-grid result is needed, the engine launches another kernel or uses a later command that sees the first kernel's result.

Why does the host pass a grid of groups instead of a count of answers?

Consider a kernel whose output has 1,000 rows, and suppose one threadgroup owns 4 rows. The grid then has 250 threadgroups. Each group receives its position in that grid and derives its first output row from that position. Within the group, each SIMD group and lane derives a smaller part of those four rows. In the Q4 matvec, the source makes the hierarchy visible: `tg` chooses a tile, `sg` chooses rows within that tile, and `lane` chooses a slice of the packed quantized block. The host grid is the count of such tiles, not the count of scalar multiplies.

The hierarchy is a division of responsibility. The host selects a tile size that is useful for a kernel family. The threadgroup assigns work to its lanes. The lane decodes packed values and accumulates its local products. SIMD reduction combines values that live in one group of 32 lanes, and threadgroup scratch combines SIMD-group results. This division means the kernel has to state exactly which lane owns which elements. If two lanes store the same output, there is a race. If no lane owns an output coordinate, it stays unwritten. A bounds check on the last tile prevents padded work from storing past the end of an array.

The simple vector-add experiment hides most of this complexity because one lane owns one output and no lane shares a result. It has no reduction, no threadgroup scratch, and no dependency between positions. That makes it a useful first kernel: the grid dimensions and pointer binding are all the machinery needed. A matvec is harder because many lanes cooperate to produce each row's dot product. Chapters 7 and 8 will show how the engine chooses different ownership patterns for a memory-bound decode and a batched prefill.

The source assumes 32 lanes per SIMD group in several places, including the `tid % 32` test and the 128-thread matvec arrangement. That is an implementation assumption of this backend, not a rule a portable Metal program should leave undocumented. When a kernel depends on a width, make that width explicit in its host dispatch and in the compile-time assumptions. A mismatch can break a reduction even when every buffer address is valid.

What exactly does a no-copy buffer buy us?

The model weights begin as a file mapping created by the C-side GGUF reader. That mapping already gives the process a virtual address for each tensor. On Apple silicon the GPU can read the same physical

PREPRINT

memory through a Metal buffer wrapper, so the engine does not first allocate a second multi-gigabyte array and copy the tensor into it. This is valuable for capacity as well as startup time: there is one backing set of pages, while both CPU and GPU have a way to address it.

The mapping is not the same thing as saying every byte is resident in physical memory. The operating system can page a file-backed mapping in as it is used. The engine's host keeps a list of Metal-visible regions and, where supported, adds them to a residency set. The source comment describes the resident set as keeping dense weights and activations wired. That controls GPU residency for the active working set; it does not mean that every unused GGUF payload is copied or permanently pinned. The C loader chooses which tensor ranges to expose, and the GPU only accesses pages for the dispatched work.

The page rounding in `metal_map` has two separate jobs. First, the start address is rounded down so the Metal buffer begins at a page boundary. Second, the end address is rounded up so the buffer length covers the complete final page. The tensor pointer may then be inside the buffer, and `find_region` computes an offset from the rounded start to the exact requested bytes. Without that offset, every tensor that began after the first byte of its page would be read from the wrong address. Without the larger page wrapper, the requested tensor tail could extend outside the buffer.

Neighbouring tensors may occupy ranges whose rounded pages overlap. The host keeps regions ordered by address and accepts a pointer only when the complete requested span fits inside a region. Thus it can wrap a mapped run of useful tensors once and bind many interior pointers at their individual offsets. This is also why callers pass the requested byte length when it matters: being inside some registered page is not enough if a kernel would read past the end of that page range.

There is an ownership distinction worth keeping straight. A no-copy buffer does not own the file's storage in the way a newly allocated Metal buffer owns its allocation. Releasing the Metal object does not make a still-live mapping safe to discard, and discarding the mapping while a command is using its wrapper is unsafe. The engine's shutdown path must finish GPU work, release the Metal wrapper, and then allow the model mapping to close. For allocations returned by `metal_alloc`, the Metal buffer owns the shared allocation, but the same in-flight rule applies: do not free it while a queued kernel still uses it.

What does the dispatch encode?

By the time the engine calls `encode`, it already has a pipeline, host pointers, parameter bytes, and a grid shape. The helper first checks that each pointer range belongs to a registered Metal buffer. A missing region returns zero before a command is encoded, which gives the caller a chance to use the CPU path. Otherwise it opens a command buffer and concurrent compute encoder on demand, binds buffers and a small parameter structure, and records the threadgroup grid.

```
1802 // Encodes kernel k over `grid` threadgroups of `threads` threads. ptrs[
       i] (host pointers inside GPU-visible
1803 // memory, lens[i] bytes from there on) bind to index i, the parameter
       struct to index n. Returns 0, encoding
1804 // nothing, if a pointer is not GPU-visible.
1805 static int encode(id<MTLComputePipelineState> ps, int k, const void **
       ptrs, const size_t *lens, int n,
1806                   const void *params, size_t params_size, MTLSize grid,
       MTLSize threads) {
1807     id<MTLBuffer> bufs[16];
1808     NSUInteger offs[16];
1809     if (skip[k]) return 1;
1810     for (int i = 0; i < n; i++)
1811         if (!(bufs[i] = find_region(ptrs[i], lens ? lens[i] : 1, &offs[i
       ]))) return 0;
1812     @autoreleasepool {
1813         if (!enc) {
1814             cmd = [[queue commandBuffer] retain];
1815             enc = [[cmd computeCommandEncoderWithDispatchType:
       MTLDispatchTypeConcurrent] retain];
```

```
1816          } else if (!(in_group && group_size > 0)) {
1817              [enc memoryBarrierWithScope:MTLBarrierScopeBuffers];  //
      wait for the previous steps' writes
1818          }
1819          group_size += in_group;
1820          [enc setComputePipelineState:ps];
1821          for (int i = 0; i < n; i++) [enc setBuffer:bufs[i] offset:offs[i
      ] atIndex:i];
1822          [enc setBytes:params length:params_size atIndex:n];
1823          [enc dispatchThreadgroups:grid threadsPerThreadgroup:threads];
```

Listing 6.6: gemma4_metal.m:1802–1823

The grid is expressed in threadgroups, and `threads` is the threads per threadgroup. This is different from dispatching a count of individual threads: if the kernel needs 1,000 output positions and each threadgroup has 128 threads, the host can dispatch eight groups and let the kernel's bounds check ignore excess positions. Individual engine call sites choose grids to match their tiling. The Q4 matvec entry point shows how the shader receives its group index, thread index, SIMD-group index, and lane index; those coordinates let each thread select its share of rows and packed blocks.

```
475 kernel void matvec_q4(device const uchar *W [[buffer(0)]], device const
      uchar *W2 [[buffer(1)]],
476                       device const void *X [[buffer(2)]], device float *
      Y [[buffer(3)]],
477                       device const float *norm_w [[buffer(4)]], constant
       MatmulParams &p [[buffer(5)]],
478                       uint tg [[threadgroup_position_in_grid]], uint tid
       [[thread_index_in_threadgroup]],
479                       uint sg [[simdgroup_index_in_threadgroup]], uint
      lane [[thread_index_in_simdgroup]]) {
480      threadgroup float part[4];
481      const int n0 = (tg * 4 + sg) * MV_R, nw = GLU ? 2 * MV_R : MV_R;
482      float scale = 1;
483      if (NORM) {
484          device const float *xf = (device const float *)X;
485          float ss = 0;
486          for (int i = tid; i < p.K; i += 128) ss += xf[i] * xf[i];
487          ss = simd_sum(ss);
488          if (lane == 0) part[sg] = ss;
489          threadgroup_barrier(mem_flags::mem_threadgroup);
```

Listing 6.7: gemma4_metal.m:475–489

Inside a kernel, a SIMD lane is not a tiny independent CPU thread with its own arbitrary control flow. A SIMD group executes a bundle together. The matvec uses the lane number to select a subrange of bytes and uses `simd_sum` for within-group accumulation. A threadgroup scratch array combines group totals when the operation needs a wider reduction. Thinking in these nested units helps explain both correctness and performance: too few groups may leave GPU cores idle; too many live values can increase register pressure; barriers are needed for communication through threadgroup memory.

How are independent steps grouped?

Metal command buffers are the unit the host submits to a queue. The engine leaves a sequence of dependent steps in one encoder and inserts a buffer memory barrier between steps unless the host explicitly opened a group whose members are independent. Closing a group restores ordinary dependency ordering for later work. Grouping is a promise from the caller that no step in the group reads another member's output; it is not a general "make this faster" switch.

```
1858 void metal_group(int open) {
1859     in_group = open;
1860     group_size = 0;
1861 }
```

Listing 6.8: `gemma4_metal.m`:1858–1861

When the C engine reaches a point where queued work should start, `metal_commit` ends the encoder, attaches a completion handler for GPU-time accounting, commits the command buffer, and retains only the latest in-flight buffer. The queue executes its command buffers in order, so waiting for the last submitted one also waits for earlier buffers on that queue. `metal_sync` commits any unsubmitted encoder and waits for that last buffer. In contrast, `metal_fence` returns a retained handle to the work already submitted; `metal_wait` waits for just that handle and releases it.

When do recorded writes become visible?

"Shared memory" answers where the bytes live; it does not answer when a write has finished. If the CPU writes an input before submitting a command, the command can read it after that write. If the GPU writes an activation and the CPU needs to inspect or update it, the CPU must wait for the relevant GPU work. If one GPU kernel writes data that a later kernel reads, the encoder must order those operations. These are three different edges in the data flow, and the host uses different mechanisms for them.

Inside one threadgroup, `threadgroup_barrier` makes shared scratch writes available across its lanes. It does not wait for another threadgroup. Between encoded kernels, the buffer-scope memory barrier orders reads and writes within the command stream. It does not block the C thread until the GPU has finished. At the CPU boundary, `metal_sync` waits for submitted queue work, while `metal_wait` waits on a retained fence for one submitted point. Keeping these scopes separate prevents the common mistake of using a local shader barrier to solve a host scheduling dependency.

The encoder's normal mode is concurrent dispatch. The host's barrier between dependent steps ensures the later step observes earlier writes. A group intentionally skips the barrier between its members. For example, query, key, and value projections read the same input but write distinct output buffers. They can be encoded as a group because none consumes another's result. The attention preparation step cannot join that group if it reads the query and key outputs: it must be ordered after the projections. A group is therefore a compact statement of a dependency fact, and the fact must be true for every member.

The final lines of this excerpt show that the host records work and then returns. It has not committed the command buffer there, and the GPU may not have started the new dispatch. This deferred behavior is useful: the CPU can record several kernels while the GPU runs earlier submitted work. But it changes how a reader should interpret a function call. A successful `metal_matvec(...)` means the step was accepted into the queue's pending command stream, not that its output is ready to read.

There is a second way to cross the CPU/GPU boundary without waiting for everything. The model's decode loop can submit the GPU work that produces a token and keep a fence for that submitted point. The CPU then prepares later work that does not depend on that token. At the point where it needs the token to select an embedding row, it waits on the fence and reads the result. This lets useful host work overlap with earlier GPU work. A fence is not a future promise that includes commands encoded after it was created; it refers to the work submitted so far. Its lifetime ends when `metal_wait` releases the handle.

Why not submit and wait after every kernel?

The simplest host program would create a command buffer, encode one kernel, commit it, wait for completion, read the output, and repeat. That is easy to reason about but adds a round trip between each short GPU step. A transformer layer has many small operations and dependencies, so a CPU wait at every edge would leave the GPU idle while the host wakes up and records the next operation. The engine instead

PREPRINT

records dependent work into the same command stream, inserts GPU-side buffer barriers between steps, and submits at useful boundaries.

The source tracks a current command buffer and encoder as well as the most recently submitted buffer. When `metal_commit` is called, it ends the current encoder and commits its buffer. It releases its old `inflight` reference and keeps a reference to the just-submitted one. Since a Metal command queue executes its buffers in order, the most recent one is enough to wait for all the earlier buffers on that queue. This design is one queue and one in-flight handle, not a general dependency graph across multiple queues.

```
1867 void metal_commit(void) {
1868     if (!enc) return;
1869     [enc endEncoding];
1870     [enc release];
1871     enc = nil;
1872     [cmd addCompletedHandler:^(id<MTLCommandBuffer> cb) { gpu_busy += cb
     .GPUEndTime - cb.GPUStartTime; }];
1873     [cmd commit];
1874     [inflight release];  // the queue runs buffers in order, so only the
      last one needs waiting for
1875     inflight = cmd;
1876     cmd = nil;
1877 }
1878
1879 static void wait_for(id<MTLCommandBuffer> cb) {
1880     [cb waitUntilCompleted];
1881     if (cb.status != MTLCommandBufferStatusCompleted) {
1882         fprintf(stderr, "metal: command buffer failed: %s\n", cb.error.
     localizedDescription.UTF8String);
1883         exit(1);
1884     }
1885 }
1886
1887 void metal_sync(void) {
1888     metal_commit();
1889     if (!inflight) return;
1890     wait_for(inflight);
1891     [inflight release];
1892     inflight = nil;
1893 }
1894
1895 void *metal_fence(void) { return inflight ? [inflight retain] : NULL; }
1896
1897 void metal_wait(void *fence) {
1898     if (!fence) return;
1899     wait_for((id<MTLCommandBuffer>)fence);
1900     [(id<MTLCommandBuffer>)fence release];
```

Listing 6.9: `gemma4_metal.m`:1867–1900

Notice what each operation promises. `metal_commit` submits pending work and returns without waiting. `metal_sync` submits pending work, blocks until completion, checks for command-buffer failure, and clears the in-flight handle. `metal_fence` retains the current submitted command buffer, if any; it does not force an unsubmitted encoder to be committed. In ordinary flow, the C wrapper calls `gpu_submit` to commit before asking for a fence. This sequence is why command submission and fence creation should be read together at their call site.

The completion handler adds GPU elapsed time after a buffer completes. It is asynchronous bookkeeping and does not make the submit call synchronous. Profiling changes this normal path: the profiling branch ends and commits a buffer for each operation and immediately waits before collecting the GPU

timestamps. That gives a simple per-kernel table, but changes how much command batching and overlap occur. The pitfall box below explains why those timing numbers answer a narrower question than ordinary end-to-end throughput.

How does this remain callable from C?

The engine does not call Objective-C methods itself. It calls plain functions declared in `gemma4_metal.h`, and `gemma4_metal.m` implements those functions around Metal objects. The C wrappers in `gemma4.c` keep the reference path available: synchronize before CPU work that depends on queued GPU writes, and use shared allocation for activations when the GPU is active.

The `STEP` macro makes the fallback rule visible. If the GPU wrapper accepts the arguments and encodes the kernel, the CPU task is skipped. If it returns zero, the engine waits for prior GPU work before the CPU task runs. That barrier is essential: a CPU fallback must not race a previous GPU write to the same activation. It also means correctness can survive unsupported shapes, but a silent fallback can make a benchmark unexpectedly slow. A future chapter will study the kernels themselves; for now, remember to count both accepted and declined operations when diagnosing a performance change.

```
664 // One step: encoded for the GPU if `gpu_call` accepts it, else run on
       the CPU threads.
665 #define STEP(gpu_call, n, cpu_task, job)
            \
666     do {
            \
667         if (!GPU(gpu_call)) { gpu_sync(); parallel_for((n), (cpu_task),
       (job)); } \
668     } while (0)
669
670 // Activations, logits and log-prob buffers: shared with the GPU when
       Metal is on, so nothing is copied.
671 static void *shared_alloc(size_t bytes) {
672 #ifdef GEMMA4_METAL
673     if (use_gpu) return metal_alloc(bytes);
674 #endif
675     return xmalloc(bytes);
676 }
677
678 static void shared_free(void *p) {  // (after gpu_sync, when the GPU may
        have used it)
679 #ifdef GEMMA4_METAL
680     if (use_gpu) { metal_free(p); return; }
681 #endif
682     free(p);
683 }
```

Listing 6.10: `gemma4.c`:664–683

The header's return convention makes declining a step part of the interface. A function returning zero means "I did not encode this operation"; it does not mean that the output was partly computed. The C caller can safely synchronize and compute the complete operation on the CPU. The project's wrapper uses this behavior throughout the backend, not just for one special kernel.

How do the C calls divide up the work?

The public functions in the header fall into four practical groups. Setup and lifetime functions initialize Metal, register model pages, allocate shared activations, and release those resources after the GPU is done. Submission functions commit work, wait for all submitted work, or wait for one fence. Group control brackets independent encoded operations. The remaining functions describe model operations such

as matmul, attention preparation, normalization, and token scoring. Each operation wrapper translates dimensions and pointers into a pipeline, parameter structure, and grid dispatch.

Call family	Examples	What the C caller learns
Setup / memory	`metal_init`, `metal_map`, `metal_alloc`	Whether a device compiled the kernels; whether a pointer is GPU-visible.
Queue control	`metal_commit`, `metal_sync`, `metal_fence`	Whether work is submitted, fully finished, or finished through one earlier point.
Operation	`metal_matmul`, `metal_attention`, `metal_argmax`	Whether this shape and type was encoded for the GPU.
Concurrency	`metal_group`	Whether adjacent operations are independent and may overlap.

This boundary is intentionally narrow. The caller does not pass an `MTLBuffer`; it passes the same host pointer it uses for the CPU implementation. The Metal host finds which registered buffer contains that pointer and computes an offset for the binding. That makes it possible for C data structures to stay ordinary pointers and arrays, but it means allocation provenance matters: an arbitrary `malloc` pointer is not automatically visible to the GPU. Inputs should be allocated by `metal_alloc` or covered by `metal_map` before an operation is attempted.

The API separates shape validation from computation. For example, an attention wrapper can decline a combination of head count, head width, window, cache format, and prompt length even when every pointer is valid. The CPU fallback then uses the same logical tensors and expected output. This is a useful way to grow a backend safely: implement the common case first, keep the reference correct for the remainder, and later add a GPU specialization when its performance matters. However, the fallback is most useful when it is observable. A call-count table grouped by function name will tell whether the workload you think is on GPU actually is.

Why are mapped weights and allocated activations different?

`metal_map` wraps memory the GGUF reader already owns through its file mapping. The source address and bytes exist before Metal is involved, and mapping registers a page-rounded view of that range. `metal_alloc` asks Metal for a new shared buffer and returns its `contents` pointer to C. The two paths produce similar-looking host pointers, but their ownership is different: the engine can release an allocated buffer with `metal_free`, while a mapped model region must also be disconnected from its backing file mapping in the correct order.

The engine uses the allocation path for temporary activations, logits, and other working arrays that both sides need. They are born as shared buffers, so the C code can write them and the Metal encoder can bind them without a staging copy. Model weights are different: copying them into new allocations would duplicate gigabytes of data. Wrapping the existing file pages avoids that copy. In both cases the host inserts the pointer into its region table so a later operation can translate a C pointer into a Metal buffer and an offset.

The call to `metal_alloc(0)` still requests a small legal allocation internally, but registers a logical size of zero. This lets empty tensors have a stable allocation result without asking Metal for a zero-length buffer. A zero-byte range is never a useful kernel input; any operation that needs actual bytes must pass its real length and be rejected if that span is not covered. This is a small example of the host adapter protecting Metal's object rules while presenting a simple pointer interface to C.

What is the engine's end-to-end Metal sequence?

It helps to read the host as a short sequence of state changes rather than as one giant Objective-C function. First, initialization selects the default device, compiles the embedded library, creates the standard pipelines, and makes a command queue. Second, model loading registers selected mapped ranges, while

PREPRINT

activation setup allocates shared buffers. Third, a C operation wrapper chooses a supported kernel specialization and asks the encoder helper to bind pointers and parameters. Fourth, the helper records a dispatch, placing a memory barrier before it when earlier writes may feed it. Fifth, submission commits the recorded command buffer, but does not normally wait. Sixth, synchronization or a fence is used when the CPU reaches a true dependency or when storage is about to be released.

This sequence distinguishes three kinds of success. Initialization succeeds if the device and library are available and the required pipelines compile. A buffer becomes bindable if its complete pointer range belongs to a registered region. An operation succeeds if the requested shape and type are supported and its dispatch is recorded. Passing one check does not imply the next: a valid device can still lack support for a particular head shape, and a valid pointer can still point to a CPU-only array. The integer return from each wrapper is the operation-level answer.

During a forward pass, the host often knows the next dependency before the current kernel has finished. It can record the work and commit at a boundary while the GPU runs. If a later C operation needs output from an unsubmitted dispatch, the host's normal ordered encoding and barriers keep the data on the GPU without a CPU round trip. If a CPU reference step must run, the engine synchronizes first. If only one earlier result is needed, a fence limits the wait to that submitted point. Resource lifetime follows the same logic: free or unmap after the final GPU use, not just after the last C expression that mentions the pointer.

For a reader implementing m7a, a useful debugging trace is therefore: device selected; library compiled; function found; pipeline made; every buffer range found; parameters bound; dispatch recorded; command buffer submitted; wait completed; CPU and GPU values compared. That checklist localizes a failure. A compilation error belongs to the shader source or pipeline variant. A refusal before encoding belongs to shape or pointer validation. A mismatch after a successful wait belongs to indexing, precision, or the kernel's arithmetic. The same trace will help in Chapter 7 when the first real matvec is added.

Experiment

Can a standalone program compile one Metal kernel from a C string, dispatch it, and use a file mapping as an input buffer? This W0 example needs no model or engine binary. It creates a temporary one-page file in the build folder, writes four floats into its mapping, wraps that page with Metal's no-copy buffer constructor, and computes the elementwise sum with a second shared buffer. The host waits before printing and removes the temporary file.

From the book root, build and run it politely:

```
nice -n 10 clang -fno-objc-arc -O2 code/examples/ch06_vecadd.m \\
  -framework Foundation -framework Metal -o code/build/ch06_vecadd
nice -n 10 ./code/build/ch06_vecadd
```

The source is intentionally a complete host program rather than a fragment. Look at the MSL string, the library and pipeline creation, the page-aligned mapped buffer, and the command buffer's commit/wait pair. The four results prove that each dispatched position added its corresponding pair. They do not measure kernel throughput. The engine's own buffer-mapping function follows the same no-copy principle, but groups larger model tensor regions and keeps them resident for repeated use.

The host has only three input facts to keep consistent: the four-element count, the byte length of the float data, and the mapped page length. The buffer wraps a full page but the kernel reads only the first 16 bytes. The second input and output are Metal-owned shared buffers, each exactly large enough for four floats. Binding index zero in the kernel matches index zero in `setBuffer`; the order is an interface just like a C function's argument order. A mismatch can still compile, because the compiler cannot infer which host buffer was intended.

The dispatch uses four threads because there are four answers, and each lane uses its grid position as the element index. The command buffer and encoder are separate objects: the queue creates a command buffer, the buffer creates an encoder, the encoder records a pipeline and its bindings, then `endEncoding`

PREPRINT

closes the recorded compute pass. `commit` submits it; `waitUntilCompleted` makes the CPU read of `cb.contents` safe. Removing the wait would turn the printed output into a race with the GPU.

The temporary mapping is deliberately writable by the CPU so it can populate the input, but the shader treats it as read-only. A real weight mapping is established by the GGUF reader and registered by the engine; the W0 program does not try to load or inspect any model file. This keeps the exercise small while still demonstrating the two key ideas used by the engine: a kernel can be compiled from text at runtime, and a GPU buffer can refer to an existing page-aligned mapping.

Experiment W0

Captured on 2026-10-22 from the command above; this is the actual program output, not an inferred timing result:

```
11 22 33 44
```

No model file was opened and no inference was run. The GPU ran only this four-element dispatch. The sample writes a page-sized temporary file because `newBufferWithBytesNoCopy` requires page-aligned address and length; only the first four floats are read by the kernel. Its exact page size depends on the machine, so the program asks `getpagesize()` rather than assuming a constant.

Pitfalls from the project history

Pitfall

Correct output can hide a CPU fallback. On the E2B, one KV head serves eight query heads. The 8-bit prefill-attention kernel originally accepted only up to four query heads per KV head, so the Metal wrapper returned zero and attention fell back to the CPU during every prefill. The output remained correct, but the notes record 27.7 seconds of CPU time for about 5 seconds of wall time in the bench run. The fix split query heads across threadgroups, respecting the 16-SIMD-group capacity; the prefill rate changed from 450 to 1,652 tokens per second in that documented run. Source: `ENGINE-MATH.md` §12, dated 2026-10-06. The general lesson is to check whether the GPU accepted a call, not only whether the final answer looks right.

Pitfall

A profile can measure the wait instead of the kernel. `GEMMA4_PROFILE=1` commits each kernel in its own command buffer and waits so it can report an individual GPU duration. The project notes record small steps at 190–240 microseconds in that mode, while a normal dispatch floor was much shorter. The per-kernel profile pays the command-buffer round trip and removes batching and concurrency, so those rows cannot be read as ordinary kernel latency. Source: `ENGINE-MATH.md` §11.7 and the comment at `gemma4_metal.m:1624--1625`. Use the profile to locate work, then measure representative throughput with normal submission behavior.

Pitfall

A mapped pointer is still in use until the GPU is done. The header explicitly says `metal_unmap` must happen after the GPU is finished with the region. Releasing the Metal wrapper while a submitted command still refers to it breaks the pointer lifetime contract. Synchronize or wait on the relevant fence before freeing/unmapping. Source: `gemma4_metal.h:13--20` and the comment at `gemma4_metal.m:1740`.

Exercises

Exercises

1. **Read.** In `block_sum`, why does each SIMD group write only when `tid % 32 == 0`? What information would be missing if the second barrier were removed?
2. **Read.** Trace one pointer from a model tensor into `find_region`, then into `setBuffer:offset:atIndex:`. Which address is page-aligned, and which offset identifies the tensor's actual start?
3. **Change.** Change the W0 vector-add program to add eight values. State which host allocation sizes, file contents, and dispatch dimensions must change together.
4. **Change.** Add a second independent kernel dispatch to the sample and record both in one command buffer. Then make the second kernel read the first kernel's output; identify the dependency and the ordering mechanism that protects it.
5. **Measure.** Dispatch the same tiny kernel repeatedly in one command buffer and in separate command buffers. Record wall time and explain why startup, compilation, submission, and synchronization must be kept out of any claim about kernel throughput.
6. **Read.** The C `STEP` macro runs a CPU task when a GPU call returns zero. Explain why it synchronizes first, and propose a counter that would reveal how often fallback occurs.

Rebuild checkpoint

Rebuild checkpoint

Milestone m7a: one Metal host call and one kernel. Implement `steps/m7a_metal_host.m` with initialization, a GPU-visible allocation, a `get_rows` call, command submission, and synchronization. Use the snapshot's C interface and compare the returned embedding rows with the CPU implementation for the same token IDs. The check is value agreement with the CPU rows; a successful pipeline compile alone proves only that the shader can be built. Free the shared buffers only after the queued GPU work has completed.

PRELIMINARY

Chapter 7

Decode kernels: matrix × vector

The question

How does decode stream nearly 2.7 GB of weights for one token without letting unpacking and scheduling hold up the memory bus?

Reading map: 18 pages

The outline assigns 18 pages to this chapter. This expanded plan puts four of those pages into a worked E4B dispatch, the SIMD reduction, arithmetic intensity, and the format comparison; counts include figures, listings, and exercise space.

Subsection	Pages
The question and decode bandwidth floor	1
The idea: bytes, lanes, and one row	3
CPU reference and generic Metal path	2
Packed Q4, Q4_K, and IQ4 kernels	4
Existing dispatch and fused work	1
Real E4B dispatch and lane byte accounting	1
SIMD reduction and threadgroup layout	1
Decode/prefill arithmetic intensity	1
Per-format variants and dispatch	1
W0 experiment	1
Bandwidth arithmetic and documented pitfalls	1
Exercises and m7b checkpoint	1
Total	**18**

The idea

Why does a single generated token touch so much memory? The model's next hidden vector must pass through its weight matrices. For each matrix row, the engine takes a dot product with that one vector; across all rows and layers, it reads almost every GPU-used weight. Chapter 1 used 2.7 GB as a rounded decode traffic estimate and 100 GB/s as an illustrative sustained rate. Dividing gives a best-case floor of 0.027 seconds per token, or about 37 tokens per second, before attention, the head, dispatch gaps, and other work. It is a floor, not a promise.

Now contrast a prompt batch. With many input rows, one weight tile can serve several dot products before it is fetched again. Chapter 8 will follow that reuse. During decode there is only one row of activations. The weights are the large stream, while the activation vector is small enough to be reused from cache. The work per byte is low: the kernel's central job is to keep independent loads in flight and make useful products as bytes arrive.

PREPRINT

A toy four-element row is [2, -1, 3, 4], and its activation is [5, 6, 2, -3]. The four products add to -2. Real rows use packed quantised codes, with one scale shared by each block of 32 or 256 weights. Lanes cooperate on a row, then one lane writes its reduced result.

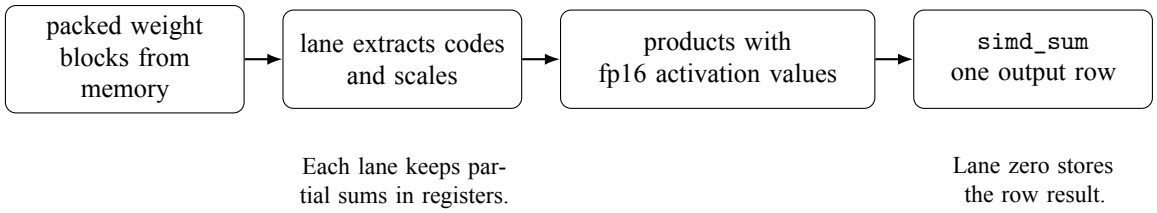


Figure 7.1: One SIMD group streams pieces of packed rows, forms partial dot products, and reduces each row.

How do the 128 threads divide that stream? In the generic and Q4 kernels, a threadgroup contains four SIMD groups of 32 lanes. A SIMD group is the cooperation unit for `simd_sum`; no threadgroup-wide barrier is needed to finish a row's dot product. In the generic path, each SIMD group owns `MV_R = 4` output rows, so one threadgroup covers 16 rows. A lane walks selected blocks of the row and accumulates its contribution in registers. At the end, all 32 lanes call `simd_sum`; lane zero stores the result. The Q4_K kernel instead sets `Q4K_R = 2`, so four SIMD groups cover eight output rows.

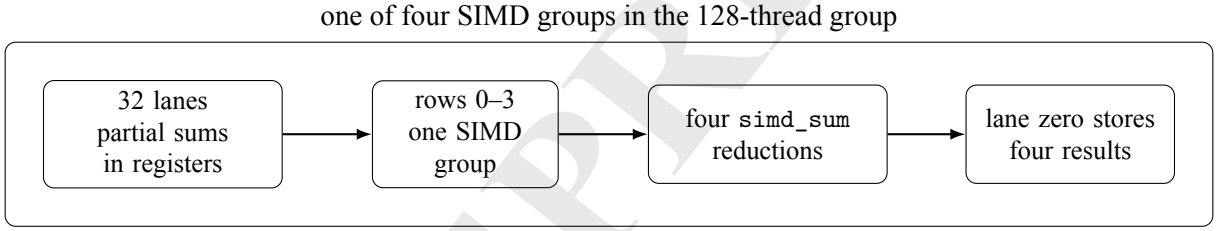


Figure 7.2: A threadgroup is four independent row tiles for the reduction. Each lane keeps one partial accumulator per row assigned to its SIMD group.

This is a two-level map, but only one level communicates during the dot product. The host tiles output rows across threadgroups; within a threadgroup, each SIMD group owns its own rows and reduces them independently. No partial row is split between SIMD groups, so the four groups do not need a shared reduction stage. The larger threadgroup exists to give the kernel four row tiles in flight and share setup such as the IQ4 lookup table. For Q4_K the same four-group shell carries two rows per group, reducing the tile from 16 rows to eight.

A concrete Q4_0 block makes the ownership visible. It stores a 2-byte fp16 scale followed by 16 packed bytes for 32 weights. Suppose the current row begins at byte 0. Lanes 0 and 1 both select block 0 (the block index is `lane / 2`), but take different halves: lane 0 starts at byte 2 of the row and reads code bytes 2–9; lane 1 starts at byte 10 and reads bytes 10–17. Each lane therefore extracts 16 four-bit codes, split across the low and high nibbles, and uses 16 activation values. The scale is at bytes 0–1. Lanes 2 and 3 repeat that pattern on block 1, whose row offset is 18 bytes. The activations and byte slices are aligned by the kernel's `il`, `k0`, and `k1` indices.

The source's arithmetic avoids building a 32-float temporary for Q4_0. It multiplies nibble values by activations, accumulates the activation sum too, then applies the shared scale and the format's minus-eight offset once. For Q4_1 the second fp16 header value is a minimum instead; the row contribution is scale times code-products plus minimum times the activation sum. This is register dequantisation: packed bytes are loaded, interpreted, and used immediately rather than expanded into a second row in device memory.

PREPRINT

The real code

What is the CPU reference doing?

The CPU path is written for reuse across a batch: it dequantises a tile of 16 weight rows into a k-major temporary and reuses it across up to four activation rows. For decode, where `T = 1`, that reuse disappears. The GPU matvec specializes for that case and does not need a large decoded matrix. This excerpt shows the reference tile and makes the difference in work shape explicit.

```
705 static void matmul_tile(void *ctx, int tile, int tid) {
706     const MatmulJob *job = ctx;
707     const int K = job->K, N = job->N, n0 = tile * NB;
708     const int nn = N - n0 < NB ? N - n0 : NB;
709     float *row = job->scratch[tid], *wt = row + K;
710 
711     for (int j = 0; j < NB; j++) {
712         if (j < nn) dequant_row(job->w, n0 + j, row);
713         else memset(row, 0, K * sizeof(float));
714         for (int k = 0; k < K; k++) wt[k * NB + j] = row[k];
715     }
```

Listing 7.1: `gemma4.c:705–716`

The tile is a useful correctness reference, but its layout answers a different question: how can the CPU amortize row conversion and use vector instructions across output columns? The matvec kernel asks how one activation vector can be shared across the lanes while packed weights stream by. The wrapper below first tries the fused deferred-normalization path; otherwise it materializes the input if needed, asks Metal to encode a matrix operation, and falls back to CPU work if Metal declines.

```
776 static void matmul(float *y, const f16 *x, const Tensor *w, int T, int
        accumulate) {
777     const int K = (int)w->ne[0], N = (int)w->ne[1];
778     if (x == deferred_norm.out && GPU(metal_matvec_norm(y, deferred_norm
        .x, deferred_norm.w, deferred_norm.eps, w->data,
779                                                         row_bytes(w), w
        ->type, K, N, accumulate)))
780         return;
781     materialize_norm(x);
782     if (GPU(metal_matmul(y, x, w->data, row_bytes(w), w->type, T, K, N,
        accumulate))) return;
783     gpu_sync();
784     matmul_f32(y, to_float(x, (size_t)T * K), w, T, accumulate);
785 }
```

Listing 7.2: `gemma4.c:776–785`

How does the generic kernel split the row?

The general `matvec` handles float, half, bfloat16, Q8_0, and other block decoders. It uses 8-value vector loads for float-like weights and 32-value blocks for quantised weights. The shared activation is loaded as `half4` groups and converted for accumulation; when input normalization is fused, it instead forms normalized values from the fp32 residual. Each lane accumulates its own partial row sum, and the result is combined with `simd_sum`. The format-specific Q4 path below removes generic branching and maps each lane directly to its packed byte slice.

In that generic kernel, `acc` is private to each lane. A lane starts at its lane index, then advances by 32 blocks, so the 32 lanes cover one block apiece when possible. Each lane reuses activation pieces across its assigned rows. The source continuation chooses each row pointer and type, adds the block contribution to the accumulator, and reduces it. The specialized Q4 listing below shows that reduction and store rule.

Why do lanes take blocks instead of individual weights?

Would it be simpler for lane 0 to read weight 0, lane 1 weight 1, and continue across the row? That mapping is attractive for a plain array, but quantised blocks couple groups of values: one scale applies to 32 codes, and a four-bit code shares a byte with another code. The source assigns complete byte slices of a block so a lane can use each packed byte once and reuse the block scale for all the codes it owns. The loading pattern is also regular across lanes, which helps memory transactions combine nearby addresses.

The activation is laid out differently from the weight. A lane's weight slice represents positions separated by 16 because the low and high nibbles in a Q4_0 code byte represent two halves of the logical block. The kernel arranges four activation values at a time in vector registers, then scales selected copies so nibble bits in their original bit positions can be multiplied without a separate shift. This may look like arithmetic trickery, but the bookkeeping is fixed and the multiply-adds are ordinary operations once the nibble positions are accounted for. Chapter 4 introduced the file layout; here the same layout determines who loads each byte.

Try two bytes by hand. If the first packed byte is `0xA3`, its low and high nibbles are 3 and 10, for positions 0 and 16. If the second byte is `0x5C`, its low nibble is 12 and its high nibble is 5, for positions 1 and 17. With a toy scale of 0.5, Q4_0 converts the first pair to -2.5 and 1.0. If their matching activations are 2 and -1, their contribution is -5 plus -1, or -6. The kernel performs the equivalent calculation across the lane's 16 codes, adds the block's contribution to its private accumulator, then advances to another block. The toy values illustrate the byte map; they are not a measured model block.

The generic kernel supports cases that do not merit a dedicated lane map. Its block decoder can return 32 float weights into a lane-local array, then the kernel takes four float4 dot products against eight activation float4 values. That temporary is local to the lane, not a device-memory buffer. For Q4_0 and Q4_1, the dedicated kernel can do less conversion work by keeping nibble codes packed and factoring the shared scale and offset through the dot product. The tradeoff is source complexity: each optimized format has to prove that its own bit layout and correction terms are applied correctly.

What does a threadgroup cover at the edge?

The grid counts row tiles. If a matrix has 1,000 rows and an ordinary matvec tile covers 16, the host dispatches 63 groups: 62 groups cover 992 rows, and the last computes some padded rows. The store checks `n >= p.N`, so the extra values are discarded. In the GLU case, each valid output row needs both a gate result and an up result. The shader doubles its internal row count but still emits one final hidden value per logical row.

Why not put all 16 rows in one SIMD group? The kernel uses four SIMD groups in a 128-thread threadgroup. Each group can reduce its own partial sums with `simd_sum` and store its rows independently. The threadgroup does not need to synchronize just to combine results from different rows. The groups share the same activation vector, so each group's nearby loads can be served efficiently, but they do not share a row accumulator. The dispatch shape follows the reduction boundary: each output row belongs to one SIMD group, and its 32 lanes contribute partial sums.

There are two distinct synchronization scopes here. `simd_sum` combines values among the lanes of one SIMD group. The optional `NORM` path needs more: each of the four SIMD groups computes one partial sum of squares, lane zero writes it to the four-element threadgroup array, and a threadgroup barrier makes all four values visible before each group computes the common scale. That barrier is not part of the dot-product reduction. It exists because the RMS calculation spans all 128 threads. If normalization has already been materialized, the matvec avoids this extra reduction and reads half activations directly.

This separation also explains why each threadgroup recomputing the normalization can cost time. A row tile is assigned to one group, so every group that processes another set of output rows may repeat the norm work. The fused path saves a separate dispatch and temporary but can add repeated activation reads and reductions. For large weight matrices those bytes may be small compared with weight traffic; for smaller operations, the extra work can be visible. The correct choice depends on the shape and on whether a separate normalized vector would itself cost a dispatch and a write/read round trip.

PREPRINT

What changes for packed Q4 blocks?

The Q4 kernel's comment in the source is a compact map: a lane takes eight consecutive bytes, representing weights `j..j+7` in low nibbles and `j+16..j+23` in high nibbles. `lane / 2` chooses one of the 16 blocks in a step, and `lane % 2` chooses its first or second eight-byte half. The kernel loads row pointers once, then streams blocks in strides of 16. Here the excerpt includes the ownership calculation and the header setup; the arithmetic follows immediately after it in the next listing.

```
475 kernel void matvec_q4(device const uchar *W [[buffer(0)]], device const
    uchar *W2 [[buffer(1)]],
476                       device const void *X [[buffer(2)]], device float *
    Y [[buffer(3)]],
477                       device const float *norm_w [[buffer(4)]], constant
     MatmulParams &p [[buffer(5)]],
478                       uint tg [[threadgroup_position_in_grid]], uint tid
     [[thread_index_in_threadgroup]],
479                       uint sg [[simdgroup_index_in_threadgroup]], uint
    lane [[thread_index_in_simdgroup]]) {
480     threadgroup float part[4];
481     const int n0 = (tg * 4 + sg) * MV_R, nw = GLU ? 2 * MV_R : MV_R;
482     float scale = 1;
483     if (NORM) {
484         device const float *xf = (device const float *)X;
485         float ss = 0;
486         for (int i = tid; i < p.K; i += 128) ss += xf[i] * xf[i];
487         ss = simd_sum(ss);
488         if (lane == 0) part[sg] = ss;
489         threadgroup_barrier(mem_flags::mem_threadgroup);
490         scale = rsqrt((part[0] + part[1] + part[2] + part[3]) / p.K + p.
    eps);
491     }
492     const int ix = lane / 2, il = (lane % 2) * 8;  // block ix of each
    step of 16; bytes il .. il+7 of it
493     const int BS = WTYPE == 3 ? 20 : 18, QO = BS - 16;  // block bytes;
    offset of the nibbles
494     device const uchar *rows[2 * MV_R];
495     for (int r = 0; r < nw; r++) {
496         const int n = min(n0 + r % MV_R, p.N - 1);  // rows past N are
    computed, never stored
497         rows[r] = (r < MV_R ? W + (ulong)n * p.row_bytes : W2 + (ulong)n
     * p.row_bytes2) + il + QO;
498     }
499     float acc[2 * MV_R];
```

Listing 7.3: `gemma4_metal.m`:475–499

For each selected block, the kernel gathers the right activation values, places scaled copies in `yl`, and adds the activation subtotal to `sumy`. The masked nibble products are arranged so the low half contributes without extra shifts and the high half is scaled by 1/16 or 1/4096 to account for its bit position. Q4_0 subtracts eight times the subtotal before multiplying by its scale. Q4_1 adds its minimum times the subtotal. The code stays packed until the multiplication needs each nibble; it does not write a dequantised weight array to memory.

```
501     for (int b = ix; b < p.K / 32; b += 16) {
502         float yl[16], sumy = 0;
503         for (int i = 0; i < 8; i += 2) {
504             const int k0 = b * 32 + il + i, k1 = k0 + 16;
505             float v0, v1, v2, v3;
506             if (NORM) {
```

```
507                 device const float *xf = (device const float *)X;
508                 v0 = xf[k0] * norm_w[k0] * scale, v1 = xf[k0 + 1] *
    norm_w[k0 + 1] * scale;
509                 v2 = xf[k1] * norm_w[k1] * scale, v3 = xf[k1 + 1] *
    norm_w[k1 + 1] * scale;
510             } else {
511                 device const half *xh = (device const half *)X;
512                 v0 = xh[k0], v1 = xh[k0 + 1], v2 = xh[k1], v3 = xh[k1 +
    1];
513             }
514             sumy += v0 + v1 + v2 + v3;
515             yl[i] = v0, yl[i + 1] = v1 / 256.f, yl[i + 8] = v2 / 16.f,
    yl[i + 9] = v3 / 4096.f;
516         }
517         for (int r = 0; r < nw; r++) {
518             device const uchar *q = rows[r] + b * BS;
519             device const ushort *qs = (device const ushort *)q;
520             float a0 = 0, a1 = 0;
521             for (int i = 0; i < 8; i += 2) {
522                 const ushort v = qs[i / 2];
523                 a0 += yl[i] * (v & 0x000F) + yl[i + 1] * (v & 0x0F00);
524                 a1 += yl[i + 8] * (v & 0x00F0) + yl[i + 9] * (v & 0xF000
    );
525             }
526             const float d = float(*(device const half *)(q - il - QO));
527             if (WTYPE == 3) acc[r] += d * (a0 + a1) + float(*(device
    const half *)(q - il - 2)) * sumy;
528             else acc[r] += d * (a0 + a1 - 8.0f * sumy);
529         }
530     }
531     for (int r = 0; r < MV_R; r++) {
532         const float v = simd_sum(acc[r]), u = GLU ? simd_sum(acc[r +
    MV_R]) : 0.0f;
533         const int n = n0 + r;
534         if (lane != 0 || n >= p.N) continue;
535         if (GLU) ((device half *)Y)[n] = half(gelu(v) * u);
536         else Y[n] = v + (p.accumulate ? Y[n] : 0.0f);
537     }
538 }
```

Listing 7.4: `gemma4_metal.m`:501–538

The reduction is still one SIMD operation per output row. A lane with index zero writes the result; the other lanes only contribute. The bounds condition matters for the final partially filled tile, and `GLU` doubles the rows being computed so gate and up values can be combined before they are written. The kernel uses fp32 accumulators even though activations and stored outputs are often fp16; keeping the running sum wider helps preserve accuracy across many blocks.

How should a reader follow the Q4 loop?

The loop variable `b` is a 32-weight block number within the row, not a byte offset. `ix` picks one of 16 block positions for the current pass, and the loop advances by 16 so each lane returns to the same relative block position in the next pass. The row pointer already includes the lane's byte offset inside the selected block. Adding `b * BS` then moves to the corresponding block in that row. For Q4_0, `BS` is 18; for Q4_1 it is 20 because the second fp16 header field stores the minimum.

The eight-byte slice contains 16 codes, but the code uses four ushort loads, each holding two packed bytes. The two nibbles in each byte occupy different bit positions in the ushort. Instead of shifting every

code into the low bits, the kernel masks each nibble in place and scales its activation by the matching power of 16. For example, a high nibble that sits in bits 8–11 gets paired with an activation scaled by 1/256. Multiplying that scaled activation by the unshifted nibble value gives the same contribution as shifting the nibble down and multiplying by the original activation. This exchanges integer shifts for a few activation scalings that can be reused.

For Q4_0, the block has one scale, so the lane combines all of its code products and applies the offset correction through `sumy`. For Q4_1, the two header halves are the scale and the minimum; the scale multiplies code products, while the minimum multiplies the sum of the corresponding activations. The code branches on the specialized `WTYPE` constant, so a compiled Q4_0 pipeline does not need to keep a runtime format test in each inner iteration.

A useful correctness check follows the same ownership map. For a single block, decode all 32 values with the scalar m3 decoder. Sum positions 0–7 and 16–23 using the activations selected by lane 0; compare that partial result with the lane's unscaled contribution before the block correction. Repeat for lane 1, then add all lane contributions and compare against a full 32-value CPU dot product. This catches a reversed nibble order or a wrong activation offset before either the GPU reduction or a large matrix hides it.

DRAFT

How does Q4_K change the byte map?

Q4_K uses one 144-byte super-block for 256 values: a 2-byte scale, a 2-byte minimum scale, 12 bytes of packed local scales and minima, and 128 bytes of nibbles. A lane maps to a super-block and sub-block with `ix`, `iq`, and `ir`. It loads the small header once, extracts the six-bit scale/minimum values, and reads the lane's parts of two nibble runs. The arithmetic combines four 32-weight groups and applies the local scale and minimum to their dot products.

```
            // three wide loads: the header (d, dmin, the 12 scale bytes) and this lane's 8 bytes of each nibble run
            device const uchar *blk = rows[r] + ib * 144;
            const uint4 hd = *(device const uint4 *)blk;
            const ushort4 q1 = *(device const ushort4 *)(blk + 16 + 32 * iq + 8 * ir), q2 = *(device const ushort4 *)(blk + 80 + 32 * iq + 8 * ir);
            const ushort s0 = hd.y >> 16 * iq, s2 = hd.z >> 16 * iq, s4 = hd.w >> 16 * iq;  // scale bytes 2iq.., 4+2iq.., 8+2iq..
            ushort sc16[4];
            sc16[0] = s0 & 0x3f3f;
            sc16[1] = s2 & 0x3f3f;
            sc16[2] = ((s4 >> 0) & 0x0f0f) | ((s0 & 0xc0c0) >> 2);
            sc16[3] = ((s4 >> 4) & 0x0f0f) | ((s2 & 0xc0c0) >> 2);
            thread const uchar *sc8 = (thread const uchar *)sc16;
            float4 a1 = 0, a2 = 0;
            for (int i = 0; i < 4; i++) {
                a1 += float4(yl[2 * i], yl[2 * i + 1], yl[2 * i + 8], yl[2 * i + 9]) *
                      float4(q1[i] & 0x000F, q1[i] & 0x0F00, q1[i] & 0x00F0, q1[i] & 0xF000);
                a2 += float4(yh[2 * i], yh[2 * i + 1], yh[2 * i + 8], yh[2 * i + 9]) *
                      float4(q2[i] & 0x000F, q2[i] & 0x0F00, q2[i] & 0x00F0, q2[i] & 0xF000);
            }
            const half2 dd = as_type<half2>(hd.x);
            const float d = dd.x, dmin = dd.y;
            acc[r] += d * ((a1[0] + a1[1] / 256.f) * sc8[0] + (a1[2] + a1[3] / 256.f) * sc8[1] / 16.f +
```

```
738                                 (a2[0] + a2[1] / 256.f) * sc8[4] + (a2[2] +
       a2[3] / 256.f) * sc8[5] / 16.f) -
739                             dmin * (sumy[0] * sc8[2] + sumy[1] * sc8[3] + sumy
       [2] * sc8[6] + sumy[3] * sc8[7]);
740                 }
741             }
```

Listing 7.5: `gemma4_metal.m`:717–741

IQ4_XS takes a different route. It also has 256 values per 136-byte super-block, but each 32-value block has a signed six-bit local scale and its nibble values index a nonuniform 16-entry level table. The IQ4 kernel in `gemma4_metal.m:642--678` prepares a 256-entry pair lookup table in threadgroup memory, so one byte retrieves its low- and high-nibble levels together. The lane multiplies those half-precision levels by activation slices and applies the block scale. This is a useful reminder that "dequantise in registers" describes where values are consumed, not one universal unpacking instruction sequence: Q4 uses masks and scale arithmetic; IQ4 uses a lookup table.

How does the Q4_K lane map change?

The Q4_K kernel has a wider super-block, so its lane coordinates are divided differently from Q4_0. In the source, `ix = lane / 8` selects one of four 256-value super-block positions in a pass. Within those eight lanes, `iq` selects one of two groups of four 32-value sub-blocks, and `ir` selects one of four eight-value slices. The activation base `k0` combines the super-block offset, the sub-block group, and the slice. Each lane reads corresponding pieces from two nibble runs, one for the first 128 values and one for the second 128.

This mapping is chosen around the stored layout. The 128 nibble bytes are arranged in four 32-byte runs. For each 256-weight super-block, the low nibbles cover one set of 32-value blocks and high nibbles cover their paired blocks. Two ushort4 loads give the lane sixteen bytes in total: eight from each nibble half. The header arrives as a uint4, which includes the base scale, minimum scale, and packed local scale metadata. The code extracts the six-bit fields once, then reuses them for four accumulation groups.

The output contribution has both a scaled code sum and a minimum correction. The eight sub-blocks each have local scale and minimum factors, so a lane forms four partial code sums against its activation slices and four activation sums for the offset part. The scale for each group is the product of the super-block scale and its local scale; the minimum term combines the minimum scale and local minimum. Applying these after the partial dot products keeps the loaded nibble codes compact. The exact masks in lines 721–726 are easy to misread, which is why the header layout in chapter 4 and the source range here should be read side by side.

What is special about IQ4_XS?

IQ4_XS uses a 136-byte super-block. Its first two bytes store the base fp16 scale, the next two hold two-bit high portions of local scales, four further bytes hold local scale nibbles, and the final 128 bytes hold the quantised codes. Each local scale combines its four-bit low field with its two high bits, then subtracts 32 to obtain a signed value. This means a six-bit scale can be negative even though the stored nibble and high-bit fields themselves are unsigned.

A byte contains two quantiser indices. The 256-entry table can store every possible byte as one pair: low-nibble level and high-nibble level. A lane loads a 32-bit piece containing four packed bytes and makes two dot products, one with the first half of the activation block and one with the second. The code converts the lookup entries from half to float for its fp32 dot-product accumulators. The levels are nonuniform: codes are more closely spaced near zero than far away. That can represent small weights more finely, at the cost of a lookup and more complicated block metadata.

The specialized kernel accepts IQ4_XS only when the row start and row stride satisfy alignment checks. That is not an arbitrary host restriction: it reads a 32-bit word directly from the packed code area. If a matrix slice begins at an incompatible address, the same word load could be misaligned. The dispatch

PREPRINT

therefore selects the IQ4 kernel only for aligned pointers and row lengths; another path must handle other shapes. Alignment is part of the data contract between the GGUF mapping, the tensor slice, the host, and the MSL pointer arithmetic.

Why are there several kernels instead of one?

The host's matrix wrapper chooses a pipeline from shape, type, and whether a second matrix is fused. A single token selects a matvec path; more than one token selects the batched `matmul`. Q4_0 and Q4_1 share the specialized Q4 kernel, Q4_K has its own kernel, and IQ3_S / IQ4_NL / aligned IQ4_XS have their lookup or packed variants. Other formats use the generic matvec decoder. Function constants specialize weight types, fused gate/up mode, and normalization so branches can be removed at pipeline creation.

```
1947      const int k = T > 1                      ? K_MATMUL
1948                    : (wtype == 2 || wtype == 3) && same ? K_MATVEC_Q4
1949                    : wtype == 12 && same    ? K_MATVEC_Q4K
1950                    : wtype == 21 && same && K <= 3072 ? K_MATVEC_IQ3S  //
        32 · 32 · IQ3S_J
1951                    : (wtype == 20 || (wtype == 23 && iq4xs_aligned)) &&
       same ? K_MATVEC_IQ4NL
1952                                             : K_MATVEC;
1953      id<MTLComputePipelineState> ps = variant(k, wtype, glu ? wtype2 :
       wtype, glu, norm_w != NULL, tok_sg, 0, quick, 0, 0);
1954      if (profiling)
1955          snprintf(prof_key, sizeof prof_key, "T%-4d K%-5d N%-6d %s%s", T
       > 1 && T <= 64 ? (T + 31) / 32 * 32 : T, K, N,
1956                   wtype == 2 ? "q4" : wtype == 3 ? "q4_1" : wtype == 12 ?
        "q4k" : wtype == 8 ? "q8" : wtype == 20 ? "iq4nl" :
1957                   wtype == 21 ? "iq3s" : wtype == 23 ? "iq4xs" : wtype ==
        30 ? "bf16" : wtype == 1 ? "f16" : "f",
1958                   w2 ? " glu" : "");
1959      const size_t lens[5] = {(size_t)N * row_bytes, glu ? (size_t)N *
       row_bytes2 : 1, 1, 1, 1};
1960      const void *ptrs[5] = {w, glu ? w2 : w, x, y, norm_w ? (const void
       *)norm_w : x};
1961      if (T == 1) {
1962          // 4 simdgroups × IQ3S_RSG, IQ4NL_R, Q4K_R, MV_R
1963          const int rows_tg = k == K_MATVEC_IQ3S ? 32 : k ==
       K_MATVEC_IQ4NL ? 16 : k == K_MATVEC_Q4K ? 8 : 16;
1964          return encode(ps, k, ptrs, lens, 5, &p, sizeof p, size2((N +
       rows_tg - 1) / rows_tg, 1), size2(128, 1));
```

Listing 7.6: `gemma4_metal.m`:1947–1964

The dispatch uses 128 threads per group, or four 32-lane SIMD groups. For ordinary matvec it assigns 16 output rows to a threadgroup; the Q4_K row count is different because its kernel has two rows per SIMD group. The grid is therefore a count of row tiles, not the number of individual output elements. The source kernel's `tg` selects a tile and `sg` selects the rows within that tile.

Fusions remove extra passes through the activation. `NORM` computes the RMS scale from the fp32 residual inside each threadgroup, then applies the norm weights as it loads activations. `GLU` computes gate and up rows together, applies GELU to the gate result, and multiplies by the up result before storing. Some variants clamp in the store; a PLE gate path can multiply by an extra vector there. These fusions save intermediate writes and dispatches, but their conditions are explicit in the host checks. A kernel only accepts shapes and type combinations it can safely interpret; returning zero lets the C wrapper use its fallback.

The host type numbers are internal codes, not GGUF names. The following map makes the kernel choice easier to follow when reading the dispatch. It is a reading aid for the source's constants and branch conditions, not a complete list of every format the loader understands.

Source type	Decode kernel	What the lane consumes
Q4_0 / Q4_1	`matvec_q4`	paired nibbles, block scale (and minimum)
Q4_K	`matvec_q4k`	256-value block and local scales
IQ3_S	`matvec_iq3s`	grid index, sign bits, scale nibble
IQ4_NL / aligned IQ4_XS	`matvec_iq4nl`	byte-pair lookup and local scale
F32 / F16 / BF16 / others	generic `matvec`	float-like or decoded block values

Q8_0 also reaches the generic matvec in this source snapshot. The format has an especially simple block: 32 signed bytes and one fp16 scale. It is the recommended first rebuild in m7b because its scalar decode is straightforward and it avoids the nonuniform lookup tables or packed scale fields of later variants. A later optimized Q8 kernel could use the same row ownership and reduction design while replacing the generic dispatch branch with direct signed-byte loads.

How do other formats fit this design?

The word "variant" covers both different byte decoders and different work layouts. The generic `matvec` is the catch-all for common floating-point types and block formats with a suitable 32-value decoder. `matvec_q4` specializes the two 32-value Q4 formats. `matvec_q4k` understands the 256-value super-block and its packed six-bit local scales. The IQ4 kernel uses a lookup table because its 16 code levels are nonuniform. IQ3_S has its own grid and sign unpacking, and is restricted to small enough K to fit its preloaded activation slices.

The lookup table's location matters. IQ4's code index differs by lane, so reading a constant-address-space table can serialize those divergent lookups. The kernel copies the 256 possible byte pairs into threadgroup memory at dispatch start; lanes then read their chosen entries locally. The table is shared by all four SIMD groups in the threadgroup. Its setup cost is paid once for many row-block calculations, and the decoded values are held in half precision because the multiply path can consume half values without first expanding an entire row to floats.

The launch cache specializes these kernels by format, fusion mode, and normalization choice. The host's `variant` helper creates a pipeline the first time a combination is needed and reuses it later. A specialized pipeline can treat `WTYPE`, `GLU`, and `NORM` as fixed values, which simplifies inner loops. This moves work to first use and consumes a slot in a bounded cache, so "one kernel per format" is not the full count: each supported combination can have its own compiled pipeline.

The host also validates the shape before encoding. Quantised decode requires K to be a multiple of 32, super-block formats require K divisible by 256, and fused two-matrix paths require compatible row counts and types. If a request falls outside those conditions, the Metal wrapper returns zero and the C layer executes its reference path. This contract keeps specialization narrow: a fast kernel only needs to handle combinations it has actually checked, while the engine still has a defined path for everything else.

How does each lane's accumulator become an output?

A lane's accumulator is not yet the answer for its row. It contains only the products from the blocks that lane visited. At the end of the loop, `simd_sum(acc[r])` combines the 32 lane-local partial sums for row `r`. Every lane executes the reduction and receives the same total, but only lane zero continues to the store. That avoids a write race while keeping the reduction in the SIMD group, where the hardware can exchange values without a shared-memory array and barrier.

In GLU mode, there are two accumulator sets. The first set belongs to gate rows and the second to the matching up rows. After reduction, the kernel applies GELU to the gate value and multiplies by the up value, then stores the combined half result. This is why the Q4 source allocates `2 * MV_R` accumulators even when it emits only `MV_R` outputs. Without fusion, the engine would need to store both intermediate vectors, launch another operation to combine them, and read them back. The fused version trades extra live registers for fewer intermediate memory operations and fewer encoded steps.

The generic kernel has additional output choices. A clamp, when present, is applied just before storing. Accumulation mode adds the new projection result to a value already in `Y`; this is used when the

PREPRINT

mathematical result is a sum of projections. A PLE gate can also multiply its row result by a separate vector. These conditions share the same reduction because the expensive part is forming the dot products. Specialization makes unused branches constant conditions, while the store logic expresses the final meaning of the projection.

The use of fp32 for partial sums is deliberate. Even if each activation is fp16 and each weight decodes to a small number, a row can contain thousands of products. Adding them in fp16 would round each intermediate total and discard small contributions repeatedly. The kernel converts the half activation vector to float registers and keeps `acc` in float until the result is stored or explicitly cast to half. That keeps the data compact at rest while leaving accumulation headroom.

What can be checked before measuring speed?

A source reading can catch several classes of bug without opening the model. First, verify the layout assumption: for a 32-value block, the block pointer advances by the correct byte stride, and the format's scale and optional minimum sit at the documented offsets. Second, verify lane coverage: the lane indices must cover every packed code exactly once, with no duplicate or omitted activation position. Third, verify the correction: symmetric Q4_0 needs the minus-eight term, while affine Q4_1 needs its minimum term. Finally, verify the final tile: invalid rows can be computed safely but must not be stored.

A tiny hand-constructed block is a good first test. Fill all activations with a small set of distinct values rather than all ones, assign code bytes with distinct low and high nibbles, and choose an exactly representable scale such as 0.5. Decode the block with the scalar reference and compare the full dot product. Then compare the sum of lane-owned partials using the same mapping. Distinct values expose a swapped nibble half that a vector of all ones could hide. This test is not a substitute for comparing against real model matrices, but it isolates the address arithmetic and the code formula from pipeline dispatch and full-model complexity.

Before timing, test one full row and a row count that is not a multiple of the threadgroup tile. A wrong row stride may still give plausible values for row zero; checking later rows reveals whether the host's `row_bytes` value and the kernel's pointer arithmetic agree. For a super-block format, include multiple 256-value blocks so both the per-super-block base and the sub-block selection change. For GLU, compare gate and up results separately against CPU references before testing the fused activation. These small checks make performance experiments more informative: if the numbers are wrong, the problem is likely format indexing or a host/kernel contract rather than bandwidth.

Experiment

Can a tiny synthetic kernel demonstrate the bytes-per-second calculation without loading any model weights? This is W0. The example creates a 1024-by-4096 fp16 matrix filled with 1.0 and a 4096-element fp16 vector also filled with 1.0, then runs a straightforward GPU row dot product five times. It reports the median wall time and counts one matrix read plus one vector read. Since the matrix is 8,388,608 bytes and the vector is 8,192 bytes, the vector traffic is small beside the matrix. This is a demonstration of the arithmetic and a sanity check that the kernel returns 4096 per row; it is not comparable to the engine's packed kernels.

Compile and run from the book root:

```
nice -n 10 clang -fobjc-arc -framework Foundation -framework Metal \
  code/examples/ch07_bw.m -o code/build/ch07_bw
nice -n 10 code/build/ch07_bw
```

The output captured on 2026-10-22 was:

PREPRINT

Experiment W0

```
synthetic fp16 matvec: 1024 x 4096, median-of-5 0.585 ms wall, 14.3 GB/s, y[0]=4096
```

This toy, this machine, not the engine. Captured on Apple M3, 2026-10-22; system load average was 4.63, and the post-run `ioreg` GPU device-utilisation sample was 35%, so this was not a quiet-GPU capture. The timer includes command submission and waiting, and the bandwidth estimate counts matrix plus vector bytes once; the kernel's repeated activation reads may be served from cache. This is the median of five same-kernel wall-time runs, not GPU time or an interleaved p10. The example omits weight quantisation, fusion, engine dispatch, and engine data sizes, so 14.3 GB/s says nothing about the engine's achieved bandwidth. The source is `code/examples/ch07_bw.m`; it writes no model data and opens no model file.

Bandwidth arithmetic and real dead ends

What does "near the memory limit" mean numerically? The allowed bandwidth rule is a lower bound: time cannot be less than bytes divided by bandwidth. In the chapter 1 example, 2.7 GB at 100 GB/s takes at least 0.027 s. At 102 GB/s, the same traffic takes about 0.0265 s. The kernel benchmark reports achieved GB/s from each kernel's weight bytes and elapsed GPU time; the project notes record 91–93 GB/s for large kernels on a quiet run, close to the estimated 102 GB/s peak. That figure is a historical reference from the snapshot notes, not a measurement made here.

The same notes preserve two useful negative results. First, normalizing the input independently inside each Q4_K LM-head threadgroup measured 74 GB/s; computing the fp16 input once and reusing it measured 93 GB/s. Repeating normalization work and input traffic can cost more than a separate preparatory step. Second, an IQ4_XS gate/up variant with eight rows per SIMD group had too few threadgroups and measured 324 microseconds versus 306 microseconds with four rows. Two rows were within noise. These are documented source-history results, not fresh results; the sources are `README.md` and `ENGINE-MATH.md` §§11.2 and 11.7. The lesson is to measure occupancy and data movement together, because a larger tile can reduce setup but leave too little parallel work.

What does a bandwidth result say, and what does it leave out?

A kernel bandwidth number divides the number of bytes attributed to its weights by the measured GPU duration. It is useful for large matvecs because the weights dominate the traffic and are too large to remain in the fast caches from one unrelated matrix to the next. It is not a direct DRAM-counter reading: cache policy, page state, row metadata, activation reuse, and the benchmark's timing boundary all shape the estimate. A 93 GB/s result against a 102 GB/s estimated peak is evidence that the kernel is close to the practical roof, not proof that every byte took the same route through memory.

The arithmetic also predicts when more arithmetic tuning is unlikely to help. A decode weight contributes one multiply and one addition for one activation, so the useful floating-point work is only a few operations per stored byte. The engine notes compare this low intensity with the machine's compute-to-bandwidth ratio and conclude that the large decode kernels are bandwidth-bound. Once their code unpacking is fast enough to keep the memory requests flowing, reducing a few instructions may not change elapsed time. Reducing bytes can matter more, but changing a format can also change model output quality; that is a separate accuracy decision.

Decode's arithmetic intensity makes that conclusion more concrete. Count a multiply and an add as two floating-point operations. A single matvec performs about two operations per weight. A Q4-style weight occupies roughly half a byte, so even before scale metadata the ratio is around four operations per byte; IQ4_XS is 136 bytes for 256 values, or about 0.53 byte per weight, giving about 3.8 operations per byte. The activation vector is reused, but the weight stream dominates this count. At a 100 GB/s bandwidth roof that intensity supports only about 0.4 TFLOP/s, far below the GPU's peak compute rate.

PREPRINT

In prefill, a weight tile serves many token rows. For a tile of 32 tokens, each stored weight supports 32 multiply-add pairs, or 64 operations, before it must be fetched again. With the same 0.53 byte IQ4_XS representation that is about 120 operations per byte, before counting activation and output traffic. The intensity rises with the number of tokens until other limits matter. This is why chapter 8 changes the question from "how quickly can one row stream weights?" to "how much reuse can a tiled matrix multiply extract from each weight load?"

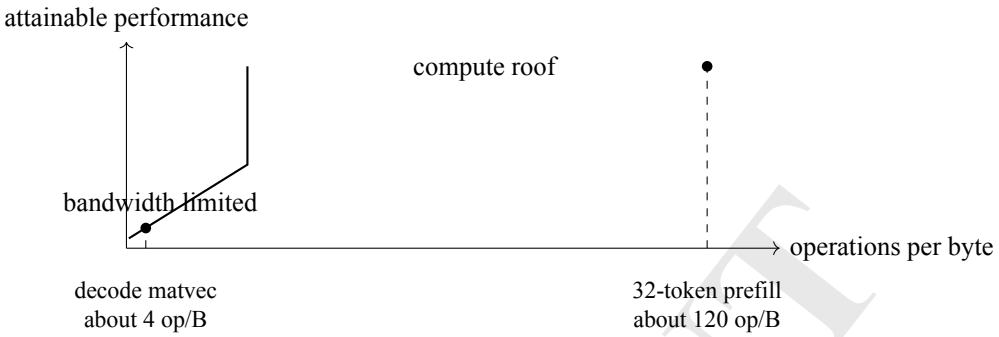


Figure 7.3: A schematic roofline, not a benchmark. Reuse moves prefill to the right by amortising each weight load across token rows.

The historical kernel figures were collected with `tools/kbench`, which creates several representative matrices and measures kernels without running a generation loop. It reports microseconds and GB/s for the LM head, gate/up, down, and attention projections. The experiment is W2 because it opens the model file to construct realistic tensors. As a reader experiment, build `tools/kbench` as described in its header, use `KB_PASSES=7`, keep the GPU quiet, record the idle gate and machine/date, and report the median or p10 rather than a single lucky run. Do not infer a whole-token rate by adding independent best-of-seven kernel times unless the benchmark conditions and timing scopes match.

The shape choice is similarly a measured compromise. More rows per SIMD group can reuse each block's header across more outputs, but it creates more accumulators and reduces the number of independent groups in flight. Fewer rows increase parallel groups and may help small matrices, but repeat loads and setup. The project notes report that two rows were within noise and eight rows were slower for the IQ4_XS gate/up case. Those results explain the current four-row choice for that kernel family; they do not prove four is optimal for every matrix size or GPU generation.

Can the bandwidth floor be checked for one real kernel?

The outline's gate/up example gives a useful check against a recorded kernel result. Its two weight matrices account for about 27.8 MB, and the historical quiet run records 306 microseconds. In decimal units, 27.8 million bytes divided by 0.000306 seconds is about 90.8 billion bytes per second, or 90.8 GB/s. That agrees with the rounded 91 GB/s value. The agreement is a consistency check: it says the chosen byte count and time produce the stated throughput. It does not show how quickly an arbitrary user machine will run the same kernel.

This also explains why the same number of bytes can produce a different token rate than the simple chapter 1 estimate. The 2.7 GB calculation treats all bytes as one continuous stream at one sustained rate. Real decode launches many matvecs of different sizes and has small steps between them. The LM head is a large transfer; a small attention projection may finish before the memory system reaches its steady rate. Adding their bytes and dividing by a single high bandwidth therefore predicts a lower bound for the large streams more than a complete timing model for the whole token.

PREPRINT

What does a real E4B dispatch look like?

Chapter 3's W1 directory probe records `blk.0.attn_q.weight` as IQ4_XS with shape `[2560, 2048]` and 2,785,280 bytes. Here K is 2,560 values per row and N is 2,048 output rows. IQ4_XS stores eight 32-value blocks in each 136-byte super-block, so each row uses ten super-blocks: 1,360 bytes. That gives 2,048 rows times 1,360 bytes, exactly the probe's tensor size. The row count is divisible by the specialized tile, so this example has no padded output rows.

The C host sees T=1, type 23, an aligned pointer, matching single-matrix mode, and a supported K. The IQ4 branch at `gemma4_metal.m:1951` therefore selects `K_MATVEC_IQ4NL`; the name is shared with IQ4_NL because the compiled kernel handles both through `WTYPE`. The `variant()` call that creates the selected pipeline is at line 1953. At line 1963 the host chooses 16 rows per threadgroup. It dispatches 2,048/16 = 128 threadgroups, each with 128 threads: four SIMD groups, 32 lanes each. Each SIMD group owns four output rows, each threadgroup computes 16, and the grid covers all 2,048 rows.

For one output row, K/32 gives 80 quantized blocks. Each lane starts at `b = lane/4` and advances by eight. Thus each lane visits ten blocks in that row. For a concrete address, lane 13 has `qq=1` and starts at block 3; it visits blocks 3, 11, 19, up to 75. In block 3 it reads the four code bytes at offsets 60–63 within the first 136-byte super-block: 8 bytes of header, 3 earlier 16-byte code groups, then the 4-byte lane slice. In block 11 it reads offsets 196–199, the same position in the next super-block. In each visited block it also loads a 2-byte base scale, 2-byte high-scale field, and 1-byte low-scale field. Each lane therefore reads 40 code bytes and issues 50 metadata-load bytes per row; those metadata loads repeat across lanes and blocks, so they are not unique bytes. Each lane participates in four rows within its SIMD group, hence issues 160 bytes of code loads and 200 bytes of scale metadata loads for its four row accumulators. The 32 lanes collectively own the 1,280 code bytes per row exactly once; the unique metadata is 80 bytes per row. So the one-pass tensor payload is still 2,785,280 bytes, while the shader's source-level load instructions request more bytes because header fields are redundantly loaded. Cache and memory transaction behavior determine how much of that repetition reaches external memory.

For the recorded gate/up case, the same accounting doubles because the kernel reads both matrices; for this attention projection it does not. The chapter 1 bandwidth arithmetic predicts a floor of 2,785,280 / 100,000,000,000 = 27.9 microseconds if the entire tensor streamed at 100 GB/s. This is an arithmetic lower bound, not an observed latency for this smaller projection. It also explains why the much larger gate/up benchmark's 27.8 MB and 306 microseconds corresponds to 90.8 GB/s: the numerator there includes both matrices, while this single tensor's unique payload is one tenth as large.

Why does the reduction stay inside a SIMD group?

The row's 32 lane partial sums are reduced with `simd_sum`, as the source does at `gemma4_metal.m:532`. Conceptually the operation combines neighboring pairs in five stages, shrinking 32 values to 16, then 8, 4, 2, and finally one total. Metal specifies the SIMD-group reduction result; it does not promise this exact physical butterfly schedule, so the diagram shows the logical tree only.

Here is the store boundary in the kernel: every lane receives the reduction result, then the lane and row checks allow only lane zero to write a valid output.

```
531      for (int r = 0; r < MV_R; r++) {
532          const float v = simd_sum(acc[r]), u = GLU ? simd_sum(acc[r +
       MV_R]) : 0.0f;
533          const int n = n0 + r;
534          if (lane != 0 || n >= p.N) continue;
535          if (GLU) ((device half *)Y)[n] = half(gelu(v) * u);
536          else Y[n] = v + (p.accumulate ? Y[n] : 0.0f);
537      }
538  }
```

Listing 7.7: `gemma4_metal.m`:531–538

PREPRINT

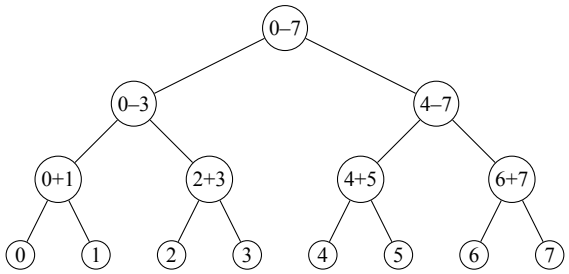


Figure 7.4: Eight lanes shown; `simd_sum` applies the same pairwise pattern to all 32 lanes in five combine stages.

The tree picture suppresses the other branches, but the code captures the key boundary: each row calls one SIMD-group reduction and only lane zero stores. A threadgroup-memory reduction would require every lane to write a partial, a barrier, one or more lanes to read and combine them, and often another barrier before reuse. That mechanism is necessary when values cross SIMD-group boundaries, as in the RMS helper's four partials, but it is unnecessary for a row whose 32 contributors already belong to one SIMD group. Keeping the dot product in registers until `simd_sum` avoids shared-memory traffic and synchronization on the hot path. The threadgroup still has a small shared lookup table for IQ4 byte decoding; that is independent of the row reduction.

Which details change with the quantised format?

The row ownership and final reduction remain recognizable, but the bytes and scale rules differ. The table compares the decode variants in this source snapshot; line ranges point to the kernels and host selection that define the behavior.

Format	Block	Unpacking in the kernel	Scale handling	Source lines
Q4_0 / Q4_1	32 values; 18 / 20 B	Paired nibbles; lane reads an eight-byte slice	Q4_0: fp16 scale and offset -8; Q4_1: scale plus fp16 minimum	475–538; 1948, 1953
Q4_K	256 values; 144 B	Nibbles plus packed six-bit local scale/minimum fields	Global scale/minimum multiplied by each local field	717–741; 1949, 1953
IQ4_XS	256 values; 136 B	Four-byte code word, byte-pair lookup into 16 nonlinear levels	fp16 base times signed six-bit local scale	642–678; 1951, 1953
IQ3_S	256 values; 110 B	Grid indices and sign bits, with a preloaded table	Base scale and per-sub-block scale nibble	590–635; 1950, 1953
Generic / Q8_0	32-value blocks vary	Scalar/block decoder returns lane-local values; Q8 uses signed bytes	Decoder applies each block's fp16 scale	421–468; 1952–1953

These are the distinctions that survive into the fast path: block size determines row-byte stride and loop count; code layout determines lane loads; scale representation determines the correction arithmetic. The C dispatch first checks token count, type compatibility, K alignment, and pointer alignment, then chooses a specialized pipeline or the generic matvec. The host range in the last column is shared with the branches described above; it is not one branch per row of the table.

The same K makes the storage tradeoff easy to compare. At K=2,560, Q4_0 uses 80 blocks of 18 bytes, or 1,440 bytes per row; Q4_1 uses 1,600 bytes because its block carries a second half scale for the minimum. Q4_K also uses 1,440 bytes per row: ten 256-value blocks at 144 bytes each. IQ4_XS needs 1,360 bytes, while IQ3_S needs 1,100. The exact rates differ because metadata is amortised over different block sizes and because the code widths and tables differ. These row sizes predict bandwidth demand, but they do not by themselves predict latency: a kernel also needs enough independent rows to fill the GPU.

PREPRINT

The generic route is not a lesser correctness path. It is the fallback for formats whose block decoder already expresses the right values, including Q8_0's 32 signed bytes and fp16 scale. A dedicated kernel is useful when it changes an important cost: it may avoid building a lane-local 32-float array, exploit a byte layout directly, or move a divergent lookup into a shared table. Each specialization also creates another correctness surface and compiled pipeline variant. The dispatch keeps the fast paths narrow, and the CPU implementation remains available when type, alignment, or shape checks reject one.

For an individual kernel, keep numerator and denominator aligned. Use the bytes of the weight matrices that the kernel actually reads, including both gate and up when fused, and the GPU duration of that encoded kernel. Do not divide by wall time from a prompt-generation command if the numerator includes only weights. Conversely, a wall time around compile, submit, execute, and wait answers a different question: what did the caller experience? The toy W0 intentionally uses that wall boundary; `kbench` uses GPU time for its per-kernel figure. They should not be compared as if they were the same statistic.

Pitfall

A larger tile can leave the GPU less busy. In the recorded IQ4_XS gate/up sweep, eight output rows per SIMD group measured 324 microseconds, slower than 306 microseconds with four. The larger tile reduced the number of threadgroups, leaving too little independent work; two rows did not show a reliable improvement over four. The fix was to keep four and judge alternatives with repeated measurements. A separate documented attempt prefetched the next matrix with a no-barrier kernel; it was correct but 3–7% slower because the small caches could not hold the prefetched lines, so bytes were read twice. The sources are `ENGINE-MATH.md` §11.7, `README.md`, and the Q4_XS note in `gemma4_metal.m` near line 641. These are historical measurements under their recorded conditions, not new measurements from this chapter.

Exercises

Exercises

1. **Read.** For one Q4_0 block, write down the byte offsets of the fp16 scale and the 16 code bytes. Follow lanes 0–3 and state which 16 activation positions each lane uses.
2. **Measure.** Use the gate/up shape and quantised row-byte count from the model notes to compute the bytes streamed. Divide by the recorded kernel time and compare the result with the stated GB/s. Keep the notes' date and machine conditions attached.
3. **Calculate.** For the E4B attention-Q tensor in this chapter, derive the row bytes, threadgroup count, SIMD groups per threadgroup, and blocks visited by one lane. Explain why source-level load bytes exceed unique tensor bytes.
4. **Read.** Why does Q4_0 subtract eight times the activation sum after its nibble products? How does Q4_1 replace that correction?
5. **Read.** Trace the type and shape checks in the host dispatch. Name a combination that reaches the generic matvec and one that selects a specialized kernel.
6. **Predict.** Estimate operations per byte for a 32-token prefill tile using the chapter's IQ4_XS size. Compare with decode and explain why chapter 8 can reuse weights more effectively.

Rebuild checkpoint

Rebuild checkpoint

Milestone m7b: a Q8_0 matvec. Implement `steps/m7b_matvec.m` for one input vector and a matrix whose rows use Q8_0 blocks. Give each SIMD group output rows, let its lanes own block contributions, multiply signed 8-bit codes by the fp16 block scale and activation values, and reduce with `simd_sum`. Compare the output against both the m3 dequantiser plus a CPU dot product and the snapshot CPU matvec on a small fixed matrix. Check all rows, including a final partial tile, and report maximum absolute error. Keep the first version to one format and no fused normalization; add those only after the plain path agrees.

PREPRINT

Chapter 8

Prefill kernels: matrix × matrix

The question

How can one read each packed weight tile once, reuse it across many prompt rows, and keep the GPU's matrix units busy?

Reading map: 18 pages

The outline assigns 18 pages. The subsection budget below carries chapter 7's bandwidth floor into prefill and reserves room for a concrete E4B tile, source, experiment, and exercise space. Counts include the diagram, listings, and boxed material.

Subsection	Pages
Question and roofline handoff: bytes per useful operation	2
One output tile and the E4B tensor walk-through	3
Device load, block decode, and threadgroup tile staging	3
Barrier, simdgroup matrix loop, and output store	2
Threadgroup-memory, register, and occupancy arithmetic	2
Matvec versus matmul, dispatch threshold, and grid	2
Long-prompt chunking and padded activation buffers	1
Measured 1.65× comparison and W2 capture plan	1
Synthetic fp16 W0 comparison	1
Two documented pitfalls, exercises, and m7c checkpoint	1
Total	**18**

The idea

Chapter 7 ended at a bandwidth floor. Decode reads almost the full weight set for one activation row: reuse is low, so the bytes dominate. Prefill changes the question. A batch of prompt rows multiplies the same matrix, and the weight tile can be reused across many rows before it is replaced. The arithmetic grows with the number of rows; the weight traffic does not. That moves a sufficiently large prefill from the bandwidth side of the roofline toward the compute side.

How far does reuse move the roofline?

A useful way to carry chapter 7's bandwidth picture forward is to count work per packed weight byte. IQ4_XS stores 256 values in 136 bytes, so one stored weight represents just over half a byte. For one decode row that weight is used once: one multiply and one add, or about 0.266 bytes for each floating-point operation when metadata and activation traffic are ignored. That is still a low-intensity stream.

PREPRINT

For a 32-row prompt tile, the same stored weight contributes to 32 output dot products. It now supports 64 floating-point operations, while the weight bytes are fetched once into the tile. The ideal weight-only traffic is about 0.0083 byte per operation. At 256 rows the ratio falls to about 0.00104 byte per operation. Real traffic includes scales, activation reads, output stores, and tiles reloaded when the row or K range changes, but these additions do not erase the factor-of-T reuse.

Compare those ratios with an illustrative machine roofline: a 2.9 TFLOP/s compute peak and 100 GB/s of bandwidth meet at about 29 operations per byte. A single quantised matvec has only about 3.8 operations per weight byte, well below that crossover. At a 32-row batch the ideal weight-only ratio is about 120 operations per byte; at 256 rows it is about 960 operations per byte (two operations per weight; ENGINE-MATH §11.2.6 (line 1226)). The estimate explains the direction of travel: decode is constrained by weight delivery, while large prefill matrices can keep the arithmetic units busy. It does not promise that every short projection reaches the peak. Dispatch, attention, conversion, and edge tiles still take time. (The machine estimates are from ENGINE-MATH §5.8 and §11.1.)

The kernel's 2.2 TFLOP/s matmul estimate is about three quarters of the stated 2.9 TFLOP/s peak. That leaves room for the work the roofline calculation omits: quantised block unpacking, barriers between K slices, output conversion, matrix-fragment setup, and inactive simdgroups at edges. It also gives a sanity check for later measurements. A claimed result above the compute roof cannot be explained by this kernel's arithmetic alone; a much lower result may be a real consequence of shape, launch count, or memory pressure rather than a contradiction of the reuse argument.

For a matrix with K input columns and N output rows, one token row needs two floating-point operations for every input-output pair. With T rows, the same weights support about T times as much arithmetic. The E4B dense feed-forward projection has K=2,560 and N=10,240. With 32 rows it performs about 1.68 billion operations, counting each multiply and add separately. The packed IQ4_XS matrix uses about 0.53 bytes per value. The comparison is the point: the GPU can do much more than one operation for each byte fetched.

The engine does not expand the entire quantised matrix first. A threadgroup decodes only the next small K slice of a row tile into threadgroup memory. Its simdgroups multiply that shared slice by their activation tiles, accumulate in fp32, and move to the next K slice. The packed values are converted to fp16 for the matrix operation; the products accumulate into fp32 matrix fragments. The chapter 7 roofline still applies: it tells us why reuse matters. Now the central tuning problem is how large a useful tile can be held without exhausting registers or shared memory.

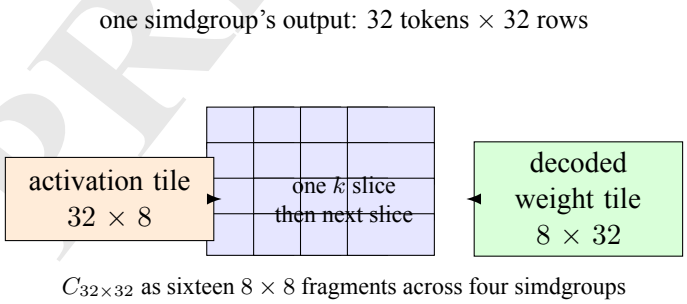


Figure 8.1: A 32-token by 32-output tile is accumulated in 8-by-8 fragments. At each K step, the activation fragment comes from device memory and the decoded weight fragment from threadgroup memory.

Why can prefill be compute-bound when decode is not?

In decode, one loaded weight usually contributes to one output dot product. In prefill, that weight contributes to each of a block of token rows. The matrix unit is designed for this second shape: it performs a small matrix product in one operation, rather than asking independent lanes to issue scalar products. As T increases, the weight bytes are shared across more arithmetic. Activation reads and output writes grow with T too, but for large weight matrices they are much smaller than repeatedly fetching the weights.

PREPRINT

The source's tile advances through K in chunks of 64. Each chunk contains two 32-value quantisation blocks per row. The kernel converts those chunks and reuses them while consuming up to 32 tokens and 32 output rows for each simdgroup tile. For a dense operation, several simdgroups work on separate token tiles while sharing the same row-block weight tile. For a small expert assignment, fewer token rows are available, so the dispatch assigns more row blocks to each threadgroup. The tile shape follows the work actually available.

Walk one E4B tile by hand

Take the feed-forward gate matrix in one layer: input width K=2,560 and output width N=10,240. With T=32 rows, one simdgroup tile covers those 32 input rows against 32 output rows. One threadgroup has four simdgroups. At the common dense setting TOK_SG=4, a 128-row call assigns each simdgroup a different 32-token slice while all four share the same 32 output rows. The threadgroup therefore covers 128 token rows by 32 output rows while loading one row block of weights per K step. For a 32-row call, only one simdgroup is active for computation; the others still participate in loading.

K=2,560 takes forty steps of 64. On the first step, each output row's first 64 values are brought from the packed weight row. Two calls to the 32-value decoder write the two halves into the threadgroup tile. The first eight K values are then paired with the first eight activation values in the first 8-by-8 multiply; the next eight values form another multiply, until all 64 values have contributed. The same decoded weights serve all four token tiles. The kernel then reuses the threadgroup buffer for K=64..127 and continues until K=2,559.

The 32-row block produces 32 output values for each token row. In the ordinary projection, each simdgroup holds sixteen 8-by-8 accumulators: four token fragments by four output fragments. Thus for 128 token rows, four simdgroups each compute a 32-by-32 result. The E4B FFN's gate and up matrices use the fused GLU mode: each 32-row block represents 16 gate rows and the matching 16 up rows. The kernel applies GELU to gate results, multiplies by the up results, and writes 16 hidden values per token. The down projection then consumes those 10,240 hidden values and emits the model width of 2,560.

A useful scale check: one 32-by-32 output tile over K=2,560 represents 2,621,440 multiply-add pairs, or 5,242,880 floating-point operations under the two-operations-per-pair convention. The values here come from the real tensor shape and tile dimensions, not a benchmark. This count describes arithmetic for one tile; it does not claim that the whole layer or model runs at that rate.

What is the tile walk?

The threadgroup has 128 threads, split into four 32-lane simdgroups. The host selects TOK_SG as 1, 2, or 4. The simdgroup index is divided into a token coordinate and a row-block coordinate. At TOK_SG=4, there is one row block and four token tiles. At TOK_SG=1, there are four row blocks and one token tile. Both arrangements use four simdgroups and keep the group busy, but they make a different reuse trade: many tokens reuse one weight tile, or one token tile consumes more weight rows.

The real code

How are 32 packed weights made usable?

The CPU implementation is the reference path. It decodes a batch of output rows once, rearranges them in K-major order, and reuses those rows as it walks activation batches. This makes its memory layout easy to compare with the GPU tile: both arrange weight values so nearby output rows can be consumed together, though the CPU uses vector accumulators rather than simdgroup matrix fragments. Read the decoder and transpose first; its scratch array is private to each worker.

```
705 static void matmul_tile(void *ctx, int tile, int tid) {
706     const MatmulJob *job = ctx;
707     const int K = job->K, N = job->N, n0 = tile * NB;
```

PREPRINT

```
708     const int nn = N - n0 < NB ? N - n0 : NB;
709     float *row = job->scratch[tid], *wt = row + K;
710 
711     for (int j = 0; j < NB; j++) {
712         if (j < nn) dequant_row(job->w, n0 + j, row);
713         else memset(row, 0, K * sizeof(float));
714         for (int k = 0; k < K; k++) wt[k * NB + j] = row[k];
715     }
716 
717     for (int t0 = 0; t0 < job->T; t0 += TB) {
718         const int nt = job->T - t0 < TB ? job->T - t0 : TB;
719         const float *x[TB];
720         for (int i = 0; i < TB; i++) x[i] = job->x + (size_t)(t0 + (i <
    nt ? i : 0)) * K;
721 
722         vec acc[TB][NB / VW] = {{0}};
723         for (int k = 0; k < K; k++) {
724             const vec *w = (const vec *)(wt + k * NB);
725             for (int i = 0; i < TB; i++) {
726                 const float xv = x[i][k];
727                 for (int j = 0; j < NB / VW; j++) acc[i][j] += xv * w[j
    ];
728             }
729         }
730         for (int i = 0; i < nt; i++)
731             for (int j = 0; j < nn; j++) {
732                 float *y = job->y + (size_t)(t0 + i) * N + n0 + j;
733                 *y = acc[i][j / VW][j % VW] + (job->accumulate ? *y : 0)
    ;
734             }
735     }
```

Listing 8.1: gemma4.c:705–735

The CPU tile makes the reuse explicit. ‘NB‘ rows are decoded and transposed before the activation loop, then a block of ‘TB‘ inputs walks K while vector accumulators hold the outputs. The GPU takes a shorter-lived tile and converts it to fp16 in threadgroup memory. Both avoid decoding the same stored row for every token in the batch.

Now compare that reference with the Metal decoder that feeds the matrix unit. For the block-format fast path, the helper calls the common decoder and converts each float to half. Its Q4_0 branch is visible in full: it reads the two-byte scale, unpacks low and high nibbles in file order, subtracts eight, and writes 32 half values. Other formats use the generic weight accessor. The final loop writes four groups of eight values into the threadgroup tile layout expected by matrix loads.

```
263 static void dequant32(device const uchar *row, int wtype, int k,
    threadgroup half *dst) {
264     half w[32];
265     if (is_blk(wtype)) {
266         float f[32];
267         dequant_blk32(row, wtype, k / 32, f);
268         for (int i = 0; i < 32; i++) w[i] = half(f[i]);
269     } else if (wtype == 2) {  // Q4_0: low nibbles are k .. k+15, high
    nibbles k+16 .. k+31
270         device const uchar *blk = row + (k / 32) * 18;
271         const half d = *(device const half *)blk;
272         device const ushort *qs = (device const ushort *)(blk + 2);
273         for (int i = 0; i < 8; i++) {
274             const ushort v = qs[i];  // bytes 2i, 2i + 1
```

```
275             w[2 * i] = d * half(int(v & 15) - 8);
276             w[2 * i + 1] = d * half(int((v >> 8) & 15) - 8);
277             w[16 + 2 * i] = d * half(int((v >> 4) & 15) - 8);
278             w[16 + 2 * i + 1] = d * half(int(v >> 12) - 8);
279         }
280     } else if (wtype == 8) {  // Q8_0
281         device const uchar *blk = row + (k / 32) * 34;
282         const half d = *(device const half *)blk;
283         device const char *q = (device const char *)(blk + 2);
284         for (int i = 0; i < 32; i++) w[i] = d * half(q[i]);
285     } else if (wtype == 1) {  // F16
286         for (int i = 0; i < 32; i++) w[i] = ((device const half *)row)[k
        + i];
287     } else {
288         for (int i = 0; i < 32; i++) w[i] = half(weight(row, wtype, k +
       i));
289     }
290     for (int g = 0; g < 4; g++) {
291         threadgroup half4 *o = (threadgroup half4 *)(dst + g * 64);
292         o[0] = half4(w[8 * g], w[8 * g + 1], w[8 * g + 2], w[8 * g + 3])
       ;
293         o[1] = half4(w[8 * g + 4], w[8 * g + 5], w[8 * g + 6], w[8 * g +
        7]);
294     }
295 }
```

Listing 8.2: `gemma4_metal.m`:263–295

For the E4B IQ4_XS weights, the block path calls `dequant_blk32`; this is where its nonuniform codebook and local scale are applied. The matrix kernel does not need to know the codebook layout. It receives half weights in the layout needed for the next multiply. That clean boundary is useful when reading the code: the block decoder answers "what is this value?", while the tile writer answers "where must this value sit for the matrix unit?"

Which work belongs to dequantisation?

The E4B file mixes formats, so the matrix kernel cannot assume that every packed byte means one simple nibble. IQ4_XS uses a 256-value super-block with a shared scale, eight local signed scales, and a nonuniform sixteen-value codebook. The 'dequant_blk32' helper handles the packed metadata and codebook lookup for one 32-value piece. The matrix tile writer treats its result as ordinary values after that point. Q4_K and Q5_K also use super-block metadata; their decoder expands their local scales and offsets in the same common helper. F16 and Q8_0 take simpler direct paths in 'dequant32'.

This boundary avoids making the simdgroup multiply-accumulate understand GGUF encodings. The matrix operation always consumes half values in an 8-by-8 layout. That means the quantised formats share the same matrix loop even though their block decoders differ. It also means the cost of an encoded weight has two parts: reading the compact bytes and performing the integer/codebook work that turns them into half values. The matmul estimate in ENGINE-MATH says dequantisation is small relative to the batched arithmetic at large row counts; for tiny batches, the conversion and pipeline startup can be a more visible part of the call.

The conversion does not write an expanded matrix back to device memory. It produces a short-lived threadgroup tile and overwrites it after use. This saves the extra allocation and the write/read traffic that a full half-precision copy would require. The trade is deliberate: each tile is decoded again on a later call or when another matrix operation needs it. For prefill, the important reuse happens before the tile is discarded: every activation fragment assigned to that row block consumes the same staged weights.

The input values are fp16 because that is the interface used by the model's producers and by the matrix unit path; accumulation is fp32. Converting dequantised weights from float to half introduces rounding in

PREPR

the weight tile, while fp32 accumulators keep the long K sum wider. The snapshot's math notes quantify fp16 input rounding and use output checks to validate engine paths, but this chapter does not claim bit identity with a different tile order or another backend. A matrix kernel can change accumulation order and therefore low bits even when every decoded value is correct.

How does the kernel stage and consume the tile?

The declaration below fixes the shared tile capacity, maps each simdgroup to token and row coordinates, and keeps the output fragments in matrix accumulators. The important point is that the weights are converted in bounded K slices rather than materialised as an N-by-K half matrix. Each thread writes one or more 32-value chunks, then all threads wait before any simdgroup reads the shared memory.

```
297 kernel void matmul(device const uchar *W [[buffer(0)]], device const
        uchar *W2 [[buffer(1)]],
298                    device const half *X [[buffer(2)]], device float *Y
        [[buffer(3)]],
299                    constant MatmulParams &p [[buffer(4)]], uint2 tg [[
        threadgroup_position_in_grid]],
300                    ushort tid [[thread_index_in_threadgroup]], ushort sg
         [[simdgroup_index_in_threadgroup]],
301                    ushort lane [[thread_index_in_simdgroup]]) {
302     const int ROW_SG = 4 / TOK_SG, cols = GLU ? 16 : 32;  // row blocks
        per threadgroup; outputs per block
303     threadgroup half wsh[4 * 32 * 64];  // ROW_SG blocks of 32 rows × 64
         k
304     const int ts = sg % TOK_SG, rs = sg / TOK_SG;  // this simdgroup's
        token tile and row block
305     const int t0 = (tg.x * TOK_SG + ts) * 32, n0 = (tg.y * ROW_SG + rs)
        * cols;
306     const bool active = t0 < p.T && n0 < p.N;  // idle simdgroups still
        help load the weights
307
308     simdgroup_float8x8 acc[4][4];
309     for (int i = 0; i < 4; i++)
310         for (int j = 0; j < 4; j++) acc[i][j] = simdgroup_float8x8(0);
311
312     device const half *xt = X + (ulong)t0 * p.K;
313     threadgroup const half *ws = wsh + rs * 2048;
314     for (int k0 = 0; k0 < p.K; k0 += 64) {
315         const int nk = min(64, p.K - k0);  // K is a multiple of 32
```

Listing 8.3: `gemma4_metal.m`:297–315

The loop distributes work over the 128 threads. A thread identifies its row, one of the two 32-value halves, and its output tile. It calls the decoder into a precise position in shared memory. After a barrier, each active simdgroup loads four activation fragments and four weight fragments and performs the 8-by-8 multiply-accumulates. A second barrier makes sure that every reader has finished before the next K step overwrites the shared tile.

Follow one row and one K slice through the code

Take output row 13 of the current 32-row block and the second 32-weight half of the current 64-wide K slice. In the loader loop, `r` is 13 and `kh` is 1. The destination expression places that row in the second eight-row band and its K values in the upper half of the 64-value slice. Its base within this row block is offset 808 half values: the second eight-row band contributes 512, the upper K half contributes 256, and the row's five-position offset contributes 40 (512 + 256 + 40 = 808). That is an address in the shared tile, not a file address.

The file address is selected first. `n` identifies the output row within the matrix, and `wrow` points to that row's packed bytes using the format-specific row stride. For a block format, `dequant32` selects the 32-value block at K offset 32, decodes its scale and codebook values, and writes the resulting halves into the destination slot. Four eight-value groups are separated by 64 halves so the next matrix load sees its expected eight-row by eight-K layout. The two address spaces are worth keeping distinct in your notes: the source pointer walks a packed GGUF row; `dst` walks a temporary tile arranged for the matrix unit.

On the consume side, `kk` advances from zero through seven. At each value it moves the activation load eight columns farther along K and selects the matching eight-K slice of the shared weights. For `kk = 0`, the first activation fragment covers token rows 0 through 7 and K positions 0 through 7; the first weight fragment covers output rows 0 through 7 and those same K positions. The multiply-accumulate adds their partial product to accumulator `acc[0][0]`. The other three activation fragments cover token rows 8 through 31, while the remaining weight fragments cover output rows 8 through 31. The nested loops update all sixteen pairs before the next `kk`.

After eight such steps, every K value in this 64-wide slice has contributed to all sixteen accumulators. The barrier after the loop is a reuse fence: it prevents one thread from replacing the shared bytes with the next slice while another simdgroup still reads the current one. The outer K loop then advances by 64 and repeats this sequence forty times for the E4B width of 2,560. The final store maps each lane's fragment coordinates back to its two values and skips rows outside T or N. This is the entire life of a tile: packed bytes, decoded shared halves, synchronized matrix fragments, fp32 partial sums, and guarded output stores.

What changes in the last token or row tile?

The threadgroup works in complete 32-token fragments, but a call can contain fewer rows. The activation structure is allocated with its maximum row capacity rounded to a multiple of 32. The kernel still checks the true call size before storing: each lane derives a token coordinate, then discards the result if that coordinate is past `p.T`. This lets the matrix unit issue a full fragment load while keeping padding out of the caller's output. A final 188-row chunk in a 256-row buffer therefore has enough allocated space for the fragment load, but only 188 output rows are committed.

Output rows have a similar edge. The row index used for a weight read is clamped to a valid row so a final partially filled row block never forms an invalid device pointer. The final store independently tests N and skips padded rows. These two guards protect different things: the input clamp makes the load address legal, while the output check prevents duplicated or padded computations from changing a real answer. The code comment says the unused rows are computed and never stored.

The fused gate/up path changes the meaning of a 32-row block. Its first 16 rows are gate weights; its next 16 are the matching up weights. Both matrices are decoded into the same shared tile, with the ‘up‘ flag selecting which device buffer to read. The accumulator array still has four output fragments, but the store consumes two gate fragments and the paired two up fragments. It clamps each side as configured, applies GELU to the gate value, multiplies by the up value, and writes the resulting 16 hidden coordinates in half precision. This combines two matrix products and the activation into one dispatch while respecting the two separate weight matrices.

The split matters when counting output rows: the host row block is 16 logical outputs in GLU mode, not 32. The weight-tile work includes both gate and up, so a single output row block has two sets of weights but only one set of final hidden values. That is why the host chooses a smaller ‘cols‘ value for GLU and why the grid covers twice as many row blocks for the same logical output width. Confusing packed gate/up rows with final hidden rows makes both the dispatch count and the operation count wrong by a factor of two.

```
316         for (int u = tid; u < ROW_SG * 64; u += 128) {  // unit u: row
        block u/64, row (u/2) % 32, k half u % 2
317             const int rb = u / 64, r = (u / 2) % 32, kh = u % 2;
318             if (kh * 32 >= nk) continue;
319             const bool up = GLU && r >= 16;
```

DRAFT

```
320             const int nb = (tg.y * ROW_SG + rb) * cols;
321             const int n = min(GLU ? nb + r % 16 : nb + r, p.N - 1);  //
    rows past N are computed, never stored
322             device const uchar *wrow = up ? W2 + (ulong)n * p.row_bytes2
     : W + (ulong)n * p.row_bytes;
323             dequant32(wrow, up ? WTYPE2 : WTYPE, k0 + 32 * kh, wsh + rb
    * 2048 + ((r / 8) * 8 + kh * 4) * 64 + (r % 8) * 8);
324         }
325         threadgroup_barrier(mem_flags::mem_threadgroup);
326         for (int kk = 0; kk < nk / 8 && active; kk++) {
327             simdgroup_half8x8 a[4], b[4];
328             for (int i = 0; i < 4; i++) simdgroup_load(a[i], xt + (ulong
    )i * 8 * p.K + k0 + kk * 8, p.K);
329             for (int j = 0; j < 4; j++) simdgroup_load(b[j], ws + (j * 8
     + kk) * 64, 8, ulong2(0, 0), true);
330             for (int i = 0; i < 4; i++)
331                 for (int j = 0; j < 4; j++)
    simdgroup_multiply_accumulate(acc[i][j], a[i], b[j], acc[i][j]);
332         }
333         threadgroup_barrier(mem_flags::mem_threadgroup);
```

Listing 8.4: `gemma4_metal.m`:316–333

Each row block is 32 rows by 64 K values, or 2,048 half values: 4 KiB. The source allocates four row blocks in `wsh[4*32*64]`, so the total is 16 KiB. At TOK_SG=4 only one row block is needed, and the other simdgroups spend their work on different token fragments. The fixed allocation simplifies indexing; at TOK_SG=4 only one row block is actively used, but the declaration still reserves all four.

What does an 8-by-8 operation own?

The source loads half-precision 8-by-8 simdgroup matrices and accumulates into corresponding float-precision 8-by-8 fragments. The multiply-accumulate call uses fp16 inputs and an fp32 accumulator. Each simdgroup owns a 32-token by 32-row result as a 4-by-4 array of 8-by-8 fragments. For each K slice it loads four activation fragments, four weight fragments, and updates all sixteen fragment products. That is why tile reuse is visible in the nesting: four token fragments reuse each weight fragment, and four output fragments reuse each activation fragment.

The store maps each lane's fragment coordinates back to two output values. In GLU mode it reads the paired gate and up accumulators, applies the selected GELU, and writes half output. In ordinary mode it stores fp32 values, with optional accumulation and clamping. Results stay in the simdgroup accumulators until the K loop ends; the code comment notes that reading the output back into matrix fragments for accumulation made the kernel about three times slower in the measured implementation. That is a source comment about this design, not a general rule for every GPU.

```
337     // Each lane writes the two elements of every 8×8 block it holds. (
    Reading y back into simdgroup matrices
338     // to accumulate instead makes the whole kernel ~3× slower, even
    when that code is never reached.)
339     const short2 fc = frag_coord(lane);
340     for (int i = 0; i < 4; i++)
341         for (int j = 0; j < (GLU ? 2 : 4); j++) {
342             const int t = t0 + 8 * i + fc.x, n = n0 + 8 * j + fc.y;  //
    N is a multiple of 8: n < N covers n + 1
343             if (t >= p.T || n >= p.N) continue;
344             float2 v = float2(acc[i][j].thread_elements()[0], acc[i][j].
    thread_elements()[1]);
345             if (GLU) {
```

PREPRINT

```
346                 const float2 u = clamp(float2(acc[i][j + 2].
    thread_elements()[0], acc[i][j + 2].thread_elements()[1]),
347                                        p.clamp.z, p.clamp.w);
348                 v = clamp(v, p.clamp.x, p.clamp.y);
349                 const float2 g = QUICK ? v / (1.0f + exp(-1.702f * v)) :
     float2(gelu(v.x), gelu(v.y));
350                 *(device half2 *)((device half *)Y + (ulong)t * p.N + n)
     = half2(g * u);
351             } else {
352                 device float2 *y = (device float2 *)(Y + (ulong)t * p.N
    + n);
353                 v = clamp(v, p.clamp.x, p.clamp.y);
354                 *y = p.accumulate ? *y + v : v;
355             }
356         }
357 }
```

Listing 8.5: `gemma4_metal.m`:337–357

What does this tile cost in fast memory?

The source declares `wsh[4 * 32 * 64]` as half values. There are 8,192 half values in that array, or 16,384 bytes (16 KiB) per threadgroup. One active row block occupies 32 rows times 64 K values times two bytes, or 4 KiB. The four row-block slots account for TOK_SG=1, where each of the four simdgroups owns a different row block. With TOK_SG=4 the address arithmetic uses a single row block and the simdgroups cooperate on four token tiles. The declaration still reserves the full 16 KiB in the pipeline.

That budget is fixed by the kernel, so it does not grow with the model's 2,560 input width: each K step overwrites the same shared buffer. If the whole decoded matrix were staged at once, one 32-row by 2,560-value block would already occupy 160 KiB as fp16. The bounded 4 KiB working row tile is forty times smaller, and the group reuses it over forty K steps. This is why the decoder is inside the K loop instead of being a preprocessing pass that expands the full weight matrix.

The other fast-memory consumer is the accumulator. A simdgroup owns sixteen 8-by-8 fp32 fragments. Each fragment contains 64 values, so the logical accumulator payload is 1,024 floats, or 4 KiB per simdgroup. Metal distributes the fragment values across its 32 lanes; the store reads two elements per lane from each fragment. That works out to 32 live fp32 accumulator values per lane, equivalent to 128 bytes of accumulator data per lane before counting operands, pointers, decoder temporaries, or loop state. It is a payload calculation, not a claim about a particular compiler's physical register allocation.

Four simdgroups in the threadgroup therefore have 16 KiB of logical accumulator payload in addition to the 16 KiB threadgroup-memory array. These are different resources: threadgroup memory is an explicit shared allocation, while accumulators are distributed per-thread state that compete with registers and can lower occupancy or spill. The source snapshot does not expose a per-pipeline register count or a hardware occupancy report, so it cannot establish how many such groups reside on an Apple GPU compute unit. The reasoning is still concrete: adding accumulator fragments increases per-lane live state; adding row blocks consumes shared memory; either can reduce the number of runnable groups or force slower storage. The documented 64-row experiment shows the register side of that trade in practice.

The source fixes the threadgroup at 128 threads, which matches four 32-lane simdgroups. It does not ask Metal for a platform maximum threadgroup-memory size or report occupancy. The safe conclusion is therefore limited to arithmetic from the declared tile: 16 KiB allocated per group and 32 fp32 accumulator values logically owned per lane. Exact residency depends on the GPU family, the compiled pipeline, and other resources. A reader reproducing m7c should measure or query those properties on the target machine instead of treating this kernel's counts as a universal Apple silicon limit.

PREPRINT

Why not make every tile wider?

More output rows can reduce repeated activation reads, but each extra 8-by-8 accumulator occupies registers in every lane. The kernel's sixteen fragments already represent a substantial live state, alongside loaded fragments, pointers, and conversion temporaries. Enlarging the row tile can lower occupancy or spill accumulators from fast registers to slower memory. The total work then falls even though each conceptual tile has better reuse.

This explains why tile size is a balance, not a monotonic "bigger is better" knob. The existing tile keeps the matrix operation large enough for reuse while keeping the accumulator grid bounded. K is streamed in 64-value pieces so the threadgroup memory is reused instead of scaling with the full input width. Dense prompt calls can use more token simdgroups; routed experts with few rows can use taller row tiles to avoid idle matrix units.

How does C select the matrix kernel?

The host wrapper first rejects unsupported shapes: K must fit the format's block size, N aligns to eight (or sixteen for fused GLU), and special types have their own constraints. For T greater than one it selects the matrix kernel. The token tile count is chosen from T: one simdgroup through 32 rows, two through 64, and four above that. Decode stays on the matrix-vector paths because a 32-token matrix tile would waste most of its work for one row.

```
1930 static int matmul(void *y, const void *x, const float *norm_w, float eps
       , const void *w, size_t row_bytes, int wtype,
1931                     const void *w2, size_t row_bytes2, int wtype2, int T,
       int K, int N, int accumulate, int quick,
1932                     const float *clamp) {
1933      const int blocks = block_type(wtype) || (w2 && block_type(wtype2));
1934      if (K % 8 || ((blocks || T == 1) && K % 32) || N % 8 || !known_type(
       wtype) || (w2 && (N % 16 || !known_type(wtype2))) ||
1935          ((super_type(wtype) || (w2 && super_type(wtype2))) && K % 256)
       || (norm_w && T != 1) || (quick && T == 1) ||
1936          (clamp && T == 1 && (w2 || wtype == 2 || wtype == 3 || wtype ==
       12 || wtype == 20 || wtype == 21 || wtype == 23)))
1937          return 0;  // (decode clamps only in the generic matvec, without
        glu)
1938      const struct { int T, K, N; uint32_t row_bytes, row_bytes2; int
       accumulate; float eps, pad, clamp[4]; } p = {  // (clamp: a float4,
       16-byte aligned)
1939          T, K, N, (uint32_t)row_bytes, (uint32_t)row_bytes2, accumulate,
       eps, 0,
1940          {clamp ? clamp[0] : -INFINITY, clamp ? clamp[1] : INFINITY,
       clamp ? clamp[2] : -INFINITY, clamp ? clamp[3] : INFINITY}};
1941      // tiling: simdgroups along the tokens; few tokens (an expert's) get
        tall tiles
1942      const int glu = w2 != NULL, tok_sg = T == 1 ? 0 : T <= 32 ? 1 : T <=
        64 ? 2 : 4;
1943      // decode: kernels for Q4_0 and the experts' IQ3_S / IQ4_NL / IQ4_XS
        (both matrices of one type in glu mode)
1944      const int same = !glu || wtype2 == wtype;
1945      const int iq4xs_aligned = (uintptr_t)w % 4 == 0 && (!glu || (
       uintptr_t)w2 % 4 == 0) && row_bytes % 4 == 0 &&
1946                                (!glu || row_bytes2 % 4 == 0);  // the
       kernel reads its nibbles as uints
1947      const int k = T > 1                     ? K_MATMUL
1948                    : (wtype == 2 || wtype == 3) && same ? K_MATVEC_Q4
1949                    : wtype == 12 && same    ? K_MATVEC_Q4K
```

```
1950                    : wtype == 21 && same && K <= 3072 ? K_MATVEC_IQ3S  // 32 · 32 · IQ3S_J
1951                    : (wtype == 20 || (wtype == 23 && iq4xs_aligned)) && same ? K_MATVEC_IQ4NL
1952                                                    : K_MATVEC;
1953     id<MTLComputePipelineState> ps = variant(k, wtype, glu ? wtype2 : wtype, glu, norm_w != NULL, tok_sg, 0, quick, 0, 0);
```

Listing 8.6: gemma4_metal.m:1930–1953

The final grid covers ceil(T/(32·TOK_SG)) token tiles and enough output row blocks to cover N. For T=128, an ordinary down projection with N=2,560 uses TOK_SG=4 and 32 output rows per row block: one token-grid position by 80 row-grid positions, or 80 threadgroups. In the fused gate/up call, each row block emits 16 hidden values because it carries 16 gate and 16 up rows; the same shape therefore uses 640 row-grid positions, or 640 threadgroups. Each group has four simdgroups, so these counts include the whole 128-thread group. These are grid dimensions calculated from the source rules, not measured performance. For a matrix operation, each threadgroup is 128 threads. The activation allocation rounds T up to a multiple of 32 because simdgroup matrix loads read complete 32-row groups. The code excerpt shows the allocation and the fp16 buffers that feed the kernel. For the E4B model width, the activation staging is proportional to padded T times 2,560; the FFN hidden activation is padded T times 10,240. Padding a 256-row chat call does not add rows; it already aligns.

```
1098 static Acts acts_new(const Model *m, int T) {
1099     T = (T + 31) / 32 * 32;
1100     const size_t f = sizeof(float), E = m->n_embd, nq = (size_t)m->n_head * m->hd_max, U = m->n_expert_used;
1101     Acts a = {.T_max = T};
1102     a.tokens = shared_alloc((size_t)T * sizeof(int));
1103     a.rows = shared_alloc((size_t)T * sizeof(int));
1104     a.x = shared_alloc(T * E * f);
1105     a.xn = shared_alloc((T + 32) * E * sizeof(f16));  // (the LM head reads rows from `first` on, in blocks of 32)
1106     if (m->n_expert) a.xn2 = shared_alloc(T * E * sizeof(f16));  // (the routed experts' input)
1107     a.q = shared_alloc(T * nq * f);
1108     a.kv = shared_alloc(T * 2 * m->nkv_max * f);
1109     a.qr = shared_alloc(T * nq * sizeof(f16));
1110     a.o = shared_alloc(T * nq * sizeof(f16));
1111     a.a = shared_alloc(T * E * f);
1112     a.b = shared_alloc(T * E * f);
```

Listing 8.7: gemma4.c:1098–1112

The same C wrapper is used by attention projections, feed-forward projections, the tied output head, and encoder matrix operations. Fused GLU supplies two weight buffers and computes gate and up together. Clamp parameters travel in the constant buffer and the kernel clamps outputs before writing. For T=1 the host may fuse normalization into matvec; that is a separate decode optimization and does not change prefill's matrix tile.

The C call site shows the sequence around that dispatch. It first offers a fused-normalization matvec when a deferred single-row norm is present; otherwise it materializes the input and tries the Metal matrix wrapper. If Metal rejects the shape or cannot encode it, C synchronizes and runs the CPU reference. The token count is passed unchanged, so the Metal host can choose between the one-row and batched kernels.

```
776 static void matmul(float *y, const f16 *x, const Tensor *w, int T, int accumulate) {
777     const int K = (int)w->ne[0], N = (int)w->ne[1];
778     if (x == deferred_norm.out && GPU(metal_matvec_norm(y, deferred_norm.x, deferred_norm.w, deferred_norm.eps, w->data,
```

```
779                                                          row_bytes(w), w
    ->type, K, N, accumulate)))
780         return;
781     materialize_norm(x);
782     if (GPU(metal_matmul(y, x, w->data, row_bytes(w), w->type, T, K, N,
    accumulate))) return;
783     gpu_sync();
784     matmul_f32(y, to_float(x, (size_t)T * K), w, T, accumulate);
785 }
```

Listing 8.8: gemma4.c:776–785

Matvec or matmul: what changes?

The comparison below uses an IQ4_XS weight as a common byte-count example. Its block stores 136 bytes for 256 values, or 0.53125 byte per value. The FLOP-per-byte column counts only weight traffic: each use contributes a multiply and add, and ignores scale metadata reloads, activations, outputs, and cache effects. For a 32-row matmul tile, a weight supports 64 operations. These are arithmetic-intensity estimates, not measured bandwidth.

Question	Matvec (T=1)	Matmul (T=32)	What that means
Weight bytes per FLOP	About 0.266 B/FLOP	About 0.0083 B/FLOP	Matmul reuses each weight across 32 rows
Threads	128 per group; four simdgroups; generic path covers 16 rows	128 per group; four simdgroups hold token/row tiles	Same group width, different work per lane
Explicit thread-group memory	Generic path declares 16 B for NORM; IQ4 lookup adds a 1 KiB table	16 KiB declared; one active 32-by-64 weight block is 4 KiB	Matvec streams packed rows; matmul stages decoded halves
Accumulator form	Per-lane scalar partial sums, then SIMD reduction	Sixteen 8-by-8 fp32 fragments per simdgroup	Dot-product reduction versus matrix reuse
Best fit	One token; weight traffic dominates	Multiple prompt rows; matrix work amortizes each weight load	Small T can still be dispatch/latency limited

The 16-byte NORM scratch comes from four fp32 partial sums in the generic kernel. The IQ4 specialization's lookup array contains 256 half2 values, or 1,024 bytes, in addition to its reduction scratch. Other matvec variants have their own layouts, so the table describes these named paths rather than every specialization. Chapter 7 covers row ownership and reductions; this comparison isolates the design change that makes a batch worth sending through matrix fragments.

Where is the threshold?

For a valid matrix shape, the source's threshold is direct: T=1 selects a matvec variant; every T greater than one selects `K_MATMUL`. Within matmul, token groups change at two additional boundaries. Up to 32 rows sets TOK_SG to 1; 33 through 64 sets it to 2; above 64 sets it to 4. These constants divide the four simdgroups between token tiles and output row blocks. The thresholds are source decisions, not values inferred from a benchmark.

```
1941     // tiling: simdgroups along the tokens; few tokens (an expert's) get
         tall tiles
1942     const int glu = w2 != NULL, tok_sg = T == 1 ? 0 : T <= 32 ? 1 : T <=
         64 ? 2 : 4;
1943     // decode: kernels for Q4_0 and the experts' IQ3_S / IQ4_NL / IQ4_XS
         (both matrices of one type in glu mode)
```

PREPRINT

```
1944     const int same = !glu || wtype2 == wtype;
1945     const int iq4xs_aligned = (uintptr_t)w % 4 == 0 && (!glu || (
         uintptr_t)w2 % 4 == 0) && row_bytes % 4 == 0 &&
1946                               (!glu || row_bytes2 % 4 == 0);  // the
         kernel reads its nibbles as uints
1947     const int k = T > 1                      ? K_MATMUL
1948                   : (wtype == 2 || wtype == 3) && same ? K_MATVEC_Q4
1949                   : wtype == 12 && same    ? K_MATVEC_Q4K
1950                   : wtype == 21 && same && K <= 3072 ? K_MATVEC_IQ3S  //
          32 · 32 · IQ3S_J
1951                   : (wtype == 20 || (wtype == 23 && iq4xs_aligned)) &&
         same ? K_MATVEC_IQ4NL
1952                                            : K_MATVEC;
1953     id<MTLComputePipelineState> ps = variant(k, wtype, glu ? wtype2 :
         wtype, glu, norm_w != NULL, tok_sg, 0, quick, 0, 0);
1954     if (profiling)
1955         snprintf(prof_key, sizeof prof_key, "T%-4d K%-5d N%-6d %s%s", T
         > 1 && T <= 64 ? (T + 31) / 32 * 32 : T, K, N,
1956                  wtype == 2 ? "q4" : wtype == 3 ? "q4_1" : wtype == 12 ?
          "q4k" : wtype == 8 ? "q8" : wtype == 20 ? "iq4nl" :
1957                  wtype == 21 ? "iq3s" : wtype == 23 ? "iq4xs" : wtype ==
          30 ? "bf16" : wtype == 1 ? "f16" : "f",
1958                  w2 ? " glu" : "");
1959     const size_t lens[5] = {(size_t)N * row_bytes, glu ? (size_t)N *
         row_bytes2 : 1, 1, 1, 1};
1960     const void *ptrs[5] = {w, glu ? w2 : w, x, y, norm_w ? (const void
         *)norm_w : x};
1961     if (T == 1) {
1962         // 4 simdgroups × IQ3S_RSG, IQ4NL_R, Q4K_R, MV_R
1963         const int rows_tg = k == K_MATVEC_IQ3S ? 32 : k ==
         K_MATVEC_IQ4NL ? 16 : k == K_MATVEC_Q4K ? 8 : 16;
1964         return encode(ps, k, ptrs, lens, 5, &p, sizeof p, size2((N +
         rows_tg - 1) / rows_tg, 1), size2(128, 1));
1965     }
1966     // token tiles on the fast grid axis: the tiles that share a weight
         tile run together, so each weight is
1967     // read from DRAM about once (the activations are small and stay in
         the cache)
1968     const int rows_tg = (glu ? 16 : 32) * (4 / tok_sg);
1969     return encode(ps, k, ptrs, lens, 4, &p, sizeof p, size2((T + 32 *
         tok_sg - 1) / (32 * tok_sg), (N + rows_tg - 1) / rows_tg),
1970                   size2(128, 1));
```

Listing 8.9: `gemma4_metal.m`:1941–1970

At T=32 one simdgroup works on one token tile and the other three can process row blocks. At T=64 the group splits between two token tiles and two row blocks. At T=128 all four simdgroups are token tiles sharing one row block. In ordinary mode a row block covers 32 outputs; with fused GLU it covers 16 logical outputs because each block holds paired gate/up rows. This gives enough independent work without increasing the 128-thread group size.

Why does this route fit E4B prompts?

Chat feeds at most 256 tokens to a call; the benchmark can use 512. The caller may process a longer prompt as several calls, so the memory footprint is capped while every call still has enough rows to reuse weight tiles. The C buffer constructor rounds the maximum call length up to 32. The GPU reads activations in those blocks, and the kernel guards stores for the final partial block. The rounding is a storage and load-safety rule, not extra user-visible tokens.

A long request is split by the caller, not by the kernel's K loop. The K loop divides one matrix multiply into weight slices; prompt chunking divides a token sequence into separate forward calls. Those are two independent loops, and confusing them leads to wrong estimates of both memory and latency. In `prefill_chunks`, the next call takes the smaller of remaining prompt rows and `T_max`; the input tokens are copied into the reusable activation structure, forward runs on that batch, and the sequence index advances. The logits are requested for the final row of each call, while the caller ultimately retains the prompt's last-row logits.

```
3925 // Runs n tokens in calls of up to a->T_max tokens; the logits are the
       last one's.
3926 static void prefill_chunks(const Model *m, Acts *a, State *st, const int
        *tok, int n, float *logits) {
3927     for (int i = 0; i < n;) {
3928         const int T = n - i < a->T_max ? n - i : a->T_max;
3929         gpu_sync();  // (the GPU reads a->tokens)
3930         memcpy(a->tokens, tok + i, T * sizeof(int));
3931         forward(m, a, st, T, T - 1, 1, logits);
3932         i += T;
3933     }
3934 }
```

Listing 8.10: `gemma4.c:3925–3934`

A concrete 700-token text prompt with a 256-row chat capacity becomes calls of 256, 256, and 188 rows. The engine reuses the activation buffers for each call rather than allocating a 700-row set. The model state and KV cache carry positions forward, so chunk two begins after chunk one's positions; chunk three follows it. The GPU is synchronized before the CPU rewrites the token staging buffer because the preceding command may still be reading it. That explicit wait is a correctness constraint, and a possible cost for very small chunks.

For the benchmark path, C sets its capacity to at least 512 rows and prefill uses calls of up to that size. Chat uses a maximum of 256 tokens per call unless an image must be run whole. A smaller call reduces the live activation footprint; once the batch is already large enough to saturate the matrix work, making it bigger can consume memory without improving per-token speed. This is the practical reason the production chat capacity and benchmark capacity differ.

The C and Metal layers divide responsibility cleanly. C owns sizes, data buffers, tensor formats, and dispatch dimensions. The Metal function constants specialize each pipeline for weight type, GLU mode, and token tile count. At runtime a single kernel does not repeatedly branch over every supported quantisation format; the selected variant is compiled with those choices fixed.

Where does the 1.65× prefill advantage come from?

The README's historical comparison used the same E4B Q4_0 model on an M3 MacBook Air with an 8-core GPU. Two alternating rounds reported llama.cpp at 269.8 and 263.6 tokens/s and this engine at 439.3 and 440.3 tokens/s for 512-token prefill: the table reports 1.65×. This is an end-to-end prefill result. It includes attention, dispatch, and the engine's other work, so it cannot be credited to the matrix kernel alone.

The engine-math estimate puts the matmuls near 2.2 TFLOP/s against about 2.9 TFLOP/s theoretical peak. That is a modelled/estimated rate, not a measurement from the current writing session; Chapter 21 covers how to re-measure it. The kernel design makes compute-heavy matrix work efficient by sharing decoded weights across token rows and using matrix operations. The larger documented prefill change came from skipping unneeded rows after layer 23: the README records a move from 260 to 440 tokens/s with identical output under its check. The 1.65× comparison is the combined engine result against llama.cpp. Chapter 21 re-measures speed with the quiet-machine gate and interleaved runs, since historical measurements on this shared Mac moved substantially with load.

PREPRINT

Experiment W2

On a quiet GPU, run the outline's prompt benchmark and then its CPU contrast: `gemma4 -m $MODEL -f text.txt --bench 512 16 -r 3` then add `-t -ngl 0` for the CPU path. Record the prefill tokens/s, the exact command, date, machine, idle-gate reading, and whether each number is wall or GPU time. The README's older 439–440 tokens/s result is historical context, not the expected value of this run. **Try it on your machine (W2):** compare prefill tokens/s for the GPU and CPU commands. The GPU path is expected to be faster; the README's 439–440 tokens/s is a dated project measurement from 2026-10-03/04 (README, "Results"), not a fresh expected value. Do not infer the matrix kernel's throughput from end-to-end tokens/s; use `tools/mmbench` only for the optional encoder-file comparison.

Experiment W0

Run the saved synthetic experiment with no model or engine process: `nice -n 10 python3 code/expected/ch08-w0-matmul.py`. The script stores deterministic inputs through Python's IEEE fp16 pack/unpack format, computes the same 32-by-64 times 64-by-32 product with scalar and 8-by-8-by-16 tiled loops, checks every output, and reports median wall time over five runs. The captured run is saved in `code/expected/ch08-w0-matmul.txt`; rerun with `> code/expected/ch08-w0-matmul.txt` to replace it. On 2026-10-22, macOS arm64 with Python 3.9.6 and load averages 4.89/4.59/4.51, the checksum was 0.085937500 and the maximum output difference was zero. The five-run median was 8.519 ms for the scalar loop and 11.000 ms for the tiled loop. The GPU idle gate is not applicable because the run is CPU-only. These toy-only times include Python interpreter and loop overhead; they say nothing about Metal, the GPU, or the engine's measured prefill rate.

Pitfall

A wider tile intended to halve activation loads reduced throughput sharply: the README records 260 tokens/s before and 44 after, because the accumulators spilled out of registers. The intended reuse was real, but the larger live accumulator set cost more than it saved. Fix: inspect compiler resource and spill information, then compare a bounded tile under the same measurement conditions. The 44 figure is the documented historical result, not a new run. (Source: snapshot README, "Things that did not work," entry "A wider matmul tile".)

Pitfall

The output-store comment records another costly experiment: reading Y back into simdgroup matrix fragments for accumulation made this kernel about three times slower. The cause was an extra matrix load and conversion on the result path; the fix in the current kernel is to keep the running sums in the fp32 fragments and write once after the K loop. This is a source-documented measurement, not a general statement about every output accumulation strategy. (Source: snapshot `gemma4_metal.m:337--338`.)

Exercises

1. Count the floating-point operations for a product with 512 rows, width 2,560, and 10,240 output rows. At 2.2 TFLOP/s, estimate its ideal arithmetic time. Hint: count each multiply and add separately.
2. A decoded weight tile serves four token fragments. Explain which data is reused and which accumulators grow when the output row tile is widened.
3. Trace the dispatch for T=48 and T=128. What is TOK_SG in each case, and how many token rows can one threadgroup cover?
4. Change the synthetic W0 script to multiply 16-by-16 tiles. Compare outputs after changing loop order; label any elapsed times toy-only.
5. The README lists 439.3 and 440.3 tokens/s for this engine. Name two other components in a prefill call that prevent assigning the entire advantage to matmul.
6. Use the table's IQ4_XS byte count to estimate weight-only bytes per FLOP for 128 token rows. State what traffic the estimate leaves out.

Rebuild checkpoint

Checkpoint m7c. Build `steps/m7c_matmul.m`: a small fp16-input, fp32-accumulate 8-by-8 matrix kernel. Compare its full output against a CPU reference on fixed small values, including non-square edges padded to eight. Then extend the loop to consume successive K tiles and verify after each extension. The checkpoint teaches the matrix mapping; it does not reproduce the E4B's quantised decoder or its speed claim.

PREPRINT

Chapter 9

Small steps and fusing them

Reading map: 10 pages

Subsection	Pages
The question and the latency budget	1
One token's chain of small steps	1
RMS norm as a four-number calculation	1
Residual, post-norm, and next-input fusion	2
Fusing the FFN tail and activation work	1
RoPE, softcap, casts, clamps, and argmax	1
Encoder groups, command buffers, and token overlap	1
W0 launch experiment, W2 profile, and documented limits	1
Exercises, Part II recap, and m7c checkpoint	1
Total	**10**

The question

Why do the little kernels matter, and how do we remove steps without changing the answer? A token's large matrix operations read gigabytes, but several hundred smaller GPU steps still sit between those operations and each one has to finish before its dependent successor can begin.

The idea

The matrix kernels from chapters 7 and 8 do most of the arithmetic and move most of the bytes. Around them are reductions and simple elementwise work: normalise a row, rotate query coordinates, add a residual, apply an activation, scale or clamp values, and select the largest logit. It is tempting to call these operations "free" because each touches relatively few numbers. On a GPU, a tiny kernel can cost more to start and drain than to do its arithmetic.

The latency budget is made of steps

One decode token passes through 42 layers. In the engine's accounting, a layer has about 12 dependent GPU steps. The Q, K, and V projections count as one step because they are submitted as a concurrent group; the later attention, output projection, residual and feed-forward work form the remaining sequence. Twelve times 42 is about 500 dependent steps per generated token.

The phrase "dependent" matters. The next step cannot use an output that does not exist yet. A threadgroup may finish early while the last threadgroups are still draining; the following dispatch then needs to ramp up. The measured dispatch floor in the documented kbench session was about 8.5 microseconds. Multiplying 500 by 10 microseconds gives 5 milliseconds of possible step overhead. That is an estimate

PREPRINT

to orient us, not a prediction that every device or every step costs exactly 10 microseconds. The 8.5 microseconds and roughly 500-step count are recorded in ENGINE-MATH §11.7 and §5.14.

At the same time, we should not count every source-level operation as a separate launch. The engine has already merged many operations into kernels that must run for another reason. The useful question is: which values are already in registers or on chip when the extra arithmetic is needed, and can the kernel produce the next consumer's input without writing and reading an intermediate array?

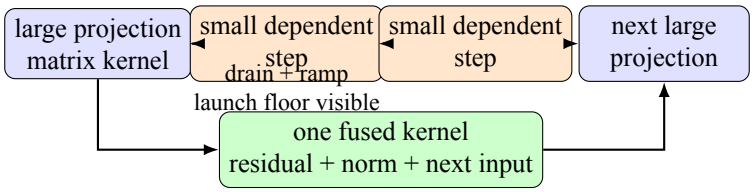


Figure 9.1: A dependent chain pays a launch and a drain between steps. Fusion is useful when the combined kernel can consume a value it already has and directly prepare the next operation's input.

What does a token's chain look like?

The normal path through a layer is not a pile of arbitrary vector kernels. Each step has a clear owner for its data. The attention projections produce query, key, and value; attention preparation normalises and rotates query/key and stores key/value; attention reads the cache; the output projection produces a residual branch. Then `add_norm` adds that branch and prepares the FFN input. The fused gate/up operation applies GELU and multiplies the gate by the up branch while writing the hidden vector. The down projection produces the FFN branch. `ffn_out` applies the tail's norm and residual, and can write the next layer's normalised input. The per-layer embedding block, when present, adds its own gate, projection, and residual steps.

The GPU is especially sensitive to steps whose result feeds the next operation immediately. During decode, there is one token row, so a small kernel has little independent work with which to hide its startup. During prefill, a kernel may process many rows; its dispatch cost is spread across more useful arithmetic. That is why the chapters on matvec and matmul focused on the large kernels, while this chapter concentrates on the seams between them.

RMS norm with four values

RMS normalisation measures the typical magnitude of a vector, then scales each coordinate by that shared amount and by its learned coordinate weight. For four values, the engine's CPU reference first finds the mean square and adds a small epsilon before taking the reciprocal square root:

$$s = 1/\sqrt{(x_0^2 + x_1^2 + x_2^2 + x_3^2)/4 + \epsilon}$$

Take x = [1, -1, 2, 0]. Its mean square is 1.5, so s is about 0.8165. Applying weights [1, 0.5, 1, 2] gives [0.8165, -0.4082, 1.6330, 0]. The final zero remains zero because scaling cannot make it nonzero.

The CPU reference at `gemma4.c:789--798` computes one scale and then writes the weighted row. The `add_norm` examples below use this helper twice: once for the branch being added and once for the updated residual when a next-input buffer is requested.

The GPU's standalone `rmsnorm` assigns one simdgroup to a row and reduces partial sums across its 32 lanes. For a single decode row that would make one simdgroup perform several passes over the vector. The fused `add_norm` instead assigns the row to a 256-thread threadgroup, so the reduction covers the row with more workers. The kernel keeps each thread's small values in a local array, reduces their squares, and reuses those values for the residual and output. The row width is at most 4096 for this kernel, so each thread needs at most 16 entries.

PREPRINT

Can the residual step prepare the next input?

After attention, the main stream `x` holds the old residual state and `a` holds the attention output. The row update is: normalise `a`, scale it by the learned post-attention weights, add it into `x`, and apply the layer output scale. In a separate design, the next FFN operation would now launch another kernel to normalise this updated `x` and multiply by the FFN input weights. But `add_norm` already owns the complete updated row and is about to write it. It can calculate that next norm while the row is local, then write the half-precision FFN input as well.

The CPU twin shows the sequence in a form we can follow one coordinate at a time. The first normalisation belongs to the branch being added. The second is conditional: when an output buffer is provided, it normalises the updated residual with `w2` and writes the next step's fp16 input.

```
1381 static void add_norm_row(void *ctx, int t, int tid) {
1382     (void)tid;
1383     const AddNormJob *job = ctx;
1384     float *x = job->x + (size_t)t * job->n;
1385     const float *a = job->a + (size_t)t * job->n, scale = rms_scale(a,
         job->n, job->eps);
1386     for (int i = 0; i < job->n; i++) x[i] = (x[i] + a[i] * scale * job->
         w[i]) * job->out_scale;
1387     if (!job->out) return;
1388     const float s2 = rms_scale(x, job->n, job->eps);
1389     for (int i = 0; i < job->n; i++) job->out[(size_t)t * job->n + i] =
         float_to_half(x[i] * s2 * job->w2[i]);
1390 }
1391
1392 // x = (x + rmsnorm(a) · w) · out_scale per row; with out (else NULL),
       also out = rmsnorm(x) · w2, the normed input
1393 // of the next step, so that it needs no step of its own
1394 static void add_norm(float *x, const float *a, const float *w, int rows,
         int n, float eps, float out_scale, f16 *out,
1395                      const float *w2) {
1396     if (out && out == deferred_norm.out) deferred_norm.out = NULL;
1397     AddNormJob job = {x, a, w, n, eps, out_scale, out, w2};
1398     STEP(metal_add_norm(x, a, w, rows, n, eps, out_scale, out, w2), rows
         , add_norm_row, &job);
1399 }
```

Listing 9.1: `gemma4.c:1381–1399`

The Metal implementation follows the same order, with a threadgroup reduction between the two normalisations. Notice that the second output is optional. The same kernel can update the residual alone, or also produce the next consumer's input. This distinction lets callers avoid calculating a value they do not need.

```
765 // row tg, 256 threads (n <= 4096): x = (x + rmsnorm(a) · w) · out_scale
       , and (has_out) out = rmsnorm(x) · w2 in fp16,
766 // the next step's normed input. A whole threadgroup per row: in decode
       there is one row, and a simdgroup alone
767 // would take several times as long.
768 struct AddNormParams { int n; float eps; float out_scale; int has_out;
       };
769
770 kernel void add_norm(device float *x [[buffer(0)]], device const float *
       a [[buffer(1)]],
771                      device const float *w [[buffer(2)]], device half *
       out [[buffer(3)]],
772                      device const float *w2 [[buffer(4)]], constant
       AddNormParams &p [[buffer(5)]],
```

```
                       uint row [[threadgroup_position_in_grid]], uint tid
    [[thread_index_in_threadgroup]],
                       uint nt [[threads_per_threadgroup]]) {
    threadgroup float tmp[32];
    const int m = (p.n + nt - 1) / nt;  // (at most 16)
    device const float *ar = a + (ulong)row * p.n;
    device float *xr = x + (ulong)row * p.n;
    float v[16], ss = 0;
    for (int j = 0; j < m; j++) {
        const int i = tid + nt * j;
        v[j] = i < p.n ? ar[i] : 0.0f;
        ss += v[j] * v[j];
    }
    const float s = rsqrt(block_sum(ss, tmp, tid, nt) / p.n + p.eps);
    ss = 0;
    for (int j = 0; j < m; j++) {
        const int i = tid + nt * j;
        if (i >= p.n) continue;
        v[j] = (xr[i] + v[j] * s * w[i]) * p.out_scale;
        xr[i] = v[j];
        ss += v[j] * v[j];
    }
    if (!p.has_out) return;
    const float s2 = rsqrt(block_sum(ss, tmp, tid, nt) / p.n + p.eps);
    for (int j = 0; j < m; j++) {
        const int i = tid + nt * j;
        if (i < p.n) out[(ulong)row * p.n + i] = half(v[j] * s2 * w2[i])
    ;
    }
}
```

Listing 9.2: `gemma4_metal.m`:765–800

The name `add_norm` is short, but the source comment describes an important scheduling rule: "the step before computes the next step's input." The next layer's pre-norm is not a free-standing stage in the normal decode chain; the preceding layer tail creates its fp16 input as part of its own work. The initial embedding still needs an initial RMS norm, because there is no earlier layer tail to do that work.

The host call site in gemma4.c, lines 2109–2116, makes the producer-consumer relationship visible. Attention produces the branch; the residual helper combines it with the main stream and writes the FFN's normalised input. The FFN consumes that buffer, and its tail can prepare the following layer. Following the buffers shows which function owns each input.

```
        attention(m, L, l, a, st, Tl, pos);
        add_norm(a->x, a->a, L->post_attn_norm, Tl, E, m->eps, 1.0f, a->
    xn, L->ffn_norm);
        ffn(m, L, a, Tl, next_norm);
        if (m->n_pl) {
            if (l == 0) pl_inputs(m, a, Tl, prev);  // (the GPU runs
    layer 0 meanwhile)
            pl_block(m, L, l, a, Tl, next_norm);
        }
        gpu_submit();  // start the GPU on this layer while the next is
    encoded
```

Listing 9.3: `gemma4.c`:2109–2116

Fusion does not mean "combine everything." The residual update and its output normalisation share a row and a reduction, so they are natural neighbours. The next big matrix operation is still a separate

kernel because it has a different parallel shape and needs to stream a large weight matrix. The goal is to remove an intermediate dispatch and memory trip where the producer already has the needed values.

The FFN tail and activation work

The feed-forward block's tail is another good place to combine steps. In a dense layer, the down projection has produced `b`; the tail normalises that branch, applies post-FFN weights, adds it to the residual, and applies the layer output scale. In a routed layer, the output also combines selected expert rows. Then the kernel can write either a half-precision copy for the next block or the next block's normalised input. All of this work belongs to one token row and shares the same final residual.

The engine's Metal `ffn_out` keeps the row's temporary values in thread-local arrays, computes the branch RMS values, forms the residual, and writes the optional output. Its source comment is a compact specification of both dense and mixture-of-experts behavior. The CPU reference in gemma4.c, lines 1847–1872, expresses the same row operation, while the host submits it at lines 1914–1920. The key result for scheduling is that the output scale and half copy need no extra dispatch.

Activation fusion appears at several scales. The ordinary gate/up matmul tile calculates both projections and writes GELU(g) times u, so neither full gate nor up result must be written as a separate intermediate. The per-layer embedding gate uses the combined matvec path for decode; for batches it computes the gate projection and applies `pl_gate`, whose elementwise work is just GELU times the layer's input. The same principle applies to a clamp placed in a kernel's output store: if the consumer needs a clamped value anyway, clamp before writing it rather than launch a separate pass.

The Metal source has a separate `pl_gate` for the batched case. It reads the gate projection and the strided per-layer input, applies GELU, multiplies, and stores fp16. That is a small kernel, but it transforms a result that the projection kernel produced for a distinct operation. At decode, the engine can fuse this activation and multiplication into the matvec itself. The two routes are shape-dependent, which is why it is useful to read the host branch as well as the kernel name.

Small work that stays small

Not every step can be absorbed into its neighbour. RoPE changes query and key coordinates according to the token position; attention preparation also performs per-head normalisation and stores cache entries. These values are consumed by the attention kernel, but the preparation has a different head-oriented layout and must write the cache. Fusing it into every attention reader could repeat work or complicate ownership. For the current design, it remains its own step.

At the end of the model, softcapping and argmax are a useful pair to inspect. The score kernel can apply the model's optional tanh softcap to each logit while it is already scanning for the maximum. It reduces the maximum across the threadgroup, so the token choice can be made without launching a second independent maximum pass. The standalone `argmax` path handles the simpler greedy selection case. Ties select the first index, matching the engine's deterministic choice. When statistics or saved log-probabilities are requested, the scoring kernel has additional work and cannot be described as "just argmax."

The softcap uses a scaled tanh, so very large logits are pulled toward a finite bound. It does not change which logit is selected when tanh is strictly increasing and all values share one positive cap; it does change their magnitudes, which matters to probability statistics. The same scan can update a running maximum and index while each thread transforms its assigned logits. That is fusion by sharing a read and reduction, not by avoiding all synchronisation: the threadgroup still has to combine partial maxima.

When do command buffers and encoders begin?

The C host calls each `metal_*` wrapper to encode work into the current Metal compute encoder. A sequence of dependent steps can be recorded in one command buffer and committed without making the CPU wait. The GPU executes encoded commands in order, and barriers between dependent dispatches

PREPRINT

preserve the data flow. This avoids a CPU round trip per small operation; it does not erase the GPU's dispatch and drain cost.

There are two explicit scheduling controls. `gpu_group(1)` and `gpu_group(0)` bracket work that is independent and may overlap instead of forcing a drain between every step. The Q/K/V projections are one obvious group: they read the same input but write distinct outputs. `gpu_submit()` commits encoded work to get it moving without waiting. A fence then names work submitted so far, and `gpu_wait(fence)` waits only when the CPU needs a result, such as expert routing or token feedback.

Decode also overlaps adjacent tokens. The CPU can encode work for token $t + 1$ while the GPU executes token t, then the sampled token is fed back on the GPU. The argmax kernel writes the selected ID where the next forward pass reads its input. This avoids waiting for the CPU merely to copy a scalar token between device steps. It is still necessary to wait before reusing a staging slot that the preceding GPU command may be reading.

So "one command buffer per token" is not the right mental model for every moment in the loop. The forward pass groups many layer operations into command buffers, submits at layer boundaries to let work start, and uses fences at real CPU dependencies. Profiling changes this organisation: with `GEMMA4_PROFILE`, each step runs alone in its own command buffer so its GPU time can be attributed. The resulting time includes the isolated command-buffer round trip and cannot be read as the steady-state cost of that kernel in the normal chain.

The real code

Which work belongs to the CPU reference?

The RMS helper in the idea section is the reference for the central normalisation calculation. For the other reference step, the CPU's `score_token` applies the softcap before accumulating its maximum. The loop gives a concrete order: read a logit, optionally transform it, write the transformed value, and track the best value. This chapter's GPU selection path follows the same logical order, with threadgroup reduction instead of a serial maximum.

```
    float softcap;
    float *logits;          // [n][n_vocab], capped in place
    const int *next;        // [n] target tokens
    uint16_t *save;         // [n][NV] or NULL
    const uint16_t *base;   // [n][NV] or NULL
    float *stats;           // [n][4]
} ScoreJob;

static void score_token(void *ctx, int i, int tid) {
    (void)tid;
    const ScoreJob *job = ctx;
    const int V = job->n_vocab, tok = job->next[i];
    float *lg = job->logits + (size_t)i * V;
    float *st = job->stats + 4 * i;
    if (job->softcap > 0)
        for (int j = 0; j < V; j++) lg[j] = job->softcap * tanhf(lg[j] /
    job->softcap);

    float mx = lg[0], mn = lg[0];
    int imax = 0;
    for (int j = 1; j < V; j++) {
        if (lg[j] > mx) { mx = lg[j]; imax = j; }
        if (lg[j] < mn) mn = lg[j];
    }
    double sum = 0;
    for (int j = 0; j < V; j++) sum += expf(lg[j] - mx);
```

```
3732      const float lse = (float)log(sum);
3733      st[0] = mx + lse - lg[tok];
```

Listing 9.4: gemma4.c:3707–3734

The separate CPU GELU reference at gemma4.c:817--818 defines the tanh approximation used by the language-model FFN. The corresponding GPU code uses Metal's precise tanh path for the main GELU helper. The fused gate/up matmul computes the activation before storing the half-precision hidden vector, avoiding an fp32 gate array that would only be read once.

How does the GPU combine scoring and selection?

The softcap and maximum scan are in one thread loop. The excerpt below includes the transformation and reduction, plus the subsequent exponential sum used when score statistics are required. The first-index tie rule comes from the shared block-argmax helper used by both this kernel and the standalone argmax kernel.

```
1561 kernel void score(device float *logits [[buffer(0)]], device const int *
       next [[buffer(1)]],
1562                   device const ushort *base [[buffer(2)]], device ushort
        *save [[buffer(3)]],
1563                   device float *stats [[buffer(4)]], constant
       ScoreParams &p [[buffer(5)]],
1564                   uint row [[threadgroup_position_in_grid]], uint tid [[
       thread_index_in_threadgroup]],
1565                   uint nt [[threads_per_threadgroup]]) {
1566     threadgroup float tv[32];
1567     threadgroup int ti[32];
1568     device float *lg = logits + (ulong)row * p.V;
1569
1570     float mx = -1e38f, mn = 1e38f;
1571     int imax = 0x7fffffff;
1572     for (int j = tid; j < p.V; j += nt) {  // soft-cap in place (each
       thread rereads only its own elements)
1573         float v = lg[j];
1574         if (p.softcap > 0) lg[j] = v = p.softcap * precise::tanh(v / p.
       softcap);
1575         if (v > mx) { mx = v; imax = j; }
1576         mn = min(mn, v);
1577     }
1578     block_argmax(mx, imax, tv, ti, tid, nt);
1579     mn = block_min(mn, tv, tid, nt);
1580     float s = 0;
1581     for (int j = tid; j < p.V; j += nt) s += exp(lg[j] - mx);
1582     const float lse = log(block_sum(s, tv, tid, nt));
1583     const int tok = next[row];
```

Listing 9.5: gemma4_metal.m:1561–1583

The standalone selector at gemma4_metal.m:1542--1555 is small enough to trace in full. Each lane visits a strided subset of vocabulary entries, retains its local best pair (value and index), and participates in a block reduction. Thread zero writes the ID to both destinations. One destination is the sampled result; the other is already staged for the next GPU-side forward pass.

What does the host actually submit?

The forward call sequence shown earlier is the place to confirm the conceptual fusion. It names the next layer's norm weights before the loop, calls attention, then uses add_norm to prepare a->xn. The call

to `ffn` consumes that buffer and its tail may prepare the following layer input. At the end of each layer, `gpu_submit()` lets the GPU start while the CPU proceeds to encode the next work. The command-buffer layer and the tensor-data layer are two views of the same schedule.

The source excerpts are intentionally short and anchored at operations that explain the design. The device kernels are not general vector libraries; they are model steps whose buffer ownership and next consumer determine whether a fusion saves work.

Experiment W0

This toy-only experiment times 1,000 tiny Metal dispatches against one dispatch that writes 1,000 synthetic values. It uses no model or weights. Run it at low priority:
`nice -n 10 swift code/examples/ch09-launch.swift`
The script compiles two tiny kernels, warms them, then records seven alternating paired runs in one command buffer per measurement. The captured output on 2026-10-22 at 13:37 local time, Apple M3, macOS 27.0.1, was 2.810417 ms for 1,000 dispatches and 0.006125 ms for one dispatch, both GPU time; both checksums were 1000. System load averages were 4.36, 4.33, and 4.64; no GPU idle-gate reading was recorded. These toy-only results compare independent writes, not engine timings, and do not estimate the speedup of fusing any particular model step. The full output is saved in `code/expected/ch09-w0.txt`.

Experiment W2

The outline's model-backed profiling experiment uses the snapshot build and a local model path:
`GEMMA4_PROFILE=1 GEMMA4_SKIP=add_norm code/build/gemma4 -m $MODEL -p "hello" --bench 1 1 -r 3`
For timing only, skipping a kernel makes output wrong; the environment-variable comment in `gemma4_metal.m:1627` says this directly. Compare isolated per-kernel GPU times with ordinary end-to-end token time and the historical kbench results: add_norm 18.8 µs, ffn_out 20.5 µs, PLE gate 12.7 µs, dispatch floor 8.5 µs (ENGINE-MATH §11.7, measured 2026-10-06 on a contended machine). Those numbers are historical context, not this run's expected values. Record date, machine, idle gate, and wall/GPU time. **Try it on your machine (W2):** compare each isolated skip-kernel timing with ordinary end-to-end token time. Skipping deliberately breaks output, so interpret the result as timing only. The listed values are dated measurements (ENGINE-MATH §11.7, 2026-10-06, contended machine).

Pitfall

Symptom: an isolated profile reports a tiny kernel taking 190–240 µs, far above the roughly 8.5 µs dispatch floor. **Cause:** `GEMMA4_PROFILE` commits every step in its own command buffer, and the measured interval includes that isolated command-buffer round trip. **Fix:** use the profile to identify which named work ran and its isolated contribution, but compare normal-path timings from ordinary execution; do not label the profile's round trip as kernel execution time. The project recorded this caveat in ENGINE-MATH §11.7 (2026-10-06) and the chapter 6 setup history.

Another limit is the arithmetic itself. Fusion may move a normalisation into the preceding kernel and change the point where intermediate values are rounded or stored. Each candidate therefore needs numerical comparison against the reference path. The documented candidate to merge additional tails into later heads was not implemented: the projected 1.3–1.5

Exercises

1. Count the dependent GPU steps in one layer from `forward` and `attention`. Which three projections are treated as one concurrent group?
2. Estimate the time represented by 500 steps at 10 microseconds per step. How does the answer change if one step is removed from each of 42 layers?
3. Write the residual update and RMS norm as separate array passes, then mark the intermediate arrays a fused `add_norm` can keep local.
4. Trace how `a->xn` is produced after attention and identify the next operation that reads it. What initial operation is still needed before layer zero?
5. Compare the score kernel with standalone argmax. Which extra statistics explain why score does more than choose a token?
6. Run the W0 toy benchmark twice under different load conditions. Keep the checksum check, but explain why its ratio cannot be applied directly to 42-layer engine decode.

Part II recap: make the big work efficient, then shorten its seams

Chapter 6 introduced Metal dispatch, mapped buffers, command buffers, and the CPU fallback. Chapter 7 showed how decode matvecs stream packed weights while fusing input normalisation and gate/up activation work. Chapter 8 used matrix units and reused each decoded weight tile across prompt rows. This chapter followed the smaller steps between those large operations and showed where residual, norm, activation, softcap, and selection work can share data or a reduction. The rebuild path now has three Metal checkpoints: m7a maps and reads embedding rows, m7b checks a `Q8_0` matvec, and m7c checks an 8-by-8 fp16 matrix multiply.

The shared lesson is to follow bytes and dependencies together. A large kernel needs a shape that suits the hardware. A small kernel should be fused only when its inputs are already present, its output can be written for the next consumer, and the numerical checks remain sound. Command buffers keep the CPU from waiting after every encoded dispatch; groups permit independent steps to overlap; fences make CPU waits explicit. The GPU still has real launch costs, so both the kernel design and the schedule matter.

Rebuild checkpoint

Part II checkpoint m7c. Build the small fp16 8×8 matrix kernel from `steps/m7c_matmul.m`, the final Metal checkpoint specified at the end of chapter 8. Compare every output element with a CPU matrix multiply on the same input. Chapter 9 adds no new m1–m8 milestone in OUTLINE.md; its contribution is the scheduling and fusion map needed to understand the later attention and full-forward checkpoints.

PREPRINT

Part III

THE MODEL

PREPRINT

Chapter 10

Attention, step 1: queries, keys, RoPE

Part III: The model

Part I built a map of the file, its numbers, and a plain CPU path. Part II moved that path onto Metal. Part III follows the model's own work: first how tokens look back at earlier tokens, then what must be remembered, and finally how the engine makes those steps fast enough to use. This opening chapter starts with the simplest question: which earlier information should a token bring forward?

The question

What does one attention head do, and what does it need from the past?

Reading map: 14 pages

This chapter is planned for OUTLINE.md's 14-page estimate. Page targets include the diagram, listings, experiment output, and exercises.

Subsection	Pages
Part III opener and question	1
Attention idea: queries, keys, values, softmax, grouping, positions, windows	4
CPU reference and source walkthrough	4
W0 toy and W1 GGUF metadata experiments	2
Documented pitfall	1
Exercises	1
Rebuild checkpoint m5a	1
Total	**14**

The idea

What question does a token ask?

Imagine three tokens have already been processed. Each has left behind two small pieces of information: a key, which describes what kind of information it offers, and a value, which is the information itself. The next token makes a query. A query is a compact way to ask, "Which of those earlier keys is useful for me right now?" A key does not contain the answer; it helps the query choose a value.

Use these toy vectors. Token A leaves key (1,0) and value (10,0); token B leaves key (0,1) and value (0,10); token C leaves key (1,1) and value (5,5). Let C's query be (1,0). Its dot products with the keys are 1, 0, and 1. The first and third keys match this query equally well; the second matches less. The output will therefore mix the three values, with more weight on A and C. Real heads have hundreds of numbers, but the roles are identical.

PREPRINT

How do scores become a useful mixture?

A raw score can be any real number and the scores need not add to one. Softmax subtracts the largest score, exponentiates each difference, and divides by their sum. For our scores (1,0,1), the exponentials are (1, e^{-1}, 1). Their sum is about 2.3679, so the weights are about (0.4223, 0.1554, 0.4223). The weighted value is about (6.3348, 3.6652). The largest scores earn the largest shares, every weight stays positive, and the shares sum to one.

Here is the chapter's one equation. The score is the query/key dot product; softmax turns the list of scores into weights. In this engine's attention path the scale is 1, as its CPU comment says: queries and keys have already been RMS-normalised per head. The CPU writes these probabilities in two passes. The formula below is the online version of the same softmax: after each score x and its value v, keep the running maximum m, the shifted exponential sum l, and the unnormalized weighted value o. At the end divide o by l. The engine's Chapter 10 reference uses two passes for clarity; Chapter 12 uses online updates to stream chunks.

$$m' = \max(m, x), \quad l' = le^{m-m'} + e^{x-m'}, \quad o' = oe^{m-m'} + e^{x-m'}v.$$

Worked with scores (1,0,1) and values ((10,0),(0,10),(5,5)): start with m=-∞, l=0, o=(0,0). After score 1, the state is (1,1,(10,0)); after score 0 it is (1,1.3679,(10,3.679)); after score 1 it is (1,2.3679,(15,8.679)). Divide the last vector by 2.3679 to get approximately (6.3348,3.6652). Equivalently, the normalized weights are (0.4223,0.1554,0.4223).

Why do four queries share one key/value head?

A full multi-head layer could make one key and one value for every query head. Gemma 4 E4B uses grouped-query attention (GQA): multiple query heads share a smaller set of key/value heads. In the model's GGUF header the overall query count is 8; the layer-specific key/value count is 2 for the representative attention layout. Thus four neighboring query heads use one KV head. The query projections remain distinct, so four different questions can be asked of the same remembered key/value stream.

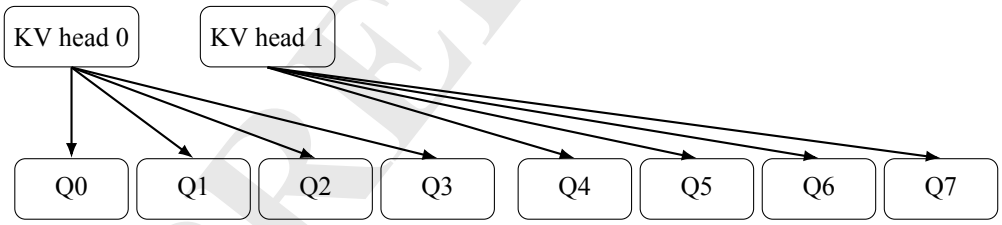


Figure 10.1: Eight query heads, grouped four to each of two key/value heads. The query heads compute separate score lists and separate output mixes.

The exact head dimension is selected per layer from GGUF metadata, not inferred from the model name. The E4B metadata and loader use a 256-number head for sliding-window layers and a 512-number head for global layers. The loader also reads a per-layer KV-head count, so a careful tool prints the array rather than assuming one value applies to every layer. With eight query heads, two KV heads means 4:1 sharing in that layout. The essential memory saving is that keys and values are stored only for the KV heads; each query still gets its own result vector.

Where do the different head widths come from?

The tensor file does not have to tell every reader a story about head width; it records numeric metadata and tensor shapes. The loader turns those values into each layer's `hd`, `n_rot`, `rope_base`, and `n_head_kv`. Sliding layers select the sliding key length, rotation count, and base. Global layers select their corresponding global values, and may also select a frequency tensor. The next excerpt follows that selection before it goes into the per-layer tensor loading.

PREPRINT

Notice that the loop reads the sliding-window pattern for each layer, rather than assuming a repeating pattern in the C code. It then reads `head_count_kv` at that layer index, falling back to the overall query count if the key is absent. For each layer, it selects the sliding or global head width and KV-head count, then checks that the rotary dimensions fit supported limits.

```
962      const int hd_full = (int)kv_num(g, "gemma4.attention.key_length", 0)
         ;
963      const int hd_swa = (int)kv_num(g, "gemma4.attention.key_length_swa",
          hd_full);
964      const int rot_full = (int)kv_num(g, "gemma4.rope.dimension_count",
         hd_full);
965      const int rot_swa = (int)kv_num(g, "gemma4.rope.dimension_count_swa"
         , hd_swa);
966      const float base_full = (float)kv_num(g, "gemma4.rope.freq_base", 1
         e6);
967      const float base_swa = (float)kv_num(g, "gemma4.rope.freq_base_swa",
          1e4);
968      if (kv_num(g, "gemma4.attention.value_length", hd_full) != hd_full
         ||
969          kv_num(g, "gemma4.attention.value_length_swa", hd_swa) != hd_swa
         )
970          die("key_length != value_length");
971
972      m.tok_embd = find_tensor(g, "token_embd.weight", 1);
973      m.output = find_tensor(g, "output.weight", 0);
974      if (!m.output) m.output = m.tok_embd;  // tied embeddings
975      m.output_norm = f32_data(find_tensor(g, "output_norm.weight", 1));
976      m.n_vocab = (int)m.output->ne[1];
977      const Tensor *rope_freqs = find_tensor(g, "rope_freqs.weight", 0);
978
979      for (int l = 0; l < m.n_layer; l++) {
980          Layer *L = &m.layers[l];
981          L->swa = (int)kv_num_at(g, "gemma4.attention.
         sliding_window_pattern", l, 0);
982          L->hd = L->swa ? hd_swa : hd_full;
983          L->n_rot = L->swa ? rot_swa : rot_full;
984          L->rope_base = L->swa ? base_swa : base_full;
985          L->n_head_kv = (int)kv_num_at(g, "gemma4.attention.head_count_kv
         ", l, m.n_head);
986          const Tensor *lf = layer_tensor(g, l, "rope_freqs.weight", 0);
987          L->rope_freqs = L->swa ? NULL : lf ? f32_data(lf) : rope_freqs ?
          f32_data(rope_freqs) : NULL;
988          if (L->hd > 512 || L->hd % 32 || L->n_rot > L->hd || L->n_rot %
         64) die("layer %d: head size %d not supported", l, L->hd);
```

Listing 10.1: `gemma4.c:962–988`

This division of responsibility matters when studying a model file. A W1 script can tell us what the metadata says, but the loader tells us how those values affect this engine. Conversely, a source constant is not evidence that a particular GGUF uses that setting: the model file is what selects the actual layer configuration. The reliable reading traces both sides and then checks the projection shapes.

What does position change?

If keys described only content, two appearances of the same word could look identical even when one came first and one came much later. RoPE changes query and key vectors according to their token position. Picture two coordinates as a tiny arrow: multiplying by a rotation matrix turns it by an angle. In the implementation, coordinate `i` is paired with coordinate `i+n_rot/2`; at position p the pair is turned by

an angle that depends on p, a base, and sometimes a per-pair divisor. The dot product between a rotated query and rotated key then carries information about their relative positions.

The code applies RoPE after per-head RMS normalization. The sliding layers use 256 rotated dimensions and base 10,000. The global layers use a 512-dimension RoPE layout and base 1,000,000, plus GGUF frequency divisors: the first 64 pairs rotate, while the remaining pairs are effectively held still. "Head dimension" and "number of dimensions with useful positional rotation" are therefore distinct ideas even when the configured rotation span equals the head size. The frequency schedule is data from the model metadata, not a hard-coded assumption in the reader.

Why do some layers look back only 512 tokens?

A sliding-window layer at position p can see positions p-511 through p, inclusive once the window is full. A global layer can see every earlier position. This gives the model two kinds of reach: most layers focus on nearby detail, while selected global layers can bring in information from far earlier in the sequence. In the E4B pattern the seven global layers are 6, 12, 18, 24, 30, 36, and 42 (one-based); the other 35 use the 512-position window. This pattern is read from the GGUF array and assigned in the loader.

That distinction matters to computation and storage. A sliding layer doesn't need to score every old position, and the KV-cache chapter will show how its rows can live in a bounded ring. A global layer's attention work grows with the available history and its cache must retain that history. Chapter 12 will make the long scan efficient; this chapter only defines who may participate in the choice.

Why are the score list and the value list different?

It helps to keep three arrays separate in your head. The keys are one row per visible position and one vector per KV head. The score list is one number per visible position for the current query. The values are another row of vectors, again one per position and KV head. The query makes the score list by comparing itself with keys; the score list then controls how much of each value enters this one output. The key helps route attention, while the value carries what gets mixed.

For the three-token example, the key at B is (0,1) and the value is (0,10). Those vectors have deliberately different directions. C's query gives B a smaller score, but B still contributes its 10-valued second component. If keys and values were the same object, the selection signal and payload would be forced to use the same representation. The actual model learns separate projections for Q, K, and V, so it can learn one feature for matching and another for what to retrieve.

An attention output is not a token or a decision. It is one vector for one head and one query position. The outputs of all query heads are laid side by side, then the output projection turns that combined vector back into the layer's working width. The CPU host call shows this final `matmul` after `job.o` has been filled. Residual addition and the feed-forward block happen later in the layer. That is why it is useful to read attention as a subroutine in the layer rather than as the whole transform.

Which positions are legal candidates?

In ordinary text, a token must not read a future token. During generation this is obvious: the future answer has not been produced yet. During prompt prefill the engine can process many prompt tokens in one call, so it must still enforce the same rule for each row. Query position 2 can use keys from positions 0, 1, and 2; position 3 can use 0 through 3. Both rows can be part of one matrix-shaped call, but their visible prefixes differ.

The CPU reference expresses this with two bounds. `s0` is the layer's first visible key. For causal text, `s1` is the current position. The `bidir` branch can instead let a supported call's rows see through the final row in that call; it is a separate mode, not the rule for E4B text. The actual E4B has per-layer embeddings, so its usual text path is causal. Keeping the bounds explicit is important for both correctness and the window boundary: at position 511, the sliding layer can see positions 0 through 511, exactly 512 positions; at 512 it sees 1 through 512.

PREPRINT

A useful hand check is to write the candidate set beside each query. For four prompt positions, the causal sets are: position 0 sees 0; position 1 sees 0–1; position 2 sees 0–2; position 3 sees 0–3. In a four-position sliding toy, query 3 sees 0–3 and query 4 sees 1–4. A mistake that includes one future key can look numerically plausible because softmax still returns positive weights that sum to one. The mask is part of the meaning of attention, not a cleanup after the math.

What does RoPE do to a pair of numbers?

The smallest useful case is a two-coordinate pair. Start with the vector (1,0). At angle 1 radian, the rotated pair is approximately (0.5403,0.8415); at angle 2 radians it is approximately (-0.4161,0.9093). The length stays 1, but its direction changes. In the model's NeoX arrangement the two coordinates of a pair are not adjacent: for a 256-dimension sliding head, index i is paired with i+128; for a 512-dimension global head, i is paired with i+256. The source comment and assignments make that layout visible.

Now let a query be at position 2 and a key be at position 1, and look only at a pair whose frequency is one radian per position. The query turns by 2 radians and the key by 1. Their dot product depends on the one-radian difference between those rotations; if both positions are shifted by the same amount, their difference remains one. This is the useful property: the dot product can distinguish relative displacement without appending a separate integer position to every vector. This two-dimensional calculation is only a picture of one pair, not a claim that every model pair uses that frequency.

The actual angle schedule starts with theta equal to the position. For each pair it multiplies by the base raised to negative 2 over n_rot, then divides by an optional pair-specific factor. This creates a range of rotation speeds. The low-index pairs turn quickly with position; later pairs turn more slowly. Global layers have an additional frequency vector in the GGUF, so the base alone does not describe their full schedule. The CPU code uses the same frequency values that the loader found in the file.

Layer kind	Head dimension	Rotated dimensions	Base
Sliding	256	256 (128 pairs)	10,000
Global	512	512 layout; first 64 pairs rotate usefully	1,000,000

This table keeps three settings separate: the length of Q/K/V vectors in a head, the rotation span selected by the loader, and the per-pair frequency divisor. For global attention, the last setting leaves many coordinates nearly unchanged by position. That behavior belongs to the model's data; a generic RoPE explanation that assumes a single base and every pair rotating would miss it. The `rope_freqs.weight` tensor is a small metadata-linked array of values, and the loader chooses the per-layer version when one exists.

Why does the code use scale 1?

Many transformer descriptions show a dot product divided by the square root of the head width before softmax. This code's CPU comment explicitly says "scale 1: the queries and keys are RMS-normed." The query and key each go through their own RMS normalization for every head, and the learned norm weights are part of that operation. The implementation's stated contract is therefore that the score passed to softmax is the direct dot product of those prepared vectors.

For a toy head with query (1,0) and key (1,1), the score is 1. There is no extra division by a width-derived factor in `attn_task`. This is a useful source-reading checkpoint: a familiar formula from another implementation is not a reason to insert a scaling operation here. The appropriate comparison is the actual engine output and model convention, not an assumed default. Chapter 20's correctness checks are where a changed arithmetic path would be compared numerically.

The code subtracts the largest score before exponentiating even though it does not apply a separate scale. This is numerical stabilization, not another modeling choice. If the scores are (100,99), their direct exponentials are very large; after subtracting 100 they become 1 and `e^-1`, preserving exactly the same normalized proportions while avoiding overflow. The CPU's first pass records the maximum and its

PREPRINT

second pass computes these shifted exponentials. Chapter 12 later changes the algorithm so it need not store every score at once, but keeps the same normalized result.

How does grouped-query attention save work?

If each of the eight query heads had its own KV head, each query would own a separate key/value stream. E4B has two KV heads, so there are only two K/V projection rows per position for the whole group layout. The engine maps the query-head index `h` to a KV-head index using the number of query heads in each group. In the excerpt this is `h / (n_head / n_head_kv)`. Grouping changes which key/value vectors are shared; it does not combine the query projections or merge their attention distributions.

Use a count rather than a vague claim of "less memory." At one position with sliding heads of width 256, eight separate KV heads would each store 256 key and 256 value numbers: 4,096 numbers total. Two KV heads store 1,024 numbers total. The ratio is one quarter. A global head width of 512 gives the corresponding totals 8,192 and 2,048. The actual KV cache adds rows across positions and layers, uses either fp16 or a compact representation, and has shared layers; Chapter 11 accounts for those details. Here the point is only the head grouping.

The same two KV streams can be useful to queries with different roles. For example, four query heads may learn to look for different relationships in the same history, each assigning its own weights across positions. Their outputs are consequently different, even though they fetched keys and values from the same group. This sharing is often less intuitive than saying "one KV head is copied four times": no copying happens in this CPU loop. The `kvh` index points each query's reads at the group's existing rows.

What work can be shared across query positions?

The CPU task is assigned one query head and a small batch of token rows. It loops over each query row, making a fresh score vector and output. This arrangement has two benefits for a readable reference: every query's candidate range is explicit, and the same KV-head mapping is used throughout the task. The cache reads may revisit the same keys for neighboring query positions; a fast prefill kernel can reuse key blocks more effectively, which is one reason Chapter 12 introduces a different kernel shape.

During one-token decode, the query list has one row and the key/value cache contains the past plus the newly written position. During prompt prefill, T can be larger: the code first prepares and stores the new K/V rows, then scores each query against its allowed prefix. This ordering means the current position's key is available when the query attends to itself. It also means a correct implementation has to make sure rows from the same batch do not accidentally see later positions.

The source's score array is sized for `n_past + T`, the largest absolute position range in the call. It only writes scores for the visible interval. That simple allocation and indexing make the reference easy to inspect, though not ideal for a very long context. The fast decode kernel in Chapter 12 streams key chunks and combines partial softmax results; the CPU version here remains valuable because it states the candidate set and output in direct loops.

What does the CPU excerpt do in order?

Trace one query through `attn_task` in source order. The task number first selects a query head and a block of up to 32 query rows. `kvh` maps the selected query head to its shared key/value head. For each query row, the function computes its absolute position and asks `first_key` for the first visible cache position. A causal text query ends at its own position, so it never reads a later token.

The query row has been stored in fp16 after normalization and RoPE. The task converts each component back to float for its dot product. For every visible position it fetches the corresponding key vector and stores the score in `p[s]`. The maximum is found during this pass. In the second pass each score is replaced with its shifted exponential and contributes to `sum`. This two-pass arrangement is intentionally easy to compare to a small hand calculation.

PREPRINT

The function then clears the output vector. In a third pass it fetches each value vector, divides that position's exponential by the shared sum, and adds the weighted components. The result is converted back to fp16. The temporary arrays and vectors are freed before the task returns. The reference uses separate loops because the intermediate score and probability are useful for inspection; a fused kernel can hold partial values in registers instead.

Look closely at the cache indices. The logical sequence position `s` is converted to a row with `s % ring`, while `kvh * hd` selects the vector within that row. For a global cache the row can correspond directly to the sequence position; for a sliding cache the ring reuses storage. The data structure behind `kv_get` is the subject of Chapter 11, but the way this attention loop asks for a row is already visible here.

The use of `p[s]` rather than `p[s-s0]` also explains why the temporary is sized by the absolute end of the call. Only entries in the visible range are read after they are written. That wastes some space when a long past is followed by a short query, but keeps each score aligned with its sequence position. This reference favors transparent indexing over compact scratch storage; a fast kernel in Chapter 12 takes a different approach.

How are projection widths related to heads?

The loader checks the Q projection's output width against the number of query heads times the selected head dimension. It checks the K projection against the number of KV heads times that same dimension. With the metadata values used here, a sliding layer therefore has a Q width of 8 times 256 and a K width of 2 times 256; a global layer uses 8 times 512 and 2 times 512. These are counts of activation numbers, not weight-file byte counts.

Each projection produces a flat row. The attention preparation routine treats it as consecutive head-sized slices: query head 0 begins at offset zero, query head 1 begins one head later, and so on. KV heads are laid out the same way but there are fewer of them. The check in `load_model` catches a mismatch between metadata and tensor shape before attention starts, which is much easier to debug than a bad offset deep in a kernel.

The output projection consumes the concatenated query-head results. This means the hidden width of the layer does not have to equal one head's width, nor does the number of KV heads determine the number of output heads. These are three separate quantities: model width, query-head count times head width, and KV-head count times head width. In E4B, the head width changes with layer kind, so one must follow the layer-specific values through both loader and attention call.

The real code

Where does the CPU reference begin?

Chapter 5 established the contract that a GPU call can decline work and leave a CPU task to do it. Attention follows that same pattern. The CPU function below is not an approximate illustration: it is the reference computation for one head and a small batch of query tokens. It selects a KV head, fetches each visible key, computes dot products, applies a stable two-pass softmax, then fetches values and accumulates their weighted sum. For clarity, these excerpts come from the CPU file before any mention of the fast Metal kernels in Chapter 12.

How does a query find its group's key and value?

This excerpt includes the full scoring and value-mixing path. The variable `h` is the query-head index. Integer division by the number of queries per KV head maps it to `kvh`; with 8 query heads and 2 KV heads, heads 0–3 map to KV head 0 and 4–7 to KV head 1. Each query gets its own softmax weights even when it shares the stored K and V vectors.

```
1320 static void attn_task(void *ctx, int task, int tid) {
1321     (void)tid;
```

PREPRINT

```c
    const AttnJob *job = ctx;
    const Layer *L = job->L;
    const int hd = L->hd, nq = job->m->n_head * hd, nkv = L->n_head_kv * hd;
    const int h = task % job->m->n_head, kvh = h / (job->m->n_head / L->n_head_kv);
    const int t0 = task / job->m->n_head * ATT_QB, t1 = t0 + ATT_QB < job->T ? t0 + ATT_QB : job->T;
    float *p = xmalloc((size_t)(job->n_past + job->T) * sizeof(float)), *o = xmalloc(hd * sizeof(float));
    float *q = xmalloc(hd * sizeof(float)), *kv = xmalloc(hd * sizeof(float));

    for (int t = t0; t < t1; t++) {
        const int pos = job->n_past + t, s0 = first_key(job->m, L, pos), s1 = job->bidir ? job->n_past + job->T - 1 : pos;
        for (int i = 0; i < hd; i++) q[i] = f16_table[job->q[(size_t)t * nq + h * hd + i]];
        float mx = -INFINITY, sum = 0;
        for (int s = s0; s <= s1; s++) {
            kv_get(job->k, job->ks, s % job->ring, nkv, kvh * hd, kv, hd);
            p[s] = dot(q, kv, hd);
            if (p[s] > mx) mx = p[s];
        }
        for (int s = s0; s <= s1; s++) { p[s] = expf(p[s] - mx); sum += p[s]; }
        for (int i = 0; i < hd; i++) o[i] = 0;
        for (int s = s0; s <= s1; s++) {
            kv_get(job->v, job->vs, s % job->ring, nkv, kvh * hd, kv, hd);
            const float w = p[s] / sum;
            for (int i = 0; i < hd; i++) o[i] += w * kv[i];
        }
        for (int i = 0; i < hd; i++) job->o[(size_t)t * nq + h * hd + i] = float_to_half(o[i]);
    }
    free(p);
    free(o);
    free(q);
    free(kv);
}
```

Listing 10.2: `gemma4.c:1320–1352`

The inner loops use a temporary score array. First, the code finds the maximum score; then it exponentiates each score relative to that maximum and sums. Subtracting the maximum changes no normalized weight, but prevents exponentials of large positive values. Finally, the normalized probabilities weight the values. Chapter 12 will replace this clear path with chunked and online algorithms that avoid retaining the whole score list.

What prepares vectors before those loops?

Attention does not consume the raw projection arrays directly. The query is RMS-normalised with its learned query weights and rotated; the new value is RMS-normalised without a learned vector; the key is normalized with its learned key weights, rotated, and written to the cache. The routine below shows this order. In a shared-KV layer, `q_only` suppresses new K/V writes because that layer reads a prior cache; the cache chapter explains the lifetime and storage details.

```c
// token t: RMSNorm and rotate the queries and the new keys; the values get an RMSNorm without weight.
// Keys and values go into the cache.
static void attn_prep(void *ctx, int t, int tid) {
    (void)tid;
    const AttnJob *job = ctx;
    const Model *m = job->m;
    const Layer *L = job->L;
    const int hd = L->hd, nq = m->n_head * hd, nkv = L->n_head_kv * hd, pos = job->n_past + t, row = pos % job->ring;
    float x[512];
    for (int h = 0; h < m->n_head; h++) {
        rmsnorm(x, job->q_new + (size_t)t * nq + h * hd, L->q_norm, hd, m->eps);
        rope(x, pos, L->n_rot, L->rope_base, L->rope_freqs);
        for (int i = 0; i < hd; i++) job->q[(size_t)t * nq + h * hd + i] = float_to_half(x[i]);
    }
    for (int h = 0; h < L->n_head_kv && !job->q_only; h++) {
        rmsnorm(x, job->v_new + (size_t)t * nkv + h * hd, NULL, hd, m->eps);
        kv_put(job->v, job->vs, row, nkv, h * hd, x, hd);
        rmsnorm(x, job->k_new + (size_t)t * nkv + h * hd, L->k_norm, hd, m->eps);
        rope(x, pos, L->n_rot, L->rope_base, L->rope_freqs);
        kv_put(job->k, job->ks, row, nkv, h * hd, x, hd);
    }
}
```

Listing 10.3: gemma4.c:1290–1311

The very small CPU RoPE helper spells out the paired-coordinate operation. For a pair x_0, x_1, it saves both old coordinates before writing the rotated pair, so the second assignment cannot accidentally use the first newly written value. It rotates only the first `n_rot` dimensions; any remaining head coordinates would pass through unchanged.

```c
// NeoX-style RoPE on the first n_rot dims of a head: pair (i, i + n_rot/2) turns by pos·base^(-2i/n_rot)/ff[i].
// The angle is built as ggml's CPU rope does (theta *= base^(-2/n_rot) per pair).
static void rope(float *x, int pos, int n_rot, float base, const float *ff) {
    const int half = n_rot / 2;
    const float theta_scale = powf(base, -2.0f / n_rot);
    float theta = (float)pos;
    for (int i = 0; i < half; i++, theta *= theta_scale) {
        const float th = theta / (ff ? ff[i] : 1.0f);
        const float c = cosf(th), s = sinf(th), x0 = x[i], x1 = x[i + half];
        x[i] = x0 * c - x1 * s;
        x[i + half] = x0 * s + x1 * c;
    }
}
```

Listing 10.4: gemma4.c:1245–1258

How do sliding and global layers choose visible keys?

The CPU reference calculates the first visible key from the current position and layer type. The main loop's upper bound is the current token for causal text; the optional bidirectional case is reserved for other model configurations. Here the assignment identifies the sliding layer's 512-position window while leaving a global layer's first position at zero.

```
1315 // first key a query at `pos` sees: all before it, or the last n_swa in
       sliding-window layers
1316 static int first_key(const Model *m, const Layer *L, int pos) { return L
       ->swa && pos >= m->n_swa ? pos - m->n_swa + 1 : 0; }
```

Listing 10.5: gemma4.c:1315–1316

Where does the layer call the CPU reference?

The host first forms Q, K, and V projections. It then calls the preparation step through STEP; if Metal does not accept the arguments, the CPU worker runs attn_prep. The attention step uses the same contract, dispatching CPU tasks over query heads and groups of query tokens. After the attention result is assembled, the output projection maps the concatenated heads back to the layer width. This is the bridge from the plain reference to the later fast kernels.

```
1355 static void attention(const Model *m, const Layer *L, int l, Acts *a,
       State *st, int T, int np) {
1356     const int hd = L->hd, nkv = L->n_head_kv * hd, q_only = L->kv_src !=
         l;
1357     float *k_new = a->kv, *v_new = L->wv ? a->kv + (size_t)a->T_max *
       nkv : k_new;
1358     const int bidir = a->bidir && L->swa, ring = kv_rows(L, st->n_ctx,
       st->ring);
1359     AttnJob job = {m, L, T, np, a->q, k_new, v_new, a->qr, st->k[l], st
       ->v[l], st->ks[l], st->vs[l], ring, a->o, bidir, q_only};
1360
1361     gpu_group(1);
1362     matmul(a->q, a->xn, L->wq, T, 0);
1363     if (!q_only) matmul(k_new, a->xn, L->wk, T, 0);
1364     if (!q_only && L->wv) matmul(v_new, a->xn, L->wv, T, 0);
1365     gpu_group(0);
1366     STEP(metal_attn_prep(job.q, job.k, job.v, job.ks, job.vs, job.q_new,
         k_new, v_new, L->q_norm,
1367                          q_only ? L->q_norm : L->k_norm, L->rope_freqs,
       T, np, m->n_head, L->n_head_kv, hd, L->n_rot,
1368                          L->rope_base, m->eps, ring, q_only),
1369          T, attn_prep, &job);
1370     STEP(metal_attention(job.o, job.q, job.k, job.v, job.ks, job.vs, T,
       np, m->n_head, L->n_head_kv, hd,
1371                          L->swa ? m->n_swa : 0, bidir, ring),
1372          m->n_head * ((T + ATT_QB - 1) / ATT_QB), attn_task, &job);
1373     matmul(a->a, job.o, L->wo, T, 0);
1374 }
```

Listing 10.6: gemma4.c:1355–1374

The project contains an easy-to-misread shared-layer case. A layer that has no K/V projection of its own still has query heads, but reads a compatible earlier cache. In the preparation call, the conditional selects the query normalization pointer for the absent key-normalization pointer. That is intentional: no new key is normalized there. The next chapter follows how the shared cache is allocated and read.

Experiment

Can we inspect attention settings without opening tensors?

This is W1. The helper below reads the GGUF signature and typed key/value entries, then stops at the end of metadata. It prints the declared tensor count but does not parse tensor records or touch weight payloads. It uses only Python's standard library and does not initialize the engine or load the model.

```
python3 code/ch10_gguf_attention.py "$MODEL"
```

Look at the head count, the per-layer KV-head array, key lengths, shared-KV count, sliding window, and pattern. A captured run on the local E4B file reports 42 layers, 8 query heads, 2 KV heads for the attention layout, key lengths of 512 globally and 256 in sliding layers, a 512-position window, and 18 shared-KV layers. The pattern identifies global layers 6, 12, 18, 24, 30, 36, and 42. This is a metadata-only inspection; the output is kept in `code/expected/ch10-w1.txt`.

The capture includes the GGUF version and tensor-record count in its first line, but deliberately says the records were not read. The scalar head count and lengths describe the common layout; the per-layer array shows which entries use the sliding configuration. A `True` entry in that array is a sliding layer in this loader; a `False` entry is global. The reported layer list is one-based for readers, while the array itself uses the zero-based indices used by C.

The script is a narrow inspector, not a complete GGUF validator. It checks the magic and rejects truncated metadata, but it does not verify tensor offsets or payload checksums. Those are different questions for Chapter 3's tooling. Its value here is that the attention settings can be read directly from the header without mapping or decoding the multi-gigabyte tensor area.

Can three toy tokens check the whole calculation?

The C program below contains the same three-key example from the idea section. Run it without a model:

```
cc -O2 code/examples/ch10_attention.c -lm -o code/build/ch10_attention
code/build/ch10_attention
```

Its captured W0 output is in `code/expected/ch10-w0.txt`. The expected scores are (1,0,1), and the probabilities are approximately (0.4223,0.1554,0.4223) because the calculation subtracts the maximum score before exponentiating. The resulting value mix is (6.3348,3.6652) to four decimal places. The program uses no NumPy, weights, or inference. It is intentionally a one-query illustration; causal masking and multiple query heads are introduced in the source path above.

The full captured output is short enough to audit:

```
scores: 1 0 1
weights: 0.4223 0.1554 0.4223
output: 6.3348 3.6652
```

The program stores only three scores and uses a maximum-subtracted exponential, so it is a useful check of arithmetic and not a performance benchmark. Changing the input changes the result deterministically; there is no random seed or model state to record.

What can the three-token toy verify?

The W0 example checks several small but important things at once. It checks that the score is the dot product of the selected query and key; that subtracting the maximum before exponentiating preserves the normalized proportions; that the weights are applied to values rather than keys; and that the resulting output has the same width as each value. Since its vectors have only two components and there are three positions, the printed result can be checked by hand without a model or numerical library.

PREPRINT

It cannot check whether the model's learned projections are correct, whether the Q/K RMS weights were loaded correctly, whether RoPE's model-specific frequency data was interpreted properly, or whether an implementation respects causal and sliding masks. Those are separate concerns. The model-free program is not a miniature Gemma 4; it is a small arithmetic check that lets us recognize the shape of attention before reading the model path.

A useful extension is to add a mask before softmax. For a query at position 1 in a four-token sequence, scores for positions 2 and 3 must not enter the normalization sum at all. Setting them to an extremely negative sentinel is one implementation technique; omitting them from the visible loop is what the CPU reference does. Merely setting their values to zero is wrong: they would still take a share of the normalized probability and reduce the contribution of legal values.

Another useful extension is to add a second query head while retaining the same keys and values. Each head should get its own query and its own score list. If both queries are equal, their outputs should match; if the queries differ, the probabilities may differ even though the K/V arrays are shared. That is the essential GQA distinction in a compact test: reuse the memory, not the query's decision.

Finally, compare a direct two-pass calculation with the online state update shown earlier. Feed the same three score/value pairs in the same order. The final maximum, exponential sum, and unnormalized weighted vector should produce the same output after division. The online path is the bridge to Chapter 12, where a long history is processed in chunks without storing one score per past position. At this stage, the direct CPU loop remains the easier result to inspect when a later kernel disagrees.

Pitfalls from the project notes

Pitfall

A shared key-normalization pointer can look like a typo. In a layer whose K/V comes from an earlier shared cache, there is no new key projection to normalize. The call therefore passes `q_norm` in the slot normally used for `k_norm`; the preparation routine's `q_only` flag skips its key/value loop entirely. The cause is shared K/V, not a mismatched tensor. The fix when reading this path is to follow the flag and cache source together. The implementation is documented at `gemma4.c:997--1008` and the call at `gemma4.c:1366--1369`; the outline records that the unused K/V weights still exist in the file.

Exercises

Exercises

1. **Read.** Work out the normalized softmax weights for scores (2,1,0). Show the maximum subtraction and explain why the final weights add to one.
2. **Read.** For 8 query heads and 2 KV heads, write the query-head to KV-head mapping for all heads. Why does sharing not force the queries to use the same output?
3. **Change.** In the W0 toy, change the key for token B to (2,0). Predict the new weights before running the program; then update its printout to show each key's score.
4. **Measure.** Change the toy to eight query vectors grouped over two KV heads. Count how many key/value vectors are stored versus the eight independent query vectors. State the reduction in K/V vector count.
5. **Read.** Use the W1 pattern to list all global layers and explain which layer can still see token 0 when processing token 700.
6. **Change.** Set the toy sliding window to 4 and print visible positions for queries at positions 2, 4, and 6. Keep the current position visible and explain the off-by-one boundary.

Rebuild checkpoint

Rebuild checkpoint

Milestone m5a: one attention head with RoPE. Implement `steps/m5a_attention.c` for a single-head, four-token input. Apply the model's q/k normalization and RoPE to layer 0's real projected q and k vectors, use its values, and compare the resulting attention vector with an engine-side dump. The implementation should separate projection, preparation, visible-key selection, score normalization, and weighted value accumulation so each part can be checked. This checkpoint stops before the cache is packed into its final ring and before the online kernels of Chapter 12.

PREPRINT

Chapter 11

The KV cache

Part III: The model

Chapter 10 followed a query as it asked earlier positions for useful values. This chapter asks what the engine must keep so those old positions can answer again, and how Gemma 4 E4B keeps that memory bounded where it can.

The question

What must the engine remember about the past, and how little can it store without changing the answer too much?

Reading map: 18 pages

This first pass assigns the full outline estimate before filling it. Page targets include diagrams, listings, the captured W0 output, exercise hints, and the checkpoint.

Subsection	Pages
Question, cache intuition, and data-flow diagram	2
E4B dimensions and per-position arithmetic	3
Shared KV layers and layer ownership	2
8-bit blocks, scales, and fp16 alternative	2
Ring storage, context growth, and rollback	3
Chat prefix reuse and the cache boundary	1
Source walkthroughs and code listings	2
W0 calculator/ring experiment and W1 metadata check	1
Measured accuracy trade and documented pitfalls	1
Exercises, Appendix F hints, and checkpoint m5b	1
Total	**18**

The idea

Why not calculate the past again?

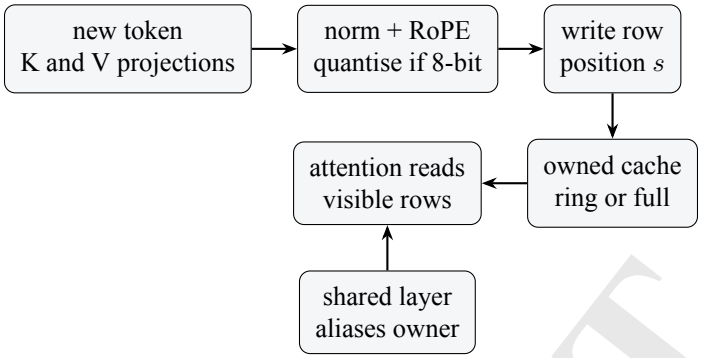
A token at position 900 can use the keys and values created at positions 0 through 899. Those vectors came from the earlier tokens and their layer inputs. They will not change just because position 900 has arrived. Rebuilding them would repeat the projections, normalization, and position rotation for every earlier token at every new step. Instead, the engine writes each key and value once and reads them again when attention needs them.

A cache row is not the token text, and it is not a copy of the hidden state. It contains the layer's projected key and value vectors, after the preparation the attention path requires. The key helps the query

PREPRINT

score a position; the value is the payload that gets mixed into the result. Chapter 10 separated those jobs. Here they share a storage policy because both are needed for every visible old position.

The moving parts fit in one picture. A new token is projected and prepared, written into the owning layer's row, and later read by attention. The top set of layers marked as shared points to an earlier owner's cache instead of allocating another one.



The diagram shows where compression occurs: after the new key/value vectors exist and before they occupy their cache row. Query vectors are prepared for the current step but are not retained as a history.

Sidebar: a different cache in the equation-paper engine

The equation-paper implementation in `gemma4_eq_gpu-rev1.m` defaults to fp32 KV storage (`g_kv_q8 = false`). Its allocation loop gives global layers `ctx_capacity` rows and sliding layers the smaller of that capacity and 512 rows; the separate legacy-attention mode gives every layer the full capacity. This source therefore confirms the mixed full-global / 512-row sliding design for its ordinary path, rather than a fixed 512-row cache for every layer. Its architecture constants describe the E2B model, with one KV head and 15 owners, so its byte counts should not be compared directly with E4B's. The profile records Rev-2 decode rate dropping from 19.91 to 4.28 tokens/s as the actual prompt grows from 35 to 2,214 tokens, while the comparison engine's measured rate rises from 23.9 to 48.3. The profile warns that these runs do not establish that cache policy alone caused the trend; its runs and workloads differ. Source: `gemma4_eq_gpu-rev1.m:2298--2318`; `scratch/gemma4-eq-repro/PROFILE.md`.

Can we budget a context with reply slack?

Take an 8,192-position context and reserve 96 sliding rows of slack for the reply and in-flight work. The source's ring rule adds the 512-position sliding window and eight safety rows, then rounds up to a multiple of eight. Here 512 + 96 + 8 = 616, which is already aligned. Each sliding owner therefore reserves 616 rows, while each global owner reserves all 8,192 context rows.

The sliding side contains 20 × 616 × 1,024 = 12,615,680 K/V values. The global side contains 4 × 8,192 × 2,048 = 67,108,864. Together that is 79,724,544 values. At two bytes each, the cache takes 159,449,088 bytes in fp16. At Q8, the same values take one byte each and one half scale per 32 values, for 84,707,328 bytes. This is an estimate from the row policy and dimensions; it excludes alignment and activation scratch.

If we instead gave every sliding layer all 8,192 positions, it would reserve 167,772,160 sliding values rather than 12,615,680. The row policy avoids more than 150 million values in the sliding portion alone. That saved storage does not remove the global cost: the four global owners still need every position in the context. This is the practical reason to distinguish the two attention patterns when planning memory.

The slack number is a design input, not a promise that every reply will use exactly that many rows. A larger prompt chunk or reserved generation length can require more room before a write wraps. If the context is small enough that 616 rows would equal or exceed it, the state uses the context-sized allocation instead. If the context later grows and the ring calculation changes, the grow routine refuses

to reinterpret the old rows. A budget should therefore record context, slack, row alignment, and cache precision together.

What if the engine kept a separate cache for every layer?

The shared layers save more than one row of bookkeeping. An E4B layer with a compatible query head can reuse an earlier layer's keys and values, so the model computes no new K/V projection there and allocates no new history. The number of consumers is still 42: every layer performs attention with its query. The number of producers is 24: those are the layers that write cache rows. It is easy to confuse these counts because the model's forward pass still visits all 42 layers.

Without sharing, the same 42-layer layout would keep 35 sliding caches and 7 global caches. With sharing, the first 24 own rows: 20 sliding and 4 global; the last 18 reuse one of those compatible owners. A per-position fp16 count without sharing would be $35 \times 1{,}024 + 7 \times 2{,}048 = 50{,}176$ values, while the actual owners account for 28,672. The shared design therefore avoids storing 21,504 values for each newly retained position, before considering that sliding owners stop growing at their ring size. This arithmetic counts storage only; it does not claim the shared consumers have no attention compute.

The layer pairing is structural. The loader searches prior layers for the same sliding/global flag, and verifies that the head width and KV-head count agree. A sliding consumer cannot borrow a global source whose key vectors have a different width or visibility rule. The model's metadata and projection tensor shapes make these constraints visible. The unused K/V tensors for shared layers remain in the GGUF file, but the model loader does not request them for execution and memory mapping does not need to wire them for the GPU.

The final eighteen layers can still have a large effect on the output. They compute their own queries, score the shared key sequence, mix the shared values, and continue through their own output projections and feed-forward blocks. Sharing the cache reuses the source memory and K/V vectors; it does not merge those layers into one. This distinction is the same kind of distinction as grouping query heads over KV heads in Chapter 10: repeated readers can share a compact memory while retaining distinct queries and outputs.

What does one E4B position cost?

Use the GGUF's actual projection widths. A sliding-window owner's K projection has 512 outputs: two KV heads times 256 values per head. Its V projection has the same width. Across both K and V that is 1,024 stored values at a position. A global owner's projection has 1,024 outputs per K or V: two KV heads times 512 values per head, or 2,048 values for K and V together.

The E4B's 42 layers do not each own a separate cache. Twenty sliding owners and four global owners allocate storage; eighteen later layers share one of those caches. Thus a newly retained position accounts for $20 \times 1{,}024 = 20{,}480$ sliding values, plus $4 \times 2{,}048 = 8{,}192$ global values: 28,672 values total across the owners. At two bytes each, this is 57,344 bytes per position, about 56 KiB, before allocator alignment. This per-position count is the full-history view: the sliding part stops growing once its rings are full.

For a 32-value block, 8-bit storage uses 32 one-byte integers and one 2-byte half scale. That is 34 bytes, or 1.0625 bytes per value. Applying it to the same 28,672 values gives 30,464 bytes per newly retained position, about 29.75 KiB. The savings approach one half at long contexts, but there is metadata for every block, so it is not exactly half. The source has one scale array for keys and another for values, and the K and V block boundaries are aligned.

This arithmetic is a useful check on larger totals. At 4,096 positions, four global owners account for $4 \times 4{,}096 \times 2{,}048 = 33{,}554{,}432$ values; at two bytes each, that is 67,108,864 bytes. The twenty sliding owners use rings, not 4,096 rows. The README reports 133 MB total fp16 cache at this setting, including those rings; its corresponding 8-bit total is 71 MB. At 32K, the README's fp16 total is 580 MB, of which 512 MB is the four global caches, and the 8-bit total is 308 MB. These are README measurements, not fresh measurements in this writing task.

PREPRINT

How much does each owner row hold?

The per-layer numbers make the total easier to audit. An owner row stores both K and V. Q8's scale bytes are counted for each separately, because the two projections are quantised independently. The following table counts the physical contents of one row; the final column includes both arrays and their scales.

Kind	Owners	KV heads	Head width	K+V values	fp16 B	Q8 B
Sliding	20	2	256	1,024	2,048	1,088
Global	4	2	512	2,048	4,096	2,176

For a sliding row, K has $2 \times 256 = 512$ values and V has another 512. In Q8, one array of 512 values uses 512 integer bytes and 16 two-byte scales, or 544 bytes. The K and V arrays together use 1,088 bytes. A scale covers 32 values, so 512 values require 16 scales; this is 32 bytes of scale storage per array. This worked count is useful because a scale is not 2 bytes per 32-value block per key-value pair; the 32-value block is counted separately for K and for V.

The global owner row doubles the width, so it doubles both the values and scale count. Across 20 sliding and 4 global owners, the row values add to 28,672 per position. The Q8 row bytes add to $20 \times 1{,}088 + 4 \times 2{,}176 = 30{,}464$. If the layer count were accidentally 42 instead of 24, the totals would be too large; if the shared layers were omitted from attention itself, the cache would be smaller but the model computation would also be wrong. The row table separates what is stored from who reads it.

How does the total grow with context?

It helps to separate a row count from a row size. One sliding owner's row contains 1,024 values across K and V. One global owner's row contains 2,048. With four global owners and twenty sliding owners, a full allocation at 4,096 positions would reserve $20 \times 4{,}096 \times 1{,}024$ plus $4 \times 4{,}096 \times 2{,}048$, or 117,440,512 fp16 values. That would use about 234.9 MB. The engine does not reserve that much for the sliding layers. At 1,600 ring rows, the sliding part holds $20 \times 1{,}600 \times 1{,}024$ values, while global layers retain all 4,096 positions. The resulting fp16 allocation is 132,644,864 bytes, about 133 MB. With scale overhead, 8-bit uses about 70.5 MB, rounded to the README's 71 MB.

At a 32K context, the global component grows eight times from 4,096 positions: its fp16 storage is 536,870,912 bytes, about 512 MiB. The sliding component remains near the same ring capacity. That is why the README's whole cache grows from 133 MB at 4K to 580 MB at 32K, rather than eight times 133 MB. The exact reported totals depend on decimal-versus-binary display units and the rounded row counts; these hand calculations are a consistency check, not a new measurement.

The result also explains where the future memory pressure comes from. Adding another thousand context positions adds rows to the four global owners, while the sliding owners still use the same ring. Increasing reply slack can enlarge those rings because the engine wants room to process the next chunk before overwriting rows the current attention call needs. The memory cost of slack is bounded by twenty ring owners; the cost of global history is proportional to context length.

Cache region	Owners	Rows	fp16 bytes
Sliding, 4K context	20	1,600	65,536,000
Global, 4K context	4	4,096	67,108,864
Total, 4K context	24	mixed	132,644,864
Global, 32K context	4	32,768	536,870,912

Each entry counts both keys and values. The row counts are already the physical storage policy: sliding rows are capped, global rows follow context. The totals line up with the README's rounded 133 MB and 580 MB results once the fixed sliding rings are included.

What does a block scale preserve?

The scale is chosen independently for every 32 adjacent cache values. That keeps a large coordinate from setting the precision of an entire head, while also avoiding a scale word for every value. Consider the

PREPRINT

32-value block whose first values are 0.37, -0.82, and 1.27 and whose largest absolute value is 1.27. Its scale is 0.01 after conversion to half. Those three entries are stored as 37, -82, and 127, and read back as 0.37, -0.82, and 1.27. A value such as 0.374 rounds to 37, so it returns as 0.37; the error is -0.004. A value only 0.002 wide near zero cannot be represented in this block except as zero or a whole 0.01 step.

The scale does not promise the same absolute error for every block. A block whose largest magnitude is 0.127 has a scale near 0.001, so it can represent smaller steps. A single outlier can make all 32 values use a larger step. This is the compromise behind block quantisation: compact integers plus enough local scale information to preserve the relative structure of most values. Keys and values are quantised separately, so a large key coordinate cannot set the scale for its paired value block.

There are 64 values per query-head pair when one key vector and one value vector each have 32 coordinates, but the actual block spans the entire KV row and advances in groups of 32 flattened values. Since the row length is a multiple of 32, no scale crosses a row boundary. At one position the E4B has 20,480 sliding values and 8,192 global values across all owners. Each of those owner rows divides cleanly into 32-value blocks: 640 sliding blocks and 256 global blocks. Each scale occupies two bytes, so scale storage is 1,792 bytes for a newly stored position, in addition to its 28,672 integer bytes. That gives the 30,464-byte 8-bit figure without relying on an average compression ratio.

The CPU reference deliberately stores the scale as fp16 before using its decoded value. If it quantised using a full-precision scale and only rounded the scale when saving, its stored integers could disagree with the GPU path near half-way rounding boundaries. The source's `h` variable is converted back through the fp16 lookup table before dividing the block values. Small implementation details like this matter when the two paths must represent the same cache state.

What is the physical layout of a row?

A row is a flat sequence of values for all KV heads of one layer. With two KV heads and a 256-value sliding head, the K part contains head 0's 256 values followed by head 1's 256 values. The V part is laid out the same way. The next token writes its own row; it does not append a token ID or a whole hidden vector. In fp16 the row's K array and V array each use 1,024 half values. In 8-bit mode, each of those arrays uses 1,024 bytes plus 32 half scales. The key scales and value scales are kept in separate arrays.

The row number and the coordinate inside that row are different indices. First choose the layer's physical row: the absolute position for a global layer, or absolute position modulo the ring length for a sliding layer. Then choose the KV head and coordinate within it. The code passes a flat offset into the store routine; the higher-level attention preparation code has already selected the row. A bug in either index can still produce plausible values: it may read a neighboring head or a stale position without leaving the allocated array.

The layout keeps an entire row's blocks aligned. Every KV width is divisible by 32, so block boundaries fall at fixed offsets. With an offset of 32, for example, the next scale belongs to values 32 through 63 in that row. A scale index is not simply the value index; it is the row's scale stride plus the offset's block number. The source uses `nkv / 32` as the per-row scale stride. When inspecting an 8-bit cache dump, always record the row, head, value offset, and scale index together.

The allocation is one shared block for all layers that own memory. Each owner gets a K segment and a V segment, and in Q8 mode each segment is followed by its scale region. The layer's pointers are set as the allocation cursor advances. Shared layers alias the owner's pointers after that allocation is laid out. This avoids one allocator call per head, token, or layer and gives the GPU a contiguous memory region to read. The price of a flat allocation is that its pointer arithmetic must stay in sync with the size calculation; that is why `kv_layer_bytes` and the state constructor use the same row width and quantisation policy.

The new block is zero-filled. Attention can ask for masked positions beyond the sequence's valid end during a batched or padded call, and deterministic zero contents keep such reads defined. Causal masks still decide whether those positions contribute. Zero filling does not make a padded row meaningful; it only prevents uninitialized memory from leaking into a calculation before masking.

PREPRINT

Why do only 24 layers allocate memory?

E4B has 18 shared-KV layers. A shared layer still computes its own query, but it reads the key/value sequence produced by a compatible earlier layer. It must match the source layer's head size and KV-head count. This is why the cache owner count is smaller than the transformer layer count, and why the model file can contain K/V projection weights that are never used in those shared layers.

The loader builds the sharing relation by looking backward for a prior layer with the same sliding/global kind. It fails if no compatible source is found. The first excerpt is the important choice: each layer starts as its own source, then the later shared layers are assigned to a previous matching layer. Only owners load K and V projections. Chapter 10 noted the curious normalization pointer in the shared call; the source confirms that shared layers have no new K projection to normalize.

```
996         L->kv_src = l;
997         for (int j = n_kv_layers - 1; l >= n_kv_layers && j >= 0; j--)
998             if ((int)kv_num_at(g, "gemma4.attention.
    sliding_window_pattern", j, 0) == L->swa) { L->kv_src = j; break; }
999         if (l >= n_kv_layers && L->kv_src == l) die("layer %d: no layer
    to share the KV cache of", l);
1000        if (L->kv_src == l) {
1001            L->wk = layer_tensor(g, l, "attn_k.weight", 1);
1002            L->wv = layer_tensor(g, l, "attn_v.weight", 0);
1003            if (L->wk->ne[1] != (int64_t)L->n_head_kv * L->hd) die("
    layer %d: unexpected attention shapes", l);
1004        } else {
1005            const Layer *S = &m.layers[L->kv_src];
1006            if (S->hd != L->hd || S->n_head_kv != L->n_head_kv) die("
    layer %d: its KV cache's shapes differ", l);
1007            L->k_norm = NULL;  // (keys come from the cache)
1008        }
1009        if (L->wq->ne[1] != (int64_t)m.n_head * L->hd) die("layer %d:
    unexpected attention shapes", l);
```

Listing 11.1: gemma4.c:996–1009

At state construction, a shared layer's K, V, and scale pointers are copied from the owner. It does not merely happen to get the same numbers; both layers address the same cache allocation. That distinction matters for arithmetic: count allocations by owners, while count attention reads by every layer that consumes them.

```
1180 static State state_new(const Model *m, int n_ctx, int slack) {
1181     State st = {.n_ctx = (n_ctx + 7) / 8 * 8, .ring = kv_ring(m, n_ctx,
    slack)};
1182     const size_t total = kv_bytes(m, st.n_ctx, slack);
1183     char *p = st.mem = shared_alloc(total);  // one allocation; zeroed,
    as positions past the sequence are read (masked)
1184     memset(p, 0, total);
1185     for (int l = 0; l < m->n_layer; l++) {
1186         const Layer *L = &m->layers[l];
1187         if (L->kv_src != l) {  // a shared-KV layer reads its source's
    cache (an earlier layer)
1188             st.k[l] = st.k[L->kv_src], st.v[l] = st.v[L->kv_src];
1189             st.ks[l] = st.ks[L->kv_src], st.vs[l] = st.vs[L->kv_src];
1190             continue;
1191         }
1192         const size_t n = (size_t)kv_rows(L, st.n_ctx, st.ring) * L->
    n_head_kv * L->hd;
1193         for (int kv = 0; kv < 2; kv++) {
1194             *(kv ? &st.v[l] : &st.k[l]) = p, p += kv_q8 ? n : n * sizeof
    (f16);
```

```
1195             *(kv ? &st.vs[l] : &st.ks[l]) = kv_q8 ? (f16 *)p : NULL, p
     += kv_q8 ? n / 32 * sizeof(f16) : 0;
1196         }
1197     }
1198     return st;
1199 }
1200
1201 // The cache made room for n_ctx positions (more than it has), its
     contents kept: only the global layers' caches
```

Listing 11.2: `gemma4.c`:1180–1201

The E4B pattern makes the source selection concrete. Layer numbers below are zero-based, as in the C arrays. The last eighteen layers begin at 24. Their sliding layers use layer 22's cache, and their global layers use layer 23's cache. The earlier owner layers still have their own caches; the later readers do not cause a second allocation.

Layer kind	Owner range	Shared consumers and source
Sliding	0–4, 6–10, 12–16, 18–22	24–28, 30–34, 36–40 use 22
Global	5, 11, 17, 23	29, 35, 41 use 23

This listing of sources is the result for the E4B pattern, not a constant baked into the engine. The code compares the metadata's sliding flag and uses the nearest preceding compatible source. Another model with a different pattern can produce a different pairing. The memory arithmetic here uses the E4B's actual 24-owner arrangement; the loader is what generalizes it.

Why use a ring for most layers?

A sliding layer can look back only 512 positions. Once the sequence is much longer than that, its older rows cannot be read by that layer again. Its storage can be reused: position s is stored in row s modulo the ring length. When a new position reaches a row, it replaces the old position that used that row. A global layer cannot discard old positions, so its storage follows the current context length.

The ring is slightly larger than the model's visible window. The engine adds slack for prompt chunks and the longest expected reply, then rounds dimensions to multiples of eight. This lets prefill write several rows without immediately colliding with rows still needed by attention. At very short contexts, the ring can simply be the whole context; it is a ring only when its length is smaller than the full allocation.

Here is a toy ring of five rows. After writing positions 0 through 7, rows contain 5, 6, 7, 3, and 4. The next position 8 goes to row 3 and replaces position 3. A sliding query at 8 needs positions 4 through 8, which are exactly rows 4, 0, 1, 2, and 3. It does not need position 3, the row that was replaced. In the real model the ring is at least the 512-position window plus write slack, so a chunk can be prepared safely.

The storage calculator distinguishes the two cases per layer: a sliding layer gets the ring row count; a global layer gets the rounded context row count. The next code totals only owner layers and accounts for both K and V. That is the central arithmetic in the implementation, written as ordinary size calculations rather than a model-wide constant.

```
1156 static int kv_ring(const Model *m, int n_ctx, int slack) {
1157     n_ctx = (n_ctx + 7) / 8 * 8;
1158     const int ring = (m->n_swa + slack + 8 + 7) / 8 * 8;
1159     return m->n_swa > 0 && ring < n_ctx ? ring : n_ctx;
1160 }
1161
1162 static int kv_rows(const Layer *L, int n_ctx, int ring) { return L->swa
     ? ring : (n_ctx + 7) / 8 * 8; }
1163
1164 // bytes of the keys (or the values) of one layer: the values, then (8-
     bit) the scales
1165 static size_t kv_layer_bytes(const Layer *L, int rows) {
```

PREPRINT

```
1166     const size_t n = (size_t)rows * L->n_head_kv * L->hd;
1167     return kv_q8 ? n + n / 32 * sizeof(f16) : n * sizeof(f16);
1168 }
1169
1170 static size_t kv_bytes(const Model *m, int n_ctx, int slack) {  // the
        KV cache of a sequence of n_ctx positions
1171     const int ring = kv_ring(m, n_ctx, slack);
1172     size_t total = 0;
1173     for (int l = 0; l < m->n_layer; l++) {
1174         const Layer *L = &m->layers[l];
1175         if (L->kv_src == l) total += 2 * kv_layer_bytes(L, kv_rows(L,
        n_ctx, ring));
1176     }
1177     return total;
1178 }
```

Listing 11.3: `gemma4.c:1156–1178`

How can the context grow without moving the ring?

The state starts with room for a requested context. Chat can need more room later, so the engine allocates a larger state and copies the existing cache rows. The global arrays grow with the context; the sliding rings keep their original row count. The operation synchronizes with the GPU before copying because an earlier attention operation may still read the old allocation.

There is an important constraint: growth cannot change the ring length. The ring is based on the final context allocation and the reserved slack. If a proposed larger context would alter that length, the source stops with “the ring would change” instead of copying wrapped rows into a different layout. It is safer to report the unsupported growth than to silently reinterpret ring indices.

The growth code copies owner rows after synchronizing with the GPU, then carries the sequence counters into the new state (`gemma4.c:1203--1218`).

A rollback also has a memory condition. If the cache still contains every position the retained prefix needs, the prefix remains usable. If the ring has overwritten one of the positions needed by that prefix, the engine cannot restore its old contents from elsewhere; it must restart from position zero. The short check in `state_keep` expresses that boundary. Its `n_high` field records how far writing has reached, which is different from the current `n_past` after an edit or undo.

How do reads walk through a wrapped ring?

Physical rows are not stored in chronological order once the ring wraps. At position 8 in a five-row toy, logical positions 4, 5, 6, 7, and 8 live in physical rows 4, 0, 1, 2, and 3, respectively. Attention therefore walks logical positions in order and maps each one to its physical row. It cannot just read rows 0 through 4 as if they were the oldest-to-newest sequence. The CPU reference makes the mapping at each position explicit. The Metal kernel starts at the first visible absolute position and advances the physical index with wraparound.

This separation between logical position and physical row also keeps RoPE correct. Position 8’s key is rotated for position 8 before it is stored in row 3. It does not become a key for position 3 when it replaces the old row. The row is merely a slot; the absolute position gives the vector its meaning. A cache dump that lists only physical row numbers loses this distinction, so a useful diagnostic prints both the absolute position and the selected row.

With a window of four ending at position 8 in a six-row ring, attention reads positions 5, 6, 7, and 8 from rows 5, 0, 1, and 2. Position 8 maps to row 2, replacing position 2. In the five-row, five-position-window example above, position 8 instead replaces position 3. Ring capacity must be at least window plus the in-flight write slack; the visible window itself must never alias a row being used for that same attention call.

PREPRINT

In the engine, `kv_ring` rounds the request to an eight-row boundary and includes the sliding window, slack, and an extra safety margin. If the proposed ring would be as large as the full context, the function uses the context-sized allocation instead. That avoids wasting time treating a small cache as a circular queue when no wrap can happen. It also means the same indexing code can use a `ring` value equal to the full context for short sequences.

What exactly does 8-bit mean here?

The default cache represents each group of 32 floating-point values by signed 8-bit integers and one half-precision scale. For a block with largest absolute value 1.27, the scale is 1.27 divided by 127, or 0.01. Values 0.37, -0.82, and 1.27 become approximately 37, -82, and 127; reading them multiplies each integer by 0.01. This tiny example shows the shared scale: nearby values use the block's same unit, and rounding can alter them.

The implementation stores the scale rounded to fp16, then converts that stored half back to float for quantization and retrieval. Values are rounded to the nearest integer and clamped to -127 through 127. It does not use -128, so the positive and negative limits are symmetric. With no scale pointer, the same API takes the fp16 path instead: each value is converted directly to half and later expanded back to float.

The C excerpt below is the reference store. It covers both storage modes, and its indexing makes clear that the scale table has one entry for each 32-value block in each row. The retrieval routine uses that same block index and scale. The Metal store follows the same data layout: one thread lane owns one coordinate in the 32-value block, and lane zero writes the shared scale. The attention kernels read that representation later.

```
1261 static void kv_put(void *data, f16 *scales, size_t row, int nkv, int off
     , const float *x, int n) {
1262     if (!scales) {
1263         for (int i = 0; i < n; i++) ((f16 *)data)[row * nkv + off + i] =
     float_to_half(x[i]);
1264         return;
1265     }
1266     int8_t *q = (int8_t *)data + row * nkv + off;
1267     for (int b = 0; b < n / 32; b++) {
1268         float amax = 0;
1269         for (int j = 0; j < 32; j++) amax = fmaxf(amax, fabsf(x[32 * b +
     j]));
1270         const f16 h = float_to_half(amax / 127);
1271         const float d = f16_table[h];
1272         scales[row * (nkv / 32) + off / 32 + b] = h;
1273         for (int j = 0; j < 32; j++) {
1274             const float r = d ? rintf(x[32 * b + j] / d) : 0;
1275             q[32 * b + j] = (int8_t)(r > 127 ? 127 : r < -127 ? -127 : r
     );
1276         }
1277     }
1278 }
1279
1280 // values [off, off + n) of row `row` of a cache, as floats
```

Listing 11.4: `gemma4.c:1261–1280`

The 8-bit representation is compact, not lossless. Every key affects a score, and every value affects a weighted sum. A small per-value rounding error can therefore change a score, the softmax weights, or the final mixture. The next chapter's fast attention kernels must reconstruct this representation as they stream keys and values.

What do cache totals leave out?

The byte totals in this chapter count only the persistent K and V arrays, together with the Q8 scale arrays when applicable. They do not include the model weights, temporary query and attention-output buffers, per-layer activation storage, tokenizer and chat entries, media rows, allocator alignment, or the Metal bookkeeping that wraps shared memory. This is why a process memory measurement is larger than the cache number. It is also why the README separately distinguishes the cache footprint from total process memory and the system's wired weights.

A memory number needs a boundary. "The 32K cache is 580 MB" describes the cache allocation; it does not mean the whole process uses 580 MB. The engine also maps and wires model weights for the GPU. Conversely, the file size is not the cache size: the K/V projection weights in the GGUF are matrices used to produce new rows, while the cache rows are the projected results retained for attention. The projection weights are shared by all positions; the cache values are different at every position.

The W0 calculator makes a deliberately narrow accounting boundary so each byte has an owner. The per-position number is the extra logical K/V content for a new position before the sliding rings fill. The 4K and 32K totals add the ring rows for sliding owners and the full context rows for global owners. The README's process-memory result adds workspace around that cache, and its "wired" number describes a different portion of the memory map. Put the quantity and unit beside any reported total.

The separate scale arrays matter to that accounting. In fp16, keys and values each occupy two bytes per coordinate. In Q8, the coordinate bytes still occupy one byte each, but every 32 coordinates also need a half scale. The two scales for a K/V pair are stored independently. If a cache is split into per-head allocations, alignment padding could add some more bytes; the engine's contiguous allocation and row-width multiple of 32 avoid that extra per-head scale padding in the simple arithmetic here.

Why is the query not part of the cache?

The current query is the question being asked at the current position. It changes for every token and every layer, so the engine computes it when needed and keeps it only in working buffers. Old queries do not have to be kept: later positions do not ask what an earlier position asked. They ask their own new questions against the old keys. The cache therefore stores the reusable side of attention, the keys and values, rather than every intermediate vector from the layer.

The key and value themselves must be retained in the representation the later attention operation expects. A key has already been normalized and position-rotated; recalculating it from a hidden state would require preserving that hidden state and replaying the projection and preparation. A value carries the per-head contents to combine, and its projection is also tied to the earlier layer input. Keeping the final K/V vectors is a smaller, direct record of the work attention needs.

This is a design choice about what to recompute. In principle, an engine could keep hidden states and project old K/V vectors again later, or keep some compressed representation and expand it before attention. Both alternatives trade cache bytes for extra computation and traffic. This engine instead stores fp16 or block-scaled int8 K/V rows. It does not retain each layer's full hidden state as the attention history. The global rows are often the larger memory cost because they must remain available for every earlier position.

Why is a small cache still read so often?

During prefill, many new positions are processed in one call. Each new key and value is written once, while each query position reads its allowed earlier keys and values. In decode, a new token contributes a new row, then the current query reads the history. The attention code streams K to form scores and V to form the weighted sum. For a global layer the amount read grows with history length. For a sliding layer the read window stays bounded even though absolute positions keep increasing.

The cache reduces repeated projection work, but it does not make attention free. At a global layer, each generated token revisits all earlier K/V rows. That is why long-context decode has a memory traffic

PREPRINT

component that grows with context, even though the model weights are also read every token. The faster attention kernels in Chapter 12 divide this scan into chunks and combine their partial results. They do not change which K/V rows exist or which positions are legal.

There are two very different counts worth keeping separate. The amount stored is the physical rows times the row width times the bytes per value, including scale metadata. The amount read by an attention operation depends on its visible positions, query heads, and the kernel's access strategy. A ring may store a little more than the nominal 512-position window because it reserves slack, but the attention mask still limits which positions affect the answer. Conversely, the global cache stores every position and global attention can read them all.

At short contexts, a 32-value block's scale overhead is small compared with the cache as a whole, but the Q8 representation can improve traffic because each value consumes roughly half the fp16 bytes. At long contexts, that traffic reduction can help, though the decode kernel still performs scale loads, integer conversion, score work, value accumulation, and chunk merging. The README's speed comparisons depend on those kernels and on GPU conditions; the bytes-per-value arithmetic by itself is not a token-per-second promise.

What can a toy cache check prove?

The calculator is a bookkeeping check. It can catch a missing factor of two for K and V, accidentally charging shared layers twice, forgetting the scale words, or giving sliding layers full-context rows. It cannot tell whether a model's K/V projection is right, whether RoPE used the correct absolute position, or whether attention masks the correct positions. Those checks need a model-aware path and belong to the later correctness chapters.

The ring simulation is focused. A position modulo the ring length must select the expected row. After enough writes, the retained logical suffix must still be present. With five rows, position 8 overwrites the row that held position 3. The toy demonstrates storage reuse, not visibility masking. A cache correctness test should separately enumerate visible absolute positions and check each expected row before reading its contents.

The block round trip checks one more boundary. The scale used for quantization must be the same half-precision scale later used for retrieval. With the sample maximum 1.27, the values 0.37, -0.82, 1.27, and 0.374 map to 37, -82, 127, and 37. Their reconstructed values are 0.37, -0.82, 1.27, and 0.37. The fourth entry demonstrates ordinary rounding; the others make the scale's sign and maximum easy to inspect. This is not a statistical accuracy test. A model-level comparison must pass many real key and value vectors through the attention calculation and compare the resulting distributions.

A useful progression for extending the toy is to check an all-zero block, where the scale is zero and every stored code must remain zero; a negative-only block, where the scale still uses the largest magnitude; and a block with an outlier, where smaller values lose more precision. Then test row boundaries and two consecutive rows so the scale index does not leak from one row into the next. Keep each test deterministic: a random block is useful later, but fixed values make a rounding bug reproducible.

What does prefix reuse actually compare?

Chat stores a list of entries that describe the rendered sequence. A plain entry can be a token ID; a media entry can point to a row of encoded image or audio data. When the user submits another turn, the template is rendered again and the new entry list is compared with the cached list from the beginning. The common prefix length is the number of consecutive entries that still match. Only entries after that boundary need a new forward pass.

Take the old rendered history `[system, user-A, model-A, user-B, open-model]`. If only the last user text changes, reuse reaches `model-A`; the new user and open marker must run. If the system instruction changes, reuse stops at the first entry. The cache is valid only for the exact prompt and settings; similar-looking token sequences can have different states.

PREPRINT

The implementation compares entries with `same_entry`, and it leaves the last matching position to be run again when logits are needed for generation. It then asks `state_keep` whether the sliding ring still retains every row the candidate prefix depends on. These checks answer separate questions: did the prompt remain the same, and does physical storage still contain the rows required to continue? A true answer to only one is not enough. The loop below first finds matching entries, then checks whether the ring can preserve that prefix. It also backs up to the start of a media part that must be run whole, so a cache is not resumed in the middle of a multi-row image or audio entry. The last matching position may be run again when generation needs its logits.

```
4260 // Runs the entries e[0..n) into the KV cache, reusing the cached prefix (room for N more positions after them);
4261 // with logits, the last position is always run, for its logits. Returns how many were reused.
4262 static int chat_run(Chat *c, const Entry *e, int n, int N, int logits) {
4263     if (n + N + 1 > c->st.n_ctx && c->st.n_ctx < c->n_ctx) {  // grow the cache: to twice its size, or to what it needs
4264         int want = 2 * c->st.n_ctx > n + N + 1 ? 2 * c->st.n_ctx : n + N + 1;
4265         state_grow(c->m, &c->st, want < c->n_ctx ? want : c->n_ctx, c->slack);
4266     }
4267     // reuse the cached prefix; a whole part is run whole, and the last position is always run (for its logits)
4268     int p = 0;
4269     while (p < n && p < c->n_cached && same_entry(e[p], c->cached[p])) p++;
4270     if (p == n && !logits) return n;
4271     if (p == n) p--;
4272     if (e[p].rows && e[p].whole) p -= e[p].row;
4273     p = state_keep(c->m, &c->st, p);  // (0 if the sliding-window cache no longer holds what p needs)
4274     const int reused = p;
4275     c->st.n_past = p;
4276     const int E = c->m->n_embd;
4277     for (int i = p; i < n;) {
4278         int T = 0, media = 0;
```

Listing 11.5: `gemma4.c:4260–4278`

This is why ‘/undo‘ can often be cheap but sometimes restarts. If it removes only the latest exchange and the cache still holds the surviving prefix, the next turn reuses that state. If the ring has advanced far enough that a needed old window position was overwritten, the cache cannot reconstruct it from the visible conversation entries; it must prefill again. Global rows remain available, but one invalid sliding layer is enough to reject continuation.

Does a chat turn reuse the same cache?

Chapter 16 follows the complete chat loop; here the key rule is that a cache row remains useful only while the rendered entry at that position is unchanged. The detailed comparison below shows how the engine finds that boundary. The final matching position is handled carefully because generation needs logits for the next token, and a changed special marker or media entry can invalidate a position even when the visible prose looks similar.

The prefix comparison belongs to Chapter 16’s chat loop, but it is worth connecting it to the physical cache. A cache is not a general-purpose transcript store. It is valid only for the exact sequence and settings that produced its rows. A new user message normally leaves the earlier exchange untouched, so the old prefix is reusable. ‘/undo‘ keeps the longest prefix that remains valid. ‘/clear‘ discards it. If the sliding ring has already overwritten a position needed by the prefix, `state_keep` says to start over.

Media makes the boundary more visible. The chat sequence can contain image or audio rows, not just token IDs. Before old media rows are freed, the cache is cut at the first position that depended on them. Otherwise a newly allocated media buffer could reuse the same address, while the cache still appears to describe the old content. This is a documented safeguard in the README's media discussion, not a generic claim about every cache implementation.

What happens when a state grows?

Suppose a chat begins with a context allocation of 4,096 positions and later asks for 8,192. A new state is first allocated at the larger size. The engine waits for outstanding GPU work because the GPU may still be reading the old cache. It copies each owner layer's existing K and V arrays, copies the scale regions when using Q8, releases the old block, and transfers the sequence counters. The new state has more rows for the global owners. Sliding ring rows are copied at the same size and retain the same modulo mapping.

This is not the same as appending a larger buffer in place. Shared Metal buffers may move, and every layer pointer would then need to be reset. Allocating a fresh block makes the new layout internally consistent before old storage is released. Synchronization is essential: copying while the GPU reads the old block would race even though the CPU code itself runs sequentially. The copy cost occurs at growth boundaries, not on every token.

The ring-length check is stricter than a simple capacity check. If the old state had a 1,600-row ring and the larger requested context still computes a 1,600-row ring, sliding data has the same physical meaning and can be copied as-is. If the new context caused the ring to become 2,000 rows, then position 1,700 would map to a different row under the new modulus. Copying bytes alone would not repair that mapping. The engine rejects this case. The chat loop chooses its initial context and reply slack so ordinary growth can preserve the ring.

There is another distinction between allocation capacity and valid contents. A fresh larger state may have space for thousands of positions, but only positions already processed belong to the cached conversation. The counters `n_past` and `n_high` tell the control flow where the current prefix ends and how far the ring has been written. Old bytes can remain in unused rows after a rollback; that does not make them valid history. State metadata, logical positions, and physical data must agree.

A one-token trace through cache storage

Imagine the engine is about to process absolute position 700. In each layer it computes a query and, for an owner layer, new K and V vectors. A sliding owner writes its prepared vectors to row 700 modulo the ring length. A global owner writes them to row 700 of its full-history arrays. A shared layer computes its query but gets K and V pointers from its owner; it has no new K/V values to write. The attention step reads the current layer's allowed positions, applies the scores to its values, and produces the output vector for that layer.

Now imagine the token is in a prompt prefill batch of 16 new positions rather than in decode. The cache still receives one new K/V row per position for every owner. The ring must have space for the visible window, the rows required during the batch, and reply slack so that a future write does not overwrite an in-flight read. This is why context capacity alone is not enough to pick the ring size. A 512-position visibility rule describes what attention may read, while the physical ring also needs enough staging room for how the engine schedules writes and reads.

When the same token is in a shared layer, only the query side is new. Chapter 10's preparation code has a `q_only` path for this case: it skips normalization, rotation, and cache writes for keys and values. The attention operation still reads the owner rows in the same logical position range. A trace across one owner and one shared consumer is a good way to catch an incorrect source pointer or a mistaken assumption that every transformer layer owns a K/V array.

The cache's state counters do not change the mathematical position. `n_past` identifies where the next segment starts; `T` identifies how many positions are processed in the call. For token `t` of that call, its logical position is `n_past + t`; that position determines both the RoPE angles and the physical row.

PREPRINT

A ring wrap changes the row, never the logical position. A prefill call can therefore write positions 700 through 715 to wrapped rows while preserving their distinct positional rotations.

How can a reader audit one cache claim?

The explanation can be checked as a short chain rather than trusted as a single comment. Start with the W1 directory output: it identifies the projection widths in the selected model file and reports the shared-KV count. Next inspect the loader's per-layer setup, where head widths and KV-head counts are read and a compatible earlier source is selected. That tells us which layers produce rows and which layers point to them.

Then inspect the allocation functions. `kv_ring` determines a physical row count for sliding layers. `kv_rows` chooses between that ring and the full context. `kv_layer_bytes` accounts for the row width and, in Q8 mode, scale storage. `kv_bytes` loops through the model but adds memory only when the layer is its own KV source. These functions establish the byte count independently of the attention arithmetic.

The state constructor then shows where the counted bytes go. It creates one allocation, assigns K/V pointers for owners, attaches scale pointers in Q8 mode, and aliases the pointers for shared layers. Next trace `attn_prep`: the absolute position determines the ring row, and the prepared vectors are stored. Finally, the attention code reads rows for the visible logical positions. A storage claim is not complete until the writer and reader agree on the row and scale indexes.

For chat reuse, follow a different chain: the renderer produces entries; `chat_run` compares them with cached entries; `state_keep` checks whether the physical ring still contains the needed prefix; then only the unmatched suffix is run. Media adds an ownership boundary because rows can be freed and their addresses reused. This chain explains why identical-looking text does not guarantee a valid cache if token IDs, media rows, markers, or positions differ.

This audit is also a practical debugging order. If a W0 byte count is wrong, check owner totals and row width before attention. If fp16 and Q8 disagree more than expected, check scale rounding and scale indexing. If an output diverges only after wraparound, print logical positions alongside physical rows. If a chat edit recomputes too much or too little, log the common-entry prefix and the result from `state_keep`. Each diagnostic narrows the question without requiring a new model inference run.

The W0 experiment: count bytes and watch a ring

Compile and run the calculator from the book root:

```
cc -O2 -o code/build/ch11_kv_size \
  code/examples/ch11_kv_size.c -lm
code/build/ch11_kv_size
```

The program is a small cache-size calculator plus a ring simulator; it prints aggregate fp16 and 8-bit totals at 4K and 32K using a 1,600-row sliding ring. Give it the E4B owner counts and dimensions above, then calculate one-token storage for fp16 and 8-bit blocks. In a separate five-row ring, write positions 0 through 8 and print the row contents after each write. Check that each write lands at position modulo five, and that the most recent five positions remain available.

Expected arithmetic for the model layout: 28,672 K/V values per position; fp16 uses 57,344 bytes before alignment; 8-bit blocks use 30,464 bytes. The calculator reports 132,644,864 fp16 bytes and 70,467,584 Q8 bytes at 4,096 positions; at 32,768 it reports 602,406,912 fp16 bytes and 320,028,672 Q8 bytes. The README reports 133 MB and 71 MB at 4K, and 580 MB and 308 MB at 32K. Treat this as a cross-check, not exact equality: the calculator fixes the ring at exactly 1,600 rows and prints bytes, while the README rounds measured memory using its own displayed units and the engine sizes the ring from each run's slack. For the toy ring, after position 7 the rows hold 5, 6, 7, 3, 4, and after position 8 they hold 5, 6, 7, 8, 4. This is a deterministic W0 check with no model weights. It does not reproduce the rounded allocation slack or the engine's asynchronous prefill; those are visible in the source. The captured run on 2026-10-22 began:

PREPRINT

```
E4B owner values per position: 28672
fp16 bytes per position: 57344
8-bit bytes per position: 30464
sample scale: 0.01; q: 37 -82 127 37; back: 0.37 -0.82 1.27 0.37
context fp16_bytes q8_bytes
4096 132644864 70467584
32768 602406912 320028672
...
after 7: 5 6 7 3 4
after 8: 5 6 7 8 4
```

The full nine-position trace is saved in `code/expected/ch11-w0.txt`.

The W1 check: confirm dimensions without reading weights

The GGUF directory probe maps the model and reads its header, metadata, and tensor directory; it does not read the tensor payload. Run `python3 code/gguf_probe.py "$MODEL"`. The captured 2026-10-22 output reports GGUF v3, 720 tensors, 56 metadata entries, a 4,715,416,704-byte file, and 18 shared-KV layers. The selected projection records are:

Tensor	Type	Shape
`blk.0.attn_k.weight`	IQ4_XS	[2560, 512]
`blk.0.attn_v.weight`	Q5_K	[2560, 512]
`blk.5.attn_k.weight`	IQ4_XS	[2560, 1024]
`blk.5.attn_v.weight`	Q5_K	[2560, 1024]

The second dimension of a K or V tensor gives its output width: 512 for the sliding layout and 1,024 for the global layout. Together with the loader's per-layer KV-head count and head sizes, that confirms the 2×256 and 2×512 layouts used in the arithmetic above. The probe reports `payload bytes read: 0`. This is W1, not an inference run. The captured header and directory are in `code/expected/ch11-w1.txt`.

What does 8-bit cost in accuracy?

The README compares this engine's 8-bit cache against its fp16-cache run on the same evaluation text. These are historical measurements recorded in the engine README, not recaptured during this chapter. The cache-accuracy table does not state its original run date; its context is the README's E4B IQ4_XS evaluation against an fp16-cache run of the same text. On the IQ4_XS E4B setup, mean KL divergence is 0.000217 at chunk size 512, 0.000179 at 2,048, and 0.000162 for one-token decode with the 8-bit cache. The fp16-cache comparison reports 0.000012, 0.000009, and 0.000009 for those same paths. The 8-bit runs retained the same top token 99.2–99.9% of the time. At 4,096 context, the reported cache memory falls from 133 MB to 71 MB.

These are comparisons of probability distributions, not claims that 0.02% of words are wrong. KL divergence measures the change in the next-token distribution across the evaluated positions; same-top-token counts only whether the largest-probability token stayed the same. A changed distribution can preserve the top token, and a small change can flip it when two candidates are close. The accuracy cost is measurable even if a short generated paragraph looks identical.

The README reports the default as a trade: about half the cache bytes, with a larger divergence than fp16. It also reports a larger 8-bit-cache divergence for E2B—about 0.0012 versus 0.000020 for fp16, around sixty times the E2B fp16 result. The README suggests the one-KV-head layout may make each rounding error affect more query heads; it labels this as an interpretation rather than a tested cause. This chapter keeps the same distinction: the measurements are recorded; the explanation remains a hypothesis.

PREPRINT

Pitfalls from the project notes

Pitfall

A ring size cannot be changed during growth. The symptom is an explicit stop in `state_grow`: "the ring would change." The cache's sliding rows are indexed by position modulo the ring length, so copying them into a different ring length would give old rows new meanings. The fix is to choose context and reply slack before creating the state; growth may enlarge global storage while preserving the ring. Source: `gemma4.c:1203--1206`.

Pitfall

Freed media can reuse an address. If a chat frees media rows while retaining cache entries that refer to them, a later allocation may reuse the same address and make stale cache state appear current. The documented fix is to cut the cache at the old media start before freeing the rows. Source: snapshot `README.md:153--155`.

Pitfall

Eight-bit cache quality depends on the model. E4B's measured divergence rises from about 0.000012 to about 0.0002 in the c=512 comparison; the E2B README comparison rises from 0.000020 to 0.0012. The symptom is a larger distribution change even though the cache is smaller. The default chooses memory and speed for the E4B; use `--kv f16` when the smaller numerical difference matters more. The README calls the proposed E2B explanation plausible, not proven. Source: snapshot `README.md:242--255` and `ENGINE-MATH.md:1315`.

Exercises

Exercises

1. **Read.** For one sliding owner, count K and V values at one position from two KV heads and a 256-value head. Repeat for a global owner with 512-value heads.
2. **Measure.** Calculate the E4B fp16 and 8-bit bytes per newly retained position from the owner counts. Then explain why this is not the same as multiplying by the full context for every layer.
3. **Read.** Trace a shared layer's `kv_src` into `state_new`. Which pointers are aliased, and which projections are not loaded for that layer?
4. **Change.** Modify the W0 ring toy to six rows and write through position 9. Print the logical position represented by every physical row.
5. **Read.** From `state_keep`, identify when it can retain a prefix and when it must return zero. Use `n_high`=1,300, ring 1,600, and prefixes 700 and 900 as test cases.
6. **Measure.** Compare the README's E4B 8-bit and fp16 mean KLD at c=512. Explain why the same-top-token percentage cannot replace KLD as an accuracy measure.

Rebuild checkpoint

Rebuild checkpoint

Milestone m5b: a ring and a quantised cache row. Implement `steps/m5b_kvcache.c`: allocate a small ring, write a chosen position into row `position % ring`, store one 32-value K or V block in fp16 and in int8-plus-half-scale form, and retrieve it. Compare the 8-bit values with `tools/check_math kv8` on the same synthetic block. Then test wraparound by writing more positions than the ring can hold and verify the visible window. This checkpoint stops before layer sharing, GPU synchronization, and the model's full per-layer dimensions.

PREPRINT

Chapter 12

Attention, step 2: the fast kernels

Part III: The model

Chapter 10 introduced the query, keys, values, and the softmax that turns scores into weights. Chapter 11 showed where the engine stores those keys and values. Now we follow the same calculation into the GPU kernels: one path for a single decode query, another for a block of prefill queries.

The question

How does the engine attend over a long sequence without turning one query into one serial walk across every key?

Reading map: 24 pages

This first pass assigns the outline's full estimate before filling the chapter. Targets count the figure, source excerpts, experiment output, exercises, hints, and checkpoint.

Subsection	Pages
Question, recap, and the two attention workloads	2
Online softmax idea and running-state picture	3
E4B head dimensions and worked decode step	3
Q8 cache reads, scale metadata, and byte trick	3
Ring indices, logical positions, and visible windows	2
Decode threadgroups, SIMD groups, chunks, and merge	3
Prefill query-block tiling and shared-memory reuse	3
CPU reference, Metal kernels, and host dispatch listings	3
W0 softmax experiment and W2 capture procedure	1
Documented dead ends, exercises, hints, and m5c	1
Total	**24**

The idea

Why does the same operation need two kernels?

For one generated token, there is one query per head and a history of keys and values. The useful parallelism is over the history: many threadgroups can read disjoint chunks of old positions. During prefill, a call has many query positions. The useful parallelism is over blocks of those queries and shared key/value tiles. One query has little matrix work; many queries can reuse a loaded tile.

Both paths compute the same result. For a query, each visible key gets a dot-product score. Softmax turns those scores into weights, and the output is the weighted sum of the corresponding values. The CPU implementation computes all scores, finds their maximum, exponentiates the shifted scores, sums

PREPRINT

them, then reads values. That is easy to follow, but it stores a score for every position and makes separate passes over keys and values.

The GPU keeps a compact summary while it streams. The state consists of a running largest score, a running sum of exponentials relative to that maximum, and a vector containing the weighted values on the same scale. When a later score establishes a new maximum, the previous sum and vector are rescaled before the new item joins them. This is the online-softmax update from Chapter 10, now applied inside the attention kernel rather than to a small example on paper.

How can one pass remember an exact softmax?

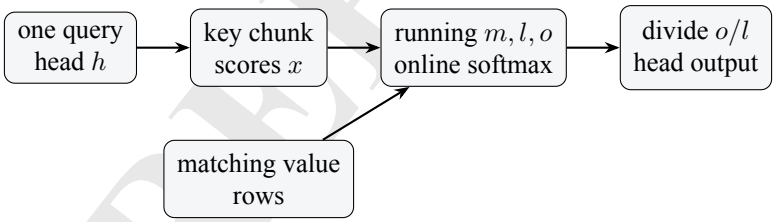
Suppose three scores are 2, 1, and 3. After seeing 2, the current maximum is 2 and the scaled sum is 1. After score 1, the state is still on the same scale and the sum is $1 + e^{-1}$. When score 3 arrives, the reference scale moves to 3: the old terms are multiplied by e^{-1}, and the new term contributes 1. The result is the same normalized distribution as if all three scores had been held in an array and normalized together.

For a stream of scores x, the kernel keeps a maximum m, a denominator l, and a value numerator vector o. Each new score changes those three pieces together:

$$m' = \max(m, x), \qquad a = e^{m-m'}, \quad b = e^{x-m'}, l' = al + b, \qquad o' = ao + bv.$$

For the numbers above, the final maximum is 3, the denominator is $e^{-1} + e^{-2} + 1$, and the numerator is $e^{-1}v_2 + e^{-2}v_1 + v_3$. Dividing the numerator by the denominator gives the ordinary softmax-weighted value. This works because all terms are expressed relative to one common maximum after each update.

The kernel does not need a score array. It needs the current key to make one score, then the value paired with that key to update the numerator. The keys and values are therefore streamed once in logical order for a single query. The numerator and denominator are not probabilities yet; they are an unnormalized representation that becomes an output only after division.



A decode threadgroup owns one query head and one contiguous chunk of positions. It writes its partial numerator, maximum, and denominator; a later kernel combines those partial states. Prefill keeps the same mathematical summary but applies it to a small row tile of queries.

Why split decode into chunks if the scan is one pass?

A single GPU group scanning 32,000 positions would leave most of the GPU idle. Decode therefore splits the visible history into chunks. Each chunk can compute its own online state independently. The merge kernel finds the largest chunk maximum and rescales every chunk's numerator and denominator to that shared scale before adding them.

The chunk state has hd numerator values plus two scalars: maximum and sum. If a head has 256 dimensions, that is 258 floats per chunk. A chunk of 32 keys is a useful starting size, but a fixed 32-key chunk would create 1,024 partials at 32K history. The extra partial buffer and its merge traffic become large. The host raises the chunk length to keep the count at most 64 for Q8 cache and 16 for fp16 in this source version. Those are caps from recorded tuning, not universal GPU constants.

The merge is mathematically the same update as the kernel's online state. If chunk A has maximum 2, denominator l_A, and numerator o_A, and chunk B has maximum 3, the combined maximum is 3. Chunk A is multiplied by e^{-1}; chunk B remains on its scale. Add both adjusted denominators and numerators, then

PREPRINT

divide once. This permits chunks to finish in any order without changing the result beyond floating-point rounding.

What dimensions does one E4B decode head carry?

The language model has 8 query heads and 2 KV heads. Four query heads share each KV head. Sliding layers use a 256-value head; global layers use a 512-value head. Consider one global query head, number 4, which is the first query head assigned to KV head 1. Its query has 512 half values. Each visible position supplies a 512-value key and a 512-value value for KV head 1. The score is one dot product across 512 dimensions, and the output is 512 weighted sums.

Take position 32,000 after the new token is stored. If this is a global layer, the visible absolute positions are 0 through 32,000: 32,001 keys. With 8-bit cache, one K or V vector has 512 bytes plus 16 half scales (32 values per scale), or 544 physical bytes. The selected head's K and V read is 1,088 bytes per position before accounting for the other three query heads sharing the KV rows and the kernel's actual cache behavior. The query itself is reused across the history.

The host selects 64 chunks maximum for Q8. A 32-key starting chunk would exceed that, so the chunk size rounds up to a multiple of 32 large enough for no more than 64 chunks. At 32,001 positions the rounded chunk length is 512, which gives 63 chunks. The group assigned to query head 4 processes its chunk's positions; the combine stage reads the 63 partials for that head. This is a worked source-level dispatch calculation, not a timing claim.

For a sliding layer, the same decode position sees only the last 512 positions, subject to the model's window definition. The chunking logic computes the visible span first, so it does not create work for the entire conversation. The ring length may be larger than the visible window to allow safe writes, but those extra slack rows are not attention inputs.

Why does the running maximum protect the sum?

A direct exponential of a large positive score can overflow, and exponentials of very negative scores can underflow to zero. Softmax does not need either extreme value on its own: subtracting the largest score leaves zero for the winner and non-positive values for every other entry. The online state keeps exactly that stable reference. If a new maximum appears, the previous terms are multiplied by a factor no greater than one. The denominator therefore remains on a scale where its largest contribution is one.

A short three-score trace makes the rescaling visible. Let the scores arrive as 2, 1, then 3. The first state is $m = 2, l = 1$. The second leaves $m = 2$ and adds e^{-1}, so $l = 1 + e^{-1}$. The third changes m to 3, multiplies that old sum by e^{-1}, then adds one. The denominator is $e^{-1} + e^{-2} + 1$. If the third score had been 1.5 instead, the maximum would stay 2; the new term would be $e^{-0.5}$ and no old term would need rescaling. Arrival order changes intermediate states, but not the exact final sum.

The numerator follows the same scaling rule coordinate by coordinate. If a value vector has 512 coordinates, every coordinate's old weighted sum gets multiplied by a whenever the maximum changes; the new value contributes b times its coordinate. This is why online attention needs a vector accumulator, not only a scalar normalization state. The accumulator can be held across iterations in lane-local registers, avoiding a score matrix and a full probability vector in memory.

Floating-point results still depend on order. In a serial CPU pass, scores are visited in increasing position order. GPU chunks compute independent summaries, and the merge adds them in chunk order. FP32 addition is not exactly associative, so a correct parallel result may differ in low bits from the CPU reference. The relevant checks are a bounded per-coordinate difference and a model-level distribution comparison when weights are available. A token ID alone is too weak: it may stay constant while many logits shift.

The fp16 values are promoted for accumulation. The online denominator and value numerators in the decode kernel are floats; the chunk combine also works in float and converts only the completed output to half. The flash path uses fp32 matrix accumulators for score and output fragments while storing probability fragments in half. These choices retain speed while avoiding a long sequence of half-precision

PREPRINT

additions. The exact reduction tree and conversion points are part of the numerical behavior and should be recorded if a rewrite changes them.

Which positions are legal at the edges?

For causal attention, a query at absolute position p sees keys from the start through p, inclusive. At $p = 0$ there is one key. At $p = 7$ there are eight. A sliding layer with a 512-position window sees from $\max(0, p - 511)$ through p. The boundary includes the query's own newly prepared key/value row. The CPU helper `first_key` computes the lower bound; the kernel also limits the upper bound for causal calls.

Prefill complicates the upper bound because several query rows are processed together. If the call starts after n_{past} and contains T rows, row t has absolute position $n_{\text{past}} + t$. In causal mode, that row sees no position after itself even if later rows from the same call already have their K/V written. In bidirectional encoder attention, every row can see through the call's final position. Thus having a K/V row physically present does not prove that a query may attend to it.

Ring storage adds a second edge condition. For a chunk of eight keys, the flash kernel assumes those eight rows do not cross the ring boundary; the host requires ring lengths aligned to eight. The tile can then use contiguous matrix loads. When a logical key block would wrap, the starting row calculation and the allocation policy ensure the selected block lies within the ring's valid contiguous segment. The scalar decode loop instead advances one row at a time and explicitly wraps. These are different address strategies for the same logical sequence.

The ring must also protect in-flight writes. Chapter 11 showed why the allocated sliding ring can be larger than the 512 positions visible to a query. Prefill can write a batch while earlier GPU work is queued. Slack rows keep the batch from overwriting cache rows still needed by that work. The attention mask continues to enforce the 512-position window; the extra slots are a concurrency and storage property, not a larger model context.

A robust ring test should combine storage and visibility. Fill a ring with a unique tag equal to each logical position. For every query position, enumerate its legal absolute positions, map each through modulo capacity, and verify that the physical slot still contains the expected tag. Then check the causal and sliding bounds separately. A pure modulo test can pass while the kernel reads the wrong window; a mask-only test can pass while ring wrap maps to a stale position.

What makes an attention dispatch decline?

The Metal wrapper returns zero if the requested geometry does not match its assumptions. It checks divisibility of the head width, maximum head width, query-head grouping, and the special decode alignment requirement. A non-eight-aligned ring is treated as a very large non-ring for the vision path; for language-model flash attention, ring alignment is part of the fast-path condition. Q8 is accepted by specialized decode and flash8 paths, but the general fallback reads fp16 and explicitly declines Q8.

These guards form a small decision tree. T=1 selects decode: split visible keys into chunks, dispatch online summaries, then combine. T>1 first checks whether a flash kernel can own the head shape and grouping. It picks fp16 flash for the ordinary cache and flash8 for Q8 when its Gs grouping condition holds. If neither applies, Q8 declines; fp16 uses the general kernel. The CPU wrapper then processes a task per head and 32-query block.

A dispatch counter should record at least the attention family, input precision, head width, T category, and whether the Metal call succeeded. A single global fallback count is harder to interpret: one unsupported audio shape could hide behind many successful language-model dispatches. The record need not be printed for every token; aggregate counts by family and shape are enough to spot a path that silently moved to CPU.

When results are wrong, debug from the outside inward. First compare the host-selected family and parameters with the intended T, head width, KV ratio, ring, and cache mode. Next verify the grid maps each expected head and query tile exactly once. Then check the legal position interval and physical

PREPRINT

row. Finally inspect the numerical update: key scale before or after the signed-byte correction, common maximum for each online state, and value scale paired with its own block. This order separates wrong dispatch and indexing from arithmetic error.

A mismatch that begins exactly at a wrap boundary points to ring addressing. A mismatch that grows with head width suggests a missing dimension partition or a scale-block offset. A mismatch only in Q8 but not fp16 points to byte unpacking, scale addressing, or the 1152 correction. A mismatch that appears only on the final few prefill rows suggests a partial-tile mask or invalid store. These are diagnostic patterns, not substitutes for a fixed comparison test.

How should the W2 comparison be made fair?

The two benchmark commands in the experiment section vary depth while holding the nominal prefill and decode lengths fixed. For a meaningful comparison, use the same prompt tokens, cache precision, build, context limit, and warm/cooldown protocol. Record any fallback counts. If a call silently runs attention on CPU, an end-to-end token rate cannot be used to judge the Metal kernel.

The isolated kbench path answers a narrower question: how long does the attention operation take for a specified head width, history, and cache representation? The end-to-end benchmark includes model weight reads and every layer operation. Both are useful. Kernel time can explain whether an attention change helped; token rate tells whether that change matters amid the full model. Keep their units separate. A microsecond GPU interval is not interchangeable with wall time per generated token.

The README's throughput numbers are a useful historical reference, but hardware state changes the result. Apple silicon may be sharing bandwidth with the display and other work. The idle gate, thermal state, memory pressure, and cooldown all affect a short kernel sweep. Interleave settings when possible rather than running all of A before all of B; this makes gradual thermal drift less likely to favor one version. Report a median and spread from repeated runs instead of the best observation.

For a regression, first verify that the exact same path is selected. Then compare the attention operation in isolation, and only then compare end-to-end generation. If the isolated time improves but token rate does not, the operation may not be the current bottleneck. If token rate falls while attention time is unchanged, check whether the benchmark changed cache mode, context depth, token count, or CPU fallback behavior. That is the difference between measuring a kernel and measuring the engine.

How does an 8-bit cache become a fast dot product?

Chapter 11 described blocks of 32 signed bytes with one half scale. The straightforward kernel loads signed bytes, converts them to floats, multiplies by the query, and applies the scale. In the documented experiment, conversion capped this path at 47 GB/s. The source uses a packed-byte transform to represent the integers as half values without a per-element integer-to-float conversion.

The cache stores signed int8 values directly. The kernel XORs each byte with 0x80, then places that unsigned byte into the low mantissa bits of the half encoding for 1024. Since the XOR maps a signed value q to q+128, the constructed half represents 1152+q. The dot product therefore contains an extra 1152 times the sum of query dimensions. For a 32-value scale block, subtract 1152 times the query sum to recover the signed-byte dot product, then apply the key scale.

For values, the same packed representation is used while accumulating weighted values. The kernel tracks a scalar weighted scale term as well as the vector numerator, then removes the 1152 offset from each output coordinate before writing the partial result. The code uses float accumulation even though cache values and scales are compact. This transformation is delicate: the XOR, packed half construction, query-sum correction, scale address, and output correction must agree.

The README's recorded tuning says the bit-pattern path reached 68 GB/s and reduced a 32K global-layer attention time from 1.52 ms to 1.04 ms. Those are historical source notes, not a fresh benchmark from this chapter. The W2 command below is the place to recapture them on a quiet GPU.

PREPRINT

When does a row number stop meaning a position?

The cache ring maps logical position s to physical row s modulo the ring length. If a ring has 520 rows, logical position 520 reuses physical row 0. Attention still walks logical positions in order. The Metal loop initializes its physical row from the first absolute position modulo the ring, then increments the row and wraps at the ring length while its logical counter increases.

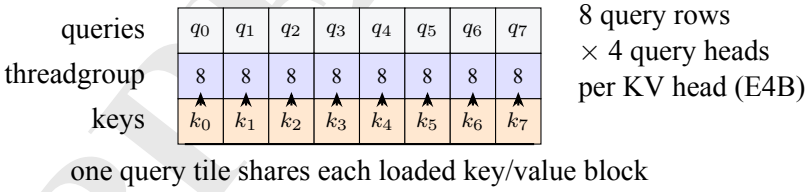
For a sliding query at position 900 with a 512-position window, the first visible position is 389, and positions 389 through 900 are read. The physical sequence starts at row $389 \bmod 520 = 389$ and wraps when it reaches 520. RoPE was applied using absolute position 900 when the current key was prepared; using the slot number instead would silently change the model. A slot is storage, not identity.

A global cache also has a row mapping, but in normal use its ring is effectively the full context, so a context that has not wrapped maps directly to its row. The kernel uses the same address formula for both. The mask decides which logical positions participate; the ring only locates their bytes. Conflating those two jobs can cause either an out-of-window read or a stale row read.

How are prefill queries tiled?

Prefill has multiple query rows, so the kernel tries to reuse key/value reads. The fast-fallback general kernel works on a query block of eight rows. The host sets `ATT_QB` to 32 in the C dispatcher, but the Metal parameter called `qblocks` expresses how many groups of eight queries each threadgroup handles: it chooses $4/G$ when a KV head has fewer than four query heads, and one otherwise. Thus for the common E4B ratio $G = 4$, each group covers one block of eight queries; successive threadgroups cover the call's query rows. The CPU task uses the same 32-query macro as its outer batching unit.

For supported head sizes, the flash-attention kernel assigns eight consecutive queries and all query heads for a KV head to a threadgroup. A 256-wide head has two 128-dimension partitions; a 512-wide head has four. Each SIMD group owns one query-head/128-dimension partition. It loads the eight query rows once, then walks key blocks. A simdgroup matrix multiply accumulates the score tile. The partial score tiles from head-dimension partitions meet in threadgroup memory so every partition can apply a consistent softmax to the complete score.



The flash kernel advances through 32 keys at a time, internally arranged as four 8-key matrix tiles. Every SIMD group runs the online softmax for the scores of its query row and accumulates the matching value rows. The diagram emphasizes the reuse: one block of K and V serves the query tile, then the next block arrives.

What does lazy max save, and what does it cost?

The prefill kernel does not always move its running maximum to the exact maximum of the newest block. It lets the active scale lag until a block maximum exceeds the current maximum by more than 8. While the difference is no greater than 8, the exponentials stay below e^8, which fits the kernel's half precision probability tile. If the threshold is crossed, it rescales the old value accumulators and denominator using the new maximum.

The threshold saves repeated rescaling work across blocks. The source's comment records a measured optimization: directly multiplying each accumulator element by the correction made that version 25 percent slower, while expressing the rescale as a diagonal matrix product performed better. The diagonal

PREPRINT

carries the correction for each query row, and simdgroup matrix multiplication applies it across the output accumulators.

This is a prefill tuning choice. The decode kernel tried a lazy max too, but the per-key branch cost more than the multiplications it avoided. The documented numbers were 1.16 ms with the lazy path versus 1.03 ms without. A numerical shortcut is only useful when it saves more than its control and synchronization costs. The project also tried processing eight keys per decode step to put more loads in flight; the README records no speed gain over one key per step (README:237).

What does a complete E4B decode assignment do?

Follow query head 4 at position 32,000 through a global layer. E4B has 8 query heads and 2 KV heads, so four consecutive query heads share each KV head. Integer division maps query head 4 to KV head 1. Its query has 512 half values. HD is specialized to 512, and each of the 32 SIMD lanes owns HD/32 = 16 dimensions. The SIMD group index chooses the query head inside the KV group; grid X chooses KV head 1, while grid Y chooses a history chunk.

The dispatcher takes the visible interval, positions 0 through 32,000: 32,001 keys. With a Q8 cap of 64 chunks, 32-key chunks would be too numerous. The source rounds the required per-chunk length upward to a multiple of 32. It picks 512 keys, giving 63 chunks. A representative group at chunk 17 starts at logical position 8,704 and ends before 9,216. Its physical ring row begins at 8,704 modulo the ring length; with a 32,768-row global allocation, that is row 8,704.

Each lane loads its 16 query coordinates once. For one key, lane 0 forms products from query coordinates 0–15, lane 1 handles 16–31, and so on. A SIMD reduction adds 32 lane subtotals into the full 512-wide score. The key scale is selected using the row address divided by 32, because one scale covers a block of 32 values. That score updates the chunk maximum and denominator. The corresponding V bytes and V scale update the 16 local output coordinates held by each lane. After 512 positions, the group writes 512 numerator values plus the maximum and denominator.

There are three levels of indexing, and each answers a different question. Logical position chooses the causal/window entry. Physical row chooses the cache slot after ring wrap. Head dimension chooses the byte and scale block inside that row. A useful debug print reports all three: position s, row r, and element offset r*nkv + kvh*hd + d. A byte address alone is hard to audit because it loses which model position the address represents.

For intuition, consider four coordinates from one hypothetical 32-value Q8 scale block. Suppose signed key bytes begin 1, -2, 3, -4 and the matching query values are 0.5, 1, -1, 2. Their partial dot is 0.5 - 2 - 3 - 8 = -12 before multiplying by the block scale. This small arithmetic is only an illustration; the real E4B score sums all 512 coordinates and uses actual cached codes and scales. It shows why scaling follows the integer products and why all lanes must reunite before softmax uses the score.

The chunk output is not yet a normalized head vector. It is the numerator and denominator on a chunk-specific maximum. The merge makes all 63 chunks share one maximum, scales them consistently, sums the 512 coordinates and denominator, and converts the result to half. The next output projection sees the same 512-value head shape as the CPU reference, with differences possible from floating-point reduction order.

Why do the lanes own dimensions instead of positions?

A decode group reduces a dot product across the head, then keeps output dimensions live while processing the matching value vector. Assigning dimensions to lanes lets the lanes reduce their partial dot products with a SIMD operation; each lane then maintains a small numerator slice. HD=512 gives each lane 16 floats; HD=256 gives eight. Function constants tell the compiler these bounds when it specializes the pipeline.

The alternatives assign lanes to keys or make lanes cover several query heads. Both can increase parallelism in one direction and add communication elsewhere. Four query heads share a KV stream, so grouping all four heads inside each lane may look attractive. The documented attempt that gave each

PREPRINT

lane four dimensions across all four query heads needed shuffles and barriers; independent SIMD groups per query head were faster. Reuse at the cache-line level did not repay the extra coordination.

The source computes h = gG + sg, where g is the KV head and G is the query-to-KV ratio. For the common ratio four, one threadgroup exposes one SIMD group per associated query head. Lanes cooperate on that head's dot product and keep its partial value vector. Head ownership and lane ownership are separate: the former selects a vector, the latter partitions its coordinates.

The threadgroup has 32 times G threads: four SIMD groups for four query heads, or 128 threads. Grid Y supplies independent chunks. This distinction helps when reading dispatch sizes: a multidimensional grid can represent head groups and chunks even though each group has a fixed small lane team.

Where does each prefill query land?

The CPU path batches tasks by query head and blocks of `ATT_QB=32` query rows. For task i, the head is i modulo the query-head count and the starting row is 32 times the integer quotient. The end is clamped to T. A worker handles a useful run of rows for one head rather than creating a new task for every token-head pair. Query rows remain distinct even when their K/V source is shared.

Metal's general kernel has separate tile geometry. Its grid uses KV head, query tile, and head-dimension partition. The query tile is eight rows multiplied by qblocks. When there are four query heads per KV head, qblocks is one; with fewer heads, it can grow to cover four blocks of eight rows. The group size is 32 times G times qblocks: one SIMD group per head in a query block. `ATT_QB=32` is the C task batching unit; the hardware tile is an eight-row unit. They are not the same size.

The flash path uses another tile: eight query rows, all associated query heads for a KV head, and one 128-dimension partition per SIMD group. A 512-wide head has four partitions. Higher dimensional heads consume more SIMD groups, limiting how many query heads fit at once. The Q8 flash host derives Gs from head width and grouping, then dispatches groups of Gs query heads. A shape guard can therefore encode a hardware resource limit, not merely a convenient software restriction.

Partial tiles are normal. If T=10, the second eight-query tile has two valid rows. The kernel computes its tile, masks positions outside the call, and checks the final query index before writing. A causal mask also applies per query: row t cannot read a key later than its own absolute position. In a bidirectional vision call, the upper bound is the final position in the call. Sliding layers add a lower bound. These are legal-position rules; tile bounds describe which lanes do work.

Why reuse K/V through the core cache?

Several query heads for one KV head read the same key and value rows. The prefill kernel schedules their SIMD groups in one threadgroup so the core can serve repeated loads from its cache. It does not explicitly copy every K/V row into threadgroup memory. That copy would add instructions and barriers and may cost more than cache reuse as the groups advance through blocks.

In flash attention, the query tile stays in simdgroup matrix registers while a key block is multiplied against it. For eight queries by eight keys, the score tile has 64 entries per query head and dimension partition. Four 8-key sub-blocks make one 32-key step. The output accumulators hold query rows against 128 dimensions. Probability fragments then multiply the corresponding value blocks, so a full score matrix never needs to live in device memory.

The general kernel is a less specialized fallback for compatible fp16 inputs. Its groups cooperate across query rows and head dimensions, while output shapes remain identical. The host returns zero for a Q8 shape it cannot read; the C caller must then run the CPU task. During tuning, count declined Metal calls, since a correct but slow CPU result can masquerade as an attention regression.

The documented E2B issue illustrates this. It had eight query heads per KV head, but the first Q8 prefill kernel accepted at most four. Metal silently declined, so output was correct while a five-second wall interval accumulated 27.7 seconds of CPU work. The fix grouped as many query heads as available matrix partitions permitted. Source: ENGINE-MATH §12 and the Metal host notes. Test a guard with one accepted shape and its nearest rejected neighbor.

PREPRINT

What does the chunk merge cost in bytes?

At HD=512 each query-head/chunk partial has hd+2=514 fp32 values, or 2,056 bytes. With 63 chunks, one head's partials occupy 129,528 bytes. Across 8 heads, that is 1,036,224 bytes, about 0.99 MiB. The combine kernel reads these entries and writes one 512-value half output per head. This is the active buffer size for this example; reusable capacity can be larger because the host retains allocations.

At HD=256 and 16 fp16 chunks, one head uses 258 times four times 16 = 16,512 bytes. Across 8 heads this is 132,096 bytes. Q8 and fp16 have different chunk caps because their decode memory paths have different measured throughput. The cap changes the number of groups, partial-buffer traffic, and combine work.

Fewer chunks are not always better. Too few groups leave the GPU underused while each group scans a long run. Too many make partial writes and merging dominate. The host's caps are a measured middle ground on the recorded machine, not a property of every GPU. A port should sweep chunk counts under correctness checks and report hardware and conditions.

The merge group has 256 workers. Each finds the shared maximum over chunks, then dimension worker d accumulates that coordinate from each chunk using the correction for its local maximum. Worker d handles d, d+256, and so on. With 256 or 512 dimensions each worker writes one or two coordinates. One group owns one query head, so no cross-head reduction is necessary.

Why keep score and argmax fusion separate from attention?

Attention fuses work that consumes intermediate values immediately: it forms query-key scores, normalizes them online, and accumulates values. It does not fuse the final vocabulary scoring. Attention produces per-head vectors that still pass through output projections, residuals, feed-forward blocks, final normalization, and the LM head. Vocabulary logits do not exist until those steps finish.

The final forward pass has two related but distinct jobs. Greedy generation needs the largest logit's ID. The GPU argmax scans 262,144 logits and writes the winning ID directly where the next pass reads it, avoiding full logits readback. Evaluation scoring does more: it can apply soft-cap, compute the maximum and log-sum-exp, calculate the selected token loss, save quantized log-probabilities, and compare against a base distribution for KL statistics. The score kernel fuses these evaluation tasks while logits are resident.

The score kernel includes an argmax reduction within its work: each worker tracks a local maximum and candidate ID after optional soft-cap, then the group reduction chooses the winner. Evaluation therefore does not need a separate argmax launch just to find the top ID. Generation uses the narrower argmax path. These are two output-use fusions around logits; neither changes attention's interface.

The word score names two different calculations. An attention score measures query/key compatibility. A language-model scoring pass examines vocabulary logits and their probability distribution. At 32,001 positions, global attention reads a long history in every relevant layer; final argmax scans one vocabulary vector once per generated token. Similar names do not mean the operations can share a dispatch.

The token ID is written where the next forward call consumes it. Keeping that choice on the GPU avoids making logits host-visible and helps decode remain pipelined. CPU work still prepares some inputs, including per-layer embeddings, so the layer sequence is not one giant kernel. Fusion removes intermediate traffic or launches when producer output is consumed immediately and resources permit both jobs together. Chapter 15 follows the token handoff.

How does the 8-bit prefill kernel load a tile?

The matrix units take half inputs. The 8-bit prefill path constructs the needed half tiles from bytes as it goes. It calculates the transposed score product, keys times transposed queries, so the key rows are the left operand and the Q tile can be loaded transposed once. It also reads the scale associated with each 32-value block and folds it into the generated tile. This avoids materializing an entire fp16 cache copy.

PREPRINT

The host chooses the specialized flash kernel when the head width is a multiple of 128, no greater than 512, the query-to-KV grouping fits the kernel, and the ring is eight-row aligned. The 8-bit path has its own grouping rule because each SIMD group builds byte tiles. If the specialized shape does not match, the host can select the general fp16 path; that path declines Q8 so the CPU fallback or another supported path can handle it. Dispatch guards are part of correctness, not merely performance tuning.

How does the host decision tree affect a whole prefill?

The host gets T and the layer's geometry, not a promise that every attention call has the same shape. A one-token decode call has the highest reuse across history but only one query row. A prompt prefill has many query rows and often enough work to use matrix tiles. A final short prefill chunk can have only a few rows even though it enters the T>1 path. The dispatcher must choose from the dimensions it receives each time.

Call	Cache	Path in this source	Output handoff
$T = 1$	fp16 or Q8	decode chunks, then combine	one half vector per query head
$T > 1$	fp16, supported shape	flash matrix kernel or general attention	one half vector per query row and head
$T > 1$	Q8, supported shape	flash8; otherwise Metal declines	CPU fallback if no Q8 path fits
$T > 1$	Q8, generic kernel only	generic Metal path declines	CPU reference remains correct

This table is a reading guide, not a new dispatcher. The exact shape conditions are in the host listing: head width alignment and maximum, query grouping, ring alignment, and bidirectional mode. The fallback row matters because the CPU implementation supports both cache representations through `kv_get`. The GPU general kernel does not automatically inherit that ability simply because its output formula is the same.

Suppose a prompt call has T=17 and G=4. The CPU task count for one head is one block starting at query 0 and one block starting at query 32; the latter is empty after the clamp, so only the first task is needed. On the Metal flash path, there are three eight-query tiles: rows 0–7, 8–15, and a final tile whose first row is 16. The third tile stores only one valid query. A task decomposition can deliberately pad hardware work, but it must suppress invalid reads and writes. The CPU task decomposition instead strides by `ATT_QB` and clamps its end.

For T=33, CPU work splits into [0,32) and [32,33) per head. Flash attention makes five eight-query tiles, with a final one-query tail. These different grouping choices are not a disagreement in semantics. They are separate ways to batch a logically ordered query set into pieces that match each execution unit.

A good dispatch trace should print the token count, history length, query-head count, KV-head count, and head width, along with cache mode, window, ring, selected pipeline name, grid dimensions, and threads per group. For prefill the query tile is also useful. Without both token count and history length, an `attention_fa` timing at a given head width does not identify how much key history or how many query blocks were processed. The profile label in the source records some of those quantities but not the entire launch contract.

Why is a stable top ID not a sufficient correctness check?

Attention is an internal vector operation. A small error in one head can be diluted by later output projections, or amplified by a close competition among vocabulary logits. Greedy text equality is useful but coarse: it observes only the winning ID at each step. It does not show whether the attention vector is close, whether a second-best logit moved substantially, or whether the first differing token is merely near a tie.

A focused test can compare three levels. First compare the output vector of one attention head with the CPU two-pass reference for fixed query, keys, values, cache scales, and ring mapping. Then compare

PREPRINT

a layer's post-attention activation, which also checks output projection and residual boundaries. Finally run a model-level KL comparison when weights and a suitable reference are available. The first two are debugging instruments; the last measures the effect in the whole model.

For the low-level comparison, include awkward cases: a chunk with one key, a final partial chunk, a history just over 32 keys, a span just over the chunk cap, a query at position zero, a full 512-position sliding window, and a ring wrap. For Q8 include a block where every byte is zero, one with both positive and negative endpoints, and values whose quantization rounds near a half-step. These cases make scale and boundary errors easier to localize than a random long sequence.

The W0 softmax test in this chapter compares one scalar payload over one million synthetic scores. It demonstrates that the online state agrees with a stable two-pass calculation and avoids retaining scores in the online path. It does not test vector lane ownership, packed Q8 decoding, ring wrap, SIMD reduction order, masking, host dispatch, Metal barriers, or end-to-end model accuracy. A successful W0 run is evidence for one equation and one implementation, not a certificate for the kernel.

The m5c checkpoint adds chunks and merging to that toy. It should compare the full output vector against a two-pass implementation and vary chunk sizes, including sizes that do not divide the history. Only after that reference is trustworthy should a Metal kernel replace it. The engine can then compare the GPU output against the CPU task on captured activations without changing tokenization, model weights, or the rest of the forward pass.

Does the final score need to be part of this kernel?

No. The attention score is a temporary compatibility between one query and one key; it is consumed immediately by the softmax and value accumulation. The vocabulary score is a later vector of 262,144 logits. It is produced after all transformer layers. The shared word does not indicate shared data or shared timing. The attention kernel's result is a compact hidden vector, not a vocabulary distribution.

The greedy argmax can ignore a soft-cap because the cap function used by the model is monotonic: it changes logit magnitudes but preserves their order. Evaluation cannot ignore it because probabilities and log-sum-exp depend on the actual capped values. The GPU score path can apply the cap while each worker reads its logits, then reuse those values for its maximum, sum, selected-token loss, and optional saved distribution. This avoids separate full-vector passes for each statistic.

There is a tie rule too. The argmax reduction chooses the lowest token ID when logits tie. A deterministic tie break makes the result reproducible across runs and matches the CPU reference. In the attention reduction, score ties do not select a discrete item; equal scores simply produce equal softmax weights after normalization. Reusing an argmax helper for the attention calculation would therefore be both mathematically wrong and conceptually confusing.

Keeping these boundaries separate makes timing interpretable. If a generation step waits for a CPU-visible argmax, the logits readback can interrupt the pipeline. Writing the selected ID into its device-side destination avoids that round trip. If evaluation requests KL data, the score kernel performs extra work and writes statistics; those costs should not be attributed to attention. The attention kernels run earlier and have already written their head outputs before the LM head creates logits.

What belongs in a kernel review?

A useful review follows data as well as control. For decode, name the query owner, KV owner, chunk, lane dimensions, key scale address, value scale address, and partial-buffer slot. For prefill, name the query tile, key block, head-dimension partition, score fragment, softmax state, and output accumulator. If a code review cannot say which thread owns each of those pieces, a synchronization or indexing bug may be hidden in a compact expression.

Review the kernel against the CPU path in the same order: query and head mapping; visible logical position range; physical ring row; key retrieval and scale; score reduction; online max and sum; value retrieval and scale; output normalization; output type. Every stage has a simple reference interpretation.

PREPRINT

The GPU can change storage and parallel order, but it should not change the meaning of a position, scale, or output coordinate.

Then inspect host dispatch. Does T=1 choose the chunked decode route? Can the flash path handle this head size and ratio? Is Q8 actually supported by the chosen kernel? Does a nonzero window affect the start position? Are ring constraints satisfied? If the Metal wrapper declines, does the caller run the expected CPU task count? A kernel can be locally correct and still never be called for the shape the benchmark intended.

Finally inspect resource and timing claims. The number of threadgroups alone does not show occupancy; register footprint, SIMD-group count, memory traffic, and merge work all matter. An online algorithm can save a score buffer and still lose if it serializes too much work in one group. A matrix kernel can improve arithmetic reuse and lose if it spills accumulators. Those are the real tradeoffs shown by the documented experiments in this chapter.

How do the scale addresses stay aligned?

The scale pointer is indexed by cache row, KV head, and 32-value block. For one layer, a row has nkv values across all KV heads. The scale row therefore has nkv/32 half values. If the kernel is reading KV head g and the first dimension in that head is d0, the byte address begins at row*nkv + g*hd + d0. Dividing that element offset by 32 selects the corresponding scale. Because the head widths are multiples of 32, every head begins on a scale-block boundary.

For a 512-wide head, each row contributes 16 scales for that head's keys and another 16 for values. The K and V data arrays have distinct scale pointers. A global owner row with two KV heads holds 1,024 key bytes and 1,024 value bytes for the two heads, along with 32 key scales and 32 value scales. The decode kernel's lane group reads one head's 512 bytes and its 16 scales; it does not need the neighboring KV head's scales to interpret that vector.

The address ‘at/32‘ in the Q8 kernel works because ‘at‘ is the element offset into that layer's K or V array, not an absolute byte address including a scale buffer. The code uses the same data offset for the K and V row because both arrays share the same row/head layout. Their scale bases differ: ‘ksc‘ for K and ‘vsc‘ for V. Mixing those pointers can yield plausible, bounded numbers while applying the wrong scale sequence, so the error may appear only in output comparisons.

The packed load consumes four adjacent bytes at a time. For HD=512 each lane owns 16 dimensions, or four 32-bit words. Each word is separated into two half2 values, which expose four coordinates. The score loop multiplies those four values with four query coordinates; the value loop updates four output coordinates. This width matches both the contiguous cache layout and the 32-value scale blocks. A wider load is not automatically faster if it crosses a scale boundary or adds conversion and register work.

The simple CPU twin remains valuable here. `kv_get` calculates the scale index for every requested dimension and multiplies the signed byte by the half scale. It is slower, but its indexing expression is direct. When a GPU result diverges, compare the CPU's first few decoded floats with the values represented by the packed half construction before looking at the full softmax. That isolates byte interpretation from reduction order.

How does attention time grow with the history?

For a global layer, a decode query reads keys and values for every visible position. Doubling the history roughly doubles this layer's K/V traffic, although launch overhead and cache behavior can make a short timing depart from a straight line. Sliding attention behaves differently: once the window is full, the number of positions read per query remains bounded by the window. The global layers therefore account for the growing-history part of attention as decode continues.

The read volume is not simply the cache size. The cache stores rows for all owners, while one attention operation reads one layer's relevant K/V vectors. Four query heads share one KV head, so each reads the same rows for its own query. The engine's threadgroup arrangement may let those loads benefit from

DRAFT

the core cache, but the arithmetic still makes four distinct score streams and four distinct output vectors. Storage sharing does not mean the attention output is shared.

A kbench sweep by history depth can separate the basic trend from the rest of generation. Record the head width, number of KV heads, cache type, window, and chunk count along with the time. If the measured time grows much faster than the number of keys, inspect conversion, scale loading, and chunk merging. If it barely changes, check whether the intended long-context global layer actually ran, whether the profile includes GPU work, and whether a sliding window or fallback changed the path.

The chapter's central trade remains: decode divides a long, low-row-count scan among groups and pays for a merge; prefill uses matrix tiles to reuse cache reads across query rows and pays for on-chip synchronization. Both preserve the same attention calculation and cache meaning. The right kernel depends on how many query rows are available, what precision the cache uses, and whether the layer is global or sliding.

Can the packed-byte correction be checked by hand?

Use four signed cache codes as a toy example: -127, -1, 0, and 126. XOR with the sign bit maps these codes to unsigned low-byte values 1, 127, 128, and 254. Placing each into the half encoding for 1024 makes the lanes see 1025, 1151, 1152, and 1278. Subtracting the common 1152 bias leaves -127, -1, 0, and 126 again. This small check catches the easy mistake of subtracting 1024 instead of 1152.

With query coordinates all equal to one, the constructed dot contribution sums to 4,606. The query sum is four, so removing 1152 times that sum removes 4608, leaving -2, which is also the direct signed-code sum. If the four query values differ, the bias is still the same 1152 times their sum; it is not 1152 times the number of dimensions unless every query value is one.

The correction happens before applying the K scale. Suppose this block's scale is 0.01. The corrected integer dot -2 becomes -0.02. Scaling the uncorrected 4606 would produce 46.06, an answer with the wrong offset but a plausible finite magnitude. This is why a byte-unpack test should compare signed integers and scaled values before it compares final attention output.

The value path has a related but distinct correction. Each value code is multiplied by its probability and scale, so the accumulated bias depends on the sum of probability-times-scale terms. The kernel keeps that scalar as w and subtracts 1152*w from each output coordinate. It cannot reuse the key correction scalar: key correction is weighted by query coordinates and key scales; value correction is weighted by probabilities and value scales. The two scalar corrections belong to different stages of attention.

This toy calculation tests the representation, not quantization error. The cache writer first rounds real values into signed int8 codes and stores one fp16 scale per 32 values. The kernel must decode those codes exactly, but the decoded approximation still differs from the original pre-quantized vector. Separate these questions: does the byte transform reproduce the stored signed code, and does the quantized cache preserve model behavior closely enough? The first is a unit-level arithmetic check; the second requires a model-level comparison.

The real code

What is the transparent reference?

The CPU task assigns one task per query head and query block. It maps the query head to its shared KV head, copies the query, scores every allowed key, finds the maximum, exponentiates and sums, then makes a second walk over values to produce the output. Here is the loop's central range; it keeps the reference easy to compare against the online GPU method.

```
1320 static void attn_task(void *ctx, int task, int tid) {
1321     (void)tid;
1322     const AttnJob *job = ctx;
1323     const Layer *L = job->L;
```

PREPRINT

```
1324      const int hd = L->hd, nq = job->m->n_head * hd, nkv = L->n_head_kv *
          hd;
1325      const int h = task % job->m->n_head, kvh = h / (job->m->n_head / L->
          n_head_kv);
1326      const int t0 = task / job->m->n_head * ATT_QB, t1 = t0 + ATT_QB <
          job->T ? t0 + ATT_QB : job->T;
1327      float *p = xmalloc((size_t)(job->n_past + job->T) * sizeof(float)),
          *o = xmalloc(hd * sizeof(float));
1328      float *q = xmalloc(hd * sizeof(float)), *kv = xmalloc(hd * sizeof(
          float));
1329
1330      for (int t = t0; t < t1; t++) {
1331          const int pos = job->n_past + t, s0 = first_key(job->m, L, pos),
           s1 = job->bidir ? job->n_past + job->T - 1 : pos;
1332          for (int i = 0; i < hd; i++) q[i] = f16_table[job->q[(size_t)t *
           nq + h * hd + i]];
1333          float mx = -INFINITY, sum = 0;
1334          for (int s = s0; s <= s1; s++) {
1335              kv_get(job->k, job->ks, s % job->ring, nkv, kvh * hd, kv, hd
          );
1336              p[s] = dot(q, kv, hd);
1337              if (p[s] > mx) mx = p[s];
1338          }
1339          for (int s = s0; s <= s1; s++) { p[s] = expf(p[s] - mx); sum +=
          p[s]; }
1340          for (int i = 0; i < hd; i++) o[i] = 0;
1341          for (int s = s0; s <= s1; s++) {
1342              kv_get(job->v, job->vs, s % job->ring, nkv, kvh * hd, kv, hd
          );
1343              const float w = p[s] / sum;
1344              for (int i = 0; i < hd; i++) o[i] += w * kv[i];
1345          }
1346          for (int i = 0; i < hd; i++) job->o[(size_t)t * nq + h * hd + i]
           = float_to_half(o[i]);
1347      }
1348      free(p);
1349      free(o);
1350      free(q);
1351      free(kv);
1352  }
```

Listing 12.1: gemma4.c:1320–1352

For one query, the CPU computes all scores before reading values. It obtains each K/V row through kv_get, which also hides whether the cache is fp16 or Q8. The double pass is a clarity choice: it makes a stable maximum available before exponentiation. The output is converted to half at the boundary so the next engine stage sees the expected activation type.

Where does decode keep its running state?

A SIMD group handles one query head and one key chunk. Thirty-two lanes divide the head dimensions; for a 512-wide head, each lane owns sixteen dimensions. The kernel streams logical positions through ring rows, forms the scaled score, updates the online denominator and vector, then writes one partial per head and chunk. The first excerpt follows the score update and the K scale. It includes both Q8 and fp16 key paths.

```
1404      const int g = tg.x, c = tg.y, h = g * (p.n_head / p.n_head_kv) + sg,
          nd = HD / 32;
```

```
1405     const int first = p.window > 0 ? max(0, p.n_past - p.window + 1) :
         0;
1406     const int nkv = p.n_head_kv * p.hd, s0 = first + c * p.keys, s1 =
         min(s0 + p.keys, p.n_past + 1);
1407     float qv[16], o[16], m = -1e30f, l = 0, qsum = 0, w = 0;  // HD / 32
          used
1408     for (int i = 0; i < nd; i++) qv[i] = float(q[h * HD + lane * nd + i
         ]), o[i] = 0, qsum += qv[i];
1409     const ulong lo = g * p.hd + lane * nd;  // this lane's dims in a row
          (with KV8, inside one block of 32)
1410     // (8-bit: each simdgroup converts the same int8 for its own head,
         and int-to-float conversions limited the
1411     // kernel to 47 GB/s; through fp16 bit patterns instead, 68 GB/s at
         32K)
1412     for (int s = s0, r = s0 % p.ring; s < s1; s++, r = r + 1 == p.ring ?
          0 : r + 1) {  // r: the row of position s
1413         const ulong at = (ulong)r * nkv + lo;
1414         // (4-wide loads: a lane's nd values are contiguous and 4-
         aligned; one at a time, byte loads were the
1415         // 8-bit cache's bottleneck)
1416         float x = 0;
1417         if (KV8) {  // byte b (an int8 + 128) ORed into 1024.0h's
         mantissa is 1024 + b: no int-to-float conversions
1418             device const uint *ku = (device const uint *)((device const
         char *)k + at);
1419             for (int i = 0; i < nd / 4; i++) {
1420                 const uint u = ku[i] ^ 0x80808080u;
1421                 const half2 lo = as_type<half2>((u & 0x00FF00FFu) | 0
         x64006400u), hi = as_type<half2>(((u >> 8) & 0x00FF00FFu) | 0
         x64006400u);
1422                 x += qv[4 * i] * float(lo.x) + qv[4 * i + 1] * float(hi.
         x) + qv[4 * i + 2] * float(lo.y) + qv[4 * i + 3] * float(hi.y);
1423             }
1424             x = (x - 1152.0f * qsum) * float(ksc[at / 32]);
1425         } else {
1426             device const half4 *ks = (device const half4 *)((device
         const half *)k + at);
1427             for (int i = 0; i < nd / 4; i++) x += dot(float4(qv[4 * i],
         qv[4 * i + 1], qv[4 * i + 2], qv[4 * i + 3]), float4(ks[i]));
1428         }
1429         x = simd_sum(x);
1430         const float m_new = max(m, x), corr = exp(m - m_new), pr = exp(x
          - m_new);
1431         l = l * corr + pr;
```

Listing 12.2: `gemma4_metal.m:1404–1431`

The second excerpt follows the value accumulation and partial-state write. It shows why Q8 tracks a correction scalar alongside the output vector.

```
1432         if (KV8) {  // as for K: o accumulates p·d·(1024 + b), and w the
          p·d it must take 1152 times back
1433             device const uint *vu = (device const uint *)((device const
         char *)v + at);
1434             const float pv = pr * float(vsc[at / 32]);
1435             w = w * corr + pv;
1436             for (int i = 0; i < nd / 4; i++) {
1437                 const uint u = vu[i] ^ 0x80808080u;
1438                 const half2 lo = as_type<half2>((u & 0x00FF00FFu) | 0
```

```
      x64006400u), hi = as_type<half2>(((u >> 8) & 0x00FF00FFu) | 0
      x64006400u);
1439                  o[4 * i] = o[4 * i] * corr + pv * float(lo.x), o[4 * i +
       1] = o[4 * i + 1] * corr + pv * float(hi.x);
1440                  o[4 * i + 2] = o[4 * i + 2] * corr + pv * float(lo.y), o
      [4 * i + 3] = o[4 * i + 3] * corr + pv * float(hi.y);
1441              }
1442          } else {
1443              device const half4 *vs = (device const half4 *)((device
      const half *)v + at);
1444              for (int i = 0; i < nd / 4; i++) {
1445                  const float4 vv = float4(vs[i]);
1446                  for (int j = 0; j < 4; j++) o[4 * i + j] = o[4 * i + j]
      * corr + pr * vv[j];
1447              }
1448          }
1449          m = m_new;
1450      }
1451      device float *out = part + ((ulong)h * p.n_chunks + c) * (p.hd + 2);
1452      for (int i = 0; i < nd; i++) out[lane * nd + i] = KV8 ? o[i] -
      1152.0f * w : o[i];
1453      if (lane == 0) { out[p.hd] = m; out[p.hd + 1] = l; }
1454  }
```

Listing 12.3: gemma4_metal.m:1432–1454

Notice two levels of parallelism in the dispatch. Grid X is the KV head; SIMD group index selects which of its query heads this group handles. Grid Y is the key chunk. Lanes cooperate on the dot product using a SIMD sum, then each lane updates its owned output dimensions. After a threadgroup finishes, its scalar maximum and sum are written once by lane zero.

How do independent chunks become one output?

The combine kernel first finds the largest chunk maximum for one query head. Each dimension worker then visits chunks, weights each stored partial by the exponential difference from that common maximum, sums the numerator and denominator, and normalizes. It uses one threadgroup per output head and spreads dimensions across its workers.

```
1456  // head h, dims tid, tid + 256: merge the chunks of attn_decode
1457  kernel void attn_combine(device const float *part [[buffer(0)]], device
      half *o [[buffer(1)]],
1458                           constant AttnParams &p [[buffer(2)]], uint h [[
      threadgroup_position_in_grid]],
1459                           uint tid [[thread_index_in_threadgroup]]) {
1460      device const float *ph = part + (ulong)h * p.n_chunks * (p.hd + 2);
1461      float mx = -1e30f;
1462      for (int c = 0; c < p.n_chunks; c++) mx = max(mx, ph[c * (p.hd + 2)
      + p.hd]);
1463      for (int d = tid; d < p.hd; d += 256) {
1464          float num = 0, den = 0;
1465          for (int c = 0; c < p.n_chunks; c++) {
1466              device const float *pc = ph + c * (p.hd + 2);
1467              const float w = exp(pc[p.hd] - mx);
1468              num += w * pc[d];
1469              den += w * pc[p.hd + 1];
1470          }
1471          o[h * p.hd + d] = half(num / den);
1472      }
```

```
}
```

Listing 12.4: gemma4_metal.m:1456–1473

This is also a useful correctness reference for parallel reduction. Each chunk can be computed independently, but partial outputs cannot be added directly because they may use different maxima. The maximum field is not optional metadata; it defines the scale of the numerator and denominator that follow it.

What changes when eight queries travel together?

The flash kernel's comment and first part describe the ownership map: eight queries per threadgroup, query heads grouped by their KV head, and one or more 128-dimension partitions. It loads the query tile and initializes the matrix accumulators. The loop's first half shows the block stepping, score tile assembly, and combination of head-dimension partial scores.

The score loop reads four 8-key tiles and combines head-dimension partitions. This excerpt shows how those score fragments are formed.

```
    for (int s0 = s_first; s0 <= s_last; s0 += 32) {  // (the same keys for the whole threadgroup)
        const int nj = min(4, (s_last - s0) / 8 + 1);  // blocks of 8 keys with some at or before s_last
        simdgroup_float8x8 sf[4];
        ulong row[4];
        for (int j = 0; j < 4; j++) row[j] = (ulong)((s0 + 8 * min(j, nj - 1)) % p.ring) * nkv + g * p.hd + d0;  // (8 never wrap)
        for (int j = 0; j < 4; j++) {
            sf[j] = simdgroup_float8x8(0);
            if (j < nj)
                for (int d = 0; d < 16; d++) {
                    simdgroup_half8x8 kf;
                    simdgroup_load(kf, k + row[j] + d * 8, nkv, ulong2(0, 0), true);
                    simdgroup_multiply_accumulate(sf[j], qf[d], kf, sf[j]);
                }
        }
```

Listing 12.5: gemma4_metal.m:1201–1214

The loop continues by masking future causal positions and positions outside the sliding window, updating the online maximum and sum, rescaling the output accumulators, and multiplying the probability tile by values. The chosen source range shows those operations in one place; the loop's layout is more informative than a short isolated arithmetic fragment.

```
        const float m_new = bm > mx + LAZY ? bm : mx, corr = exp(mx - m_new);  // (lazy max: see attention_fa8)
        float rs = 0;
        simdgroup_half8x8 pf[4];
        for (int j = 0; j < 4; j++) {
            const float p0 = exp(x[2 * j] - m_new), p1 = exp(x[2 * j + 1] - m_new);
            rs += p0 + p1;
            pf[j].thread_elements()[0] = half(p0);
            pf[j].thread_elements()[1] = half(p1);
        }
        rs += simd_shuffle_xor(rs, 1);
        rs += simd_shuffle_xor(rs, 8);
        l = l * corr + rs;
```

```
1249          mx = m_new;
1250          if (!simd_all(corr == 1.0f)) {  // O's rows times corr, as a
      product with diag(corr): changing the
1251              simdgroup_float8x8 dm;      // accumulators' elements
      directly made the whole kernel 25% slower
1252              dm.thread_elements()[0] = fc.x == fc.y ? corr : 0.0f;
1253              dm.thread_elements()[1] = fc.x == fc.y + 1 ? corr : 0.0f;
1254              for (int d = 0; d < 16; d++) simdgroup_multiply(acc[d], dm,
      acc[d]);
1255          }
1256          for (int j = 0; j < 4; j++)
1257              if (j < nj)
1258                  for (int d = 0; d < 16; d++) {
1259                      simdgroup_half8x8 vf;
1260                      simdgroup_load(vf, v + row[j] + d * 8, nkv);
1261                      simdgroup_multiply_accumulate(acc[d], pf[j], vf, acc
      [d]);
1262                  }
1263      }
1264      const int t = t0 + fc.x;
1265      if (t >= p.T) return;
1266      device half *oh = o + (ulong)t * nq + h * p.hd + d0 + fc.y;
1267      for (int d = 0; d < 16; d++)
1268          for (int i = 0; i < 2; i++) oh[d * 8 + i] = half(acc[d].
      thread_elements()[i] / l);
```

Listing 12.6: gemma4_metal.m:1237–1268

Who chooses chunk sizes and kernel families?

The host dispatcher has the model-wide facts: whether this is a single query, the KV precision, window, head ratio, head width, and ring alignment. For decode it computes the visible span, adjusts keys per chunk to respect the partial-count cap, allocates a reusable partial buffer, and launches decode plus combine. The prefill branch selects a specialized flash kernel for supported sizes, then falls back to the general fp16 attention kernel when applicable.

```
2119      if (T == 1) {  // decode: split over the keys, then merge (a single
      query: bidir changes nothing)
2120          // keys per threadgroup: 32, or more once there would be over
      MAX_CHUNKS chunks per KV head, so that the
2121          // chunks' outputs (hd + 2 floats per head each), which
      attn_combine reads on few threadgroups, stay small.
2122          // Fewer, longer chunks were faster down to 64 chunks for the 8-
      bit cache and 16 for fp16 (at 32K, a global
2123          // layer's decode attention: 8-bit 1.04 ms with 64, 1.06 with
      128, 1.27 with 16; fp16 1.5 - 1.7 ms with 16,
2124          // 1.9 - 2.0 with 32 to 128); 1024 chunks of 32 (the first
      version) made 17 MB per layer to merge.
2125          const int MAX_CHUNKS = kv8 ? 64 : 16;
2126          static float *part;         // per query head and chunk: hd
      outputs, max, sum
2127          static size_t part_size;
2128          const int first = window > 0 && n_past - window + 1 > 0 ? n_past
       - window + 1 : 0, span = n_past + 1 - first;
2129          const int keys = span <= 32 * MAX_CHUNKS ? 32 : (span + 32 *
      MAX_CHUNKS - 1) / (32 * MAX_CHUNKS) * 32;
2130          const int n_chunks = (span + keys - 1) / keys;
```

```
2131         const size_t need = (size_t)n_head * n_chunks * (hd + 2) *
     sizeof(float);
2132         if (need > part_size) {  // (at most 128 chunks: a superseded
     buffer may still be in use, so it is kept)
2133             part_size = need;
2134             part = metal_alloc(part_size);
2135         }
2136         const AttnParams p = {T, n_head, n_head_kv, hd, 0, 0, 0, n_past,
      n_chunks, window, 0, 0, 0, ring, 0, keys, kv8};
2137         if (profiling) snprintf(prof_key, sizeof prof_key, "hd %d past %
     d%s", hd, n_past / 128 * 128, kv8 ? " kv8" : "");
2138         return ENCODE_PS(variant(K_ATTN_DECODE, 0, 0, 0, 0, 0, hd, 0, 0,
      kv8), K_ATTN_DECODE, p, size2(n_head_kv, n_chunks),
2139                          size2(32 * (n_head / n_head_kv), 1), q, k, v,
     part, ksb, vsb) &&
2140                ENCODE(K_ATTN_COMBINE, p, size2(n_head, 1), size2(256, 1)
     , part, o);
```

Listing 12.7: gemma4_metal.m:2119–2140

The prefill selection has two specialised flash cases followed by a guarded general path. This second excerpt contains the grid and threadgroup sizes for those paths.

```
2142     const int G = n_head / n_head_kv;
2143     const int NPh = hd / 128, Gs = G < 16 / (NPh ? NPh : 1) ? G : 16 / (
     NPh ? NPh : 1);  // (as in attention_fa8)
2144     if (hd % 128 == 0 && hd <= 512 && G <= 8 && G % Gs == 0 && ring % 8
     == 0 && (G <= 4 || kv8)) {  // the language model's heads (G > 4: the
      8-bit kernel only)
2145         const AttnParams p = {T, n_head, n_head_kv, hd, 0, 0, 0, n_past,
      0, window, 0, bidir, 1, ring, 0, 0, kv8};
2146         if (profiling)
2147             snprintf(prof_key, sizeof prof_key, "T%-4d hd %d past %d%s%s
     ", T, hd, n_past / 128 * 128, bidir ? " bidir" : "",
2148                      kv8 ? " kv8" : "");
2149         if (!kv8)
2150             return ENCODE_PS(variant(K_ATTENTION_FA, 0, 0, 0, 0, 0, hd,
     0, 0, 0), K_ATTENTION_FA, p, size2(n_head_kv, (T + 7) / 8),
2151                              size2(32 * G * (hd / 128), 1), q, k, v, o);
2152         return ENCODE_PS(variant(K_ATTENTION_FA8, 0, 0, 0, 0, 0, hd, 0,
     0, 0), K_ATTENTION_FA8, p, size2(n_head_kv * (G / Gs), (T + 7) / 8),
2153                          size2(32 * Gs * (hd / 128), 1), q, k, v, o, ksb
     , vsb);
2154     }
2155     if (kv8) return 0;  // (the general kernel reads an fp16 cache only)
2156     // a threadgroup takes the G query heads of a KV head over qblocks
     blocks of 8 queries (4 simdgroups at least),
2157     // so their reads of the same keys and values come from the core's
     cache
2158     const int qb = G >= 4 ? 1 : 4 / G;
2159     const AttnParams p = {T, n_head, n_head_kv, hd, 0, 0, 0, n_past, 0,
     window, 0, bidir, qb, ring, 0, 0, 0};
2160     if (profiling) snprintf(prof_key, sizeof prof_key, "T%-4d hd %d past
      %d%s", T, hd, n_past / 128 * 128, bidir ? " bidir" : "");
2161     return ENCODE_PS(variant(K_ATTENTION, 0, 0, 0, 0, 0, hd, 0, 0, 0),
     K_ATTENTION, p,
2162                      MTLSizeMake(n_head_kv, (T + 8 * qb - 1) / (8 * qb),
      (hd / 8 + 31) / 32), size2(32 * G * qb, 1), q, k, v, o);
```

Listing 12.8: `gemma4_metal.m:2142–2162`

The C caller is small because the dispatch policy lives in Metal's host wrapper. It passes `ATT_QB` into the CPU task count, while the Metal host chooses exact grid sizes from the same head and token dimensions. It also exposes the fallback contract: a zero return from the Metal routine tells `STEP` to use the CPU reference.

```
1355 static void attention(const Model *m, const Layer *L, int l, Acts *a,
     State *st, int T, int np) {
1356     const int hd = L->hd, nkv = L->n_head_kv * hd, q_only = L->kv_src !=
      l;
1357     float *k_new = a->kv, *v_new = L->wv ? a->kv + (size_t)a->T_max *
     nkv : k_new;
1358     const int bidir = a->bidir && L->swa, ring = kv_rows(L, st->n_ctx,
     st->ring);
1359     AttnJob job = {m, L, T, np, a->q, k_new, v_new, a->qr, st->k[l], st
     ->v[l], st->ks[l], st->vs[l], ring, a->o, bidir, q_only};
1360
1361     gpu_group(1);
1362     matmul(a->q, a->xn, L->wq, T, 0);
1363     if (!q_only) matmul(k_new, a->xn, L->wk, T, 0);
1364     if (!q_only && L->wv) matmul(v_new, a->xn, L->wv, T, 0);
1365     gpu_group(0);
1366     STEP(metal_attn_prep(job.q, job.k, job.v, job.ks, job.vs, job.q_new,
      k_new, v_new, L->q_norm,
1367                          q_only ? L->q_norm : L->k_norm, L->rope_freqs,
     T, np, m->n_head, L->n_head_kv, hd, L->n_rot,
1368                          L->rope_base, m->eps, ring, q_only),
1369          T, attn_prep, &job);
1370     STEP(metal_attention(job.o, job.q, job.k, job.v, job.ks, job.vs, T,
     np, m->n_head, L->n_head_kv, hd,
1371                          L->swa ? m->n_swa : 0, bidir, ring),
1372          m->n_head * ((T + ATT_QB - 1) / ATT_QB), attn_task, &job);
1373     matmul(a->a, job.o, L->wo, T, 0);
```

Listing 12.9: `gemma4.c:1355–1373`

Experiment

Can the online state match a two-pass softmax without weights?

This W0 exercise generates a fixed million-score stream and compares online softmax with a conventional two-pass CPU calculation. Both compute a weighted scalar payload as well as the denominator; the online path does not retain the million scores. It prints the two weighted scalar results and their absolute difference. The deterministic synthetic values make the output reproducible; the test says nothing about model accuracy or GPU speed.

Compile and run from the book root:

```
nice -n 10 cc -O2 -std=c11 -Wall -Wextra \
  -o code/build/ch12_softmax code/examples/ch12_softmax.c
nice -n 10 code/build/ch12_softmax
```

The captured output is in `code/expected/ch12-w0.txt`. Check that the maximum difference is small and both methods produce the same result to the printed precision. The CPU method is a verification target, not a benchmark comparison: it performs the extra storage and passes on purpose.

What must a GPU capture record?

This is W2. In a scratch copy with your local model, run the snapshot build and the two depth cases from the project root. The exact command is:

```
nice -n 10 code/build/gemma4 -m "$MODEL" --bench 512 64 --depth 4096
nice -n 10 code/build/gemma4 -m "$MODEL" --bench 512 64 --depth 16384
```

Use the snapshot's `kbench` attention mode as well to isolate attention time by depth. Record the machine, date, idle-gate reading, two-minute cooldown, statistic across interleaved runs, and whether the reported time is wall or GPU time. Inspect decode throughput at the final history length and compare the 4K, 16K, and 33K shapes with the dated README figures. **Try it on your machine (W2):** inspect decode throughput at each history length and isolated attention time by depth. The attention/cache argument predicts longer histories cost more; the listed 29.9, 27.5, and 24.7 tokens/s are dated project measurements (README, "The 8-bit KV cache"; ENGINE-MATH §11.7, 2026-10-06), not fresh results.

Dead ends and pitfalls from the project notes

Pitfall: fewer conversions can beat a more obvious vector load

The first Q8 decode attempts tried four-wide loads, matrix decode with half of each eight-column product wasted on padding, and a lane layout that assigned four dimensions across all four query heads. These changes either left conversion as the bottleneck or added shuffles and barriers. The byte-to-half bit-pattern trick improved the recorded 32K global-layer path from 47 to 68 GB/s. The source comments in `gemma4_metal.m:1410--1421` preserve the conversion rationale; the measured comparison is recorded in the host tuning comment at lines 2120–2124.

Dead end: more chunks do not always mean more parallelism

The initial 32-key chunks produced 1,024 partials at 32K, around 17 MB of merge data per layer in the documented first version. Making the chunk count smaller reduced that traffic. Recorded tests found 64 Q8 chunks and 16 fp16 chunks faster than more numerous chunks at 32K. The current dispatcher encodes those caps and raises the chunk size when the span grows.

Pitfall: a loop bound can change where the accumulators live

The project notes describe a vision-attention loop whose run-time bound prevented unrolling. Its per-thread accumulators spilled to memory, producing a 206 MB grid allocation. Constant bounds kept them in registers; the documented footprint fell from 300 MB to 99 MB with bit-identical output. That is a separate vision kernel, but the lesson applies to attention source reading: a loop that looks equivalent can change register allocation. Source: `README.md:341`; the language-model flash kernel comments on this concern at `gemma4_metal.m:1198--1200`.

Exercises

Exercises

1. **Read.** Start with scores 1, 3, and 2 and scalar values 10, 20, and 40. Write the running maximum, denominator, and numerator after each score. Show that the final normalized value matches a two-pass softmax.
2. **Calculate.** At 32K positions, how many 32-key chunks would a fixed chunk length create? Compare that count with the Q8 cap. What chunk length does the dispatcher choose after rounding to a multiple of 32?
3. **Trace.** For global query head 7, identify its KV head and its position within that KV head's four-query group. Use the dispatcher and kernel indices to follow it to grid X, grid Y, SIMD group, and lane dimensions.
4. **Explain.** Why can the partial results from different chunks not be added directly? Give the correction factor when one chunk has maximum 2 and the shared maximum is 5.
5. **Calculate.** For a 256-dimension Q8 head, how many scale values are read for one K row? How many bytes hold the byte values and scales together? Repeat for K and V.
6. **Measure.** Run the W0 program after changing the deterministic score function. Compare error and runtime separately; do not interpret this synthetic test as a GPU benchmark.

Rebuild checkpoint

Rebuild checkpoint

Milestone m5c: chunked attention with a merge. Build a small C implementation that takes one query, a key/value array, and a chunk length. Compute each chunk's online maximum, denominator, and value numerator, then merge them with a common maximum. Test against the plain two-pass attention calculation over fixed random inputs and lengths that do not divide evenly by the chunk size. Include a wrapped ring case by mapping logical positions through modulo capacity. The test should compare every output coordinate within a stated tolerance; checking only the top coordinate would miss many errors.

PRELIMINARY

Chapter 13

The feed-forward block and per-layer embeddings

PREPRINT

The question

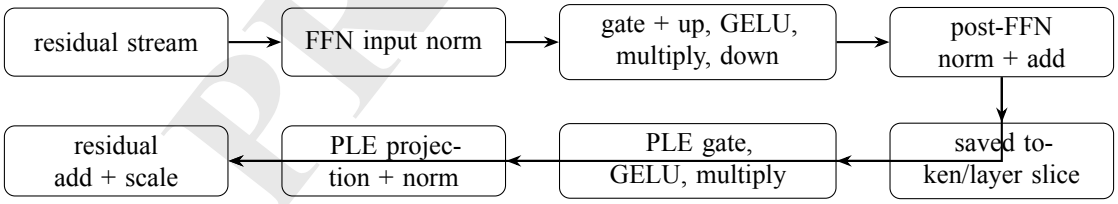
Where do the other 2.8 billion parameters hide, and what does the feed-forward block do with a token?

The idea

What happens after attention has mixed the token with its neighbours?

Attention lets a token gather information from other positions. The next block works on each position separately. It first widens the token's vector, applies a gate, and compresses the result back to the model width. In a normal dense transformer this feed-forward network (FFN) is a large share of the layer's learned weights. Gemma 4 E4B adds another operation after it: a small per-layer embedding (PLE) branch, driven by both the token ID and the current layer's transformed state.

Keep the residual stream in view. The FFN does not replace the vector that attention produced. It computes a branch result, normalises that result with a learned vector, and adds it to the residual. PLE then makes its own branch from the updated residual and the layer-specific input prepared earlier. Its residual add comes before the layer's output scale. That order is a real part of the model.



The diagram shows one position through one layer. The saved PLE input is not computed inside the FFN: the engine prepares all token/layer inputs before the layer loop. At layer five, for example, the PLE gate uses slice five of the current token's prepared vector. The PLE branch joins the same residual stream that has already received the FFN update.

A decode token also has an attention update before this picture begins. In the CPU path that update is followed by a post-attention residual and a normed FFN input. In the GPU path, fused steps can prepare the normed input as they write the residual. The chapter focuses on the tail shared by both paths: FFN, its residual, PLE, then the scale.

Why widen and then narrow?

The E4B model width is 2,560 numbers. The FFN gate and up projections each produce 10,240 numbers, four times wider. The gate branch passes through GELU; each resulting number then controls the cor-

responding up-branch number through an element-by-element multiply. This is the gated GELU linear unit, usually shortened to GeGLU. A final down projection returns the 10,240 values to 2,560 so they can be added to the residual.

Think of the up projection as proposing features and the gate as deciding how much of each feature should pass. Both projections read the same normalised input. The gate is not a scalar for the whole token: there are 10,240 gate values, paired by index with 10,240 up values. The down matrix mixes those gated features back into the model width.

The activation is the engine's GELU approximation, with the tanh form used by this text path. The `quick` option in `glu` selects a sigmoid approximation for a different encoder use; the FFN caller passes `quick = 0` and no clamp. That distinction matters: a function with the same name serves more than one workload, but the model's text FFN takes the GELU path.

The real shapes

How do the file's shapes describe three different projections?

GGUF stores matrix dimensions in input-first order. For layer zero the gate and up tensors each have dimensions `[2560, 10240]`: 2,560 input features and 10,240 output rows. The down tensor is `[10240, 2560]`, consuming the expanded vector and producing a residual-width vector. The C loader uses `ne[0]` as the row length and `ne[1]` as the number of rows, so reading the order backwards is a common source of an accidental transpose.

Here is a W1 inspection of the actual E4B GGUF. The type labels and tensor descriptors came from its metadata directory; the byte sizes are computed from the declared quantisation block format and tensor shape. No tensor payload was read. For IQ4 XS, each block of 256 values occupies 136 bytes. The table gives the exact byte count for each projection: about 13.3 MiB each, or 39.8 MiB for the three weights together.

Layer-zero tensor	Type	GGUF dimensions	Values	Bytes
FFN gate	IQ4_XS	2560 × 10240	26,214,400	13,926,400
FFN up	IQ4_XS	2560 × 10240	26,214,400	13,926,400
FFN down	IQ4_XS	10240 × 2560	26,214,400	13,926,400
PLE gate (`inp_gate`)	F32	2560 × 256	655,360	2,621,440
PLE projection (`proj`)	F32	256 × 2560	655,360	2,621,440

The three FFN matrices total 41,779,200 bytes in this file, around 39.8 MiB. The two layer-zero PLE matrices occupy 5,242,880 bytes on disk as F32. The loader converts those PLE matrices into F16 copies, so their active copies take 2,621,440 bytes together. This difference between on-disk type and runtime copy is why a table should report both when discussing memory.

Does the GPU build both widened vectors in separate passes?

The simple C reference does. It materialises the gate projection into one temporary array, the up projection into another, then loops over both to apply GELU and multiply. That sequence is easy to follow and provides the fallback when the GPU cannot handle the tensor types. The CPU path may therefore need temporary space proportional to two expanded activations.

The Metal path has a fused gate-and-up kernel. In the shared matvec implementation, `GLU` makes each SIMD group compute output rows from both matrices. It accumulates each set of rows in the same input traversal, reduces both sums, and writes only the final GELU-times-up result. The outputs of the two projections never need separate device arrays. This saves an intermediate write and read and avoids two dispatches for decode. It does not fuse the later down projection or the FFN residual update; those are following steps.

The kernel supports multiple weight types, so its central loop is longer than the final store shown here. For this model's layer-zero gate and up tensors, both are `IQ4_XS` and use the quantised block branch of

PREPRINT

that kernel. The final lines make the fusion visible: both dot products have already been accumulated as `v` and `u`; only their gated product is written.

Why are there two wide matrices instead of one?

A plain two-layer MLP could make one wide projection, apply a nonlinearity, and project down. GeGLU spends another matrix of the same size on the gate. That is a substantial cost: at layer zero the two wide matrices together account for 27,852,800 encoded bytes. The gate earns that cost by letting each feature be suppressed, passed through, or scaled according to a learned response to the same input. A single activation applied to the wide projection cannot make that separate, input-dependent decision for every feature.

The gate/up pair also explains why the fusion is more than a cosmetic rewrite. A standalone implementation would write two arrays of 10,240 half-precision values, then read them again in the elementwise stage. For one token that is 40 KiB of temporary writes and another 40 KiB of reads, before considering allocator or command overhead. The fused decode kernel keeps the two accumulators in registers until both reductions are ready and writes the 20 KiB hidden result once. These activation sizes are small beside the encoded weights, but one dependent dispatch repeated across 42 layers can be noticeable on a decode path.

Do not overread this comparison as a measured speed claim. It describes the data that the two implementations must move and the dispatch structure in the source. Actual time depends on quantised block decoding, the number of rows, and GPU occupancy. Chapter 21 has the measurement rules; this chapter's W1 experiment only establishes the model file's shapes.

The C fallback is deliberately easier to inspect. It allocates its two floating-point scratch arrays lazily and grows them to fit the token batch. The GPU implementation avoids those projection intermediates for the common fused case. In either path, the result `h` is half precision, which is what the next matrix multiply consumes. This precision boundary is visible in `glu` and in the Metal store.

What changes when the call contains many token rows?

The layer shapes do not change with batch size. Each row starts at the model width and makes two 10,240-value projections; the down projection returns one 2,560-value row. The CPU fallback calls the row-aware matmul helper for each projection, then applies the activation and product over the flattened T-by-10,240 result. The GPU dispatch has a separate matrix path for prefill, because it can reuse each quantised weight tile across several token rows instead of streaming that tile again for every row.

For one decode row, the compute is matrix-vector work. The matrices are large and each is read once for that token. For a prompt chunk, the same matrix is used against many rows. The input rows and output fragments become a matrix multiply, and each weight tile can serve more than one row while it is in the GPU's working set. The book introduced that difference in Chapter 8; here the important point is that gate, up and down are three distinct projections in both modes. Fusion of gate and up is specifically a decode kernel choice, not a claim that all three projections become one operation for every batch size.

The C `glu` wrapper makes this distinction explicit. It tries `metal_glu_norm` when the FFN input is still a deferred norm. Otherwise it materialises the norm and tries `metal_glu`, then drops to the C implementation if the GPU declines. The C result is the reference path and the safety net for unsupported combinations. The next `matmul` is the down projection, and the FFN output stage combines its result with the residual. Following dispatch code is where the batch behavior and fallback become concrete; the matrix shapes remain the same even as each kernel uses a different tiling strategy.

The project also has distinct quantised matvec kernels for several formats. `IQ4_XS` block size differs from the old `Q4_0` path, so a fused gate/up implementation still has to decode the file's actual block layout correctly. The chapter does not need a second quantisation tutorial: Chapter 4 covers the block bytes, and Chapter 7 covers how a SIMD group distributes output rows. Here the useful connection is that both matrices travel through the same shared input loop while the gate and up accumulators remain separate.

PREPRINT

What does the CPU reference keep explicit?

This excerpt derives K and N from the gate tensor, tries the fused Metal operation when the input norm is still deferred, and otherwise materialises the norm before the two CPU matmuls. The final loop applies the ordinary GELU and elementwise product. The boundary between the CPU reference and the GPU path is a useful guide: the math stays the same even though the amount of temporary storage differs.

```
1409 static void glu(f16 *h, const f16 *xn, const Tensor *g, const Tensor *u,
         int T, int quick, const float *clamp) {
1410     const int K = (int)g->ne[0], N = (int)g->ne[1];
1411     if (!quick && !clamp && xn == deferred_norm.out &&
1412         GPU(metal_glu_norm(h, deferred_norm.x, deferred_norm.w,
        deferred_norm.eps, g->data, row_bytes(g), g->type,
1413                            u->data, row_bytes(u), u->type, K, N)))
1414         return;
1415     materialize_norm(xn);
1416     if (clamp ? !quick && GPU(metal_glu_clamp(h, xn, g->data, row_bytes(
        g), g->type, u->data, row_bytes(u), u->type, T, K,
1417                                                 N, clamp))
1418               : GPU(metal_glu(h, xn, g->data, row_bytes(g), g->type, u->
        data, row_bytes(u), u->type, T, K, N, quick)))
1419         return;
1420     gpu_sync();
1421     if ((size_t)T * N > glu_tmp_n) {
1422         for (int i = 0; i < 2; i++) free(glu_tmp[i]), glu_tmp[i] =
        xmalloc((size_t)T * N * sizeof(float));
1423         glu_tmp_n = (size_t)T * N;
1424     }
1425     const float *x = to_float(xn, (size_t)T * K);
1426     matmul_f32(glu_tmp[0], x, g, T, 0);
1427     matmul_f32(glu_tmp[1], x, u, T, 0);
1428     for (size_t i = 0; i < (size_t)T * N; i++) {
1429         const float g = clamp ? fminf(fmaxf(glu_tmp[0][i], clamp[0]),
        clamp[1]) : glu_tmp[0][i];
1430         const float up = clamp ? fminf(fmaxf(glu_tmp[1][i], clamp[2]),
        clamp[3]) : glu_tmp[1][i];
1431         h[i] = float_to_half((quick ? g / (1.0f + expf(-1.702f * g)) :
        gelu(g)) * up);
1432     }
```

Listing 13.1: `gemma4.c:1409–1432`

The Metal kernel stores the paired result in fp16. This short excerpt shows the GLU store only; the surrounding kernel has accumulated gate and up separately without writing either full projection to memory.

```
460     for (int r = 0; r < MV_R; r++) {
461         const float v = simd_sum(acc[r]), u = GLU ? simd_sum(acc[r +
        MV_R]) : 0.0f;
462         const int n = n0 + r;
463         if (lane != 0 || n >= p.N) continue;
464         if (GLU) ((device half *)Y)[n] = half(gelu(v) * u);
465         else if (PLG) ((device half *)Y)[n] = half(gelu(v) * ((device
        const float *)W2)[n]);
466         else Y[n] = clamp(v, p.clamp.x, p.clamp.y) + (p.accumulate ? Y[n
        ] : 0.0f);
467     }
468 }
```

Listing 13.2: `gemma4_metal.m:460–468`

The per-layer input

Where does a token get a different extra vector for every layer?

E4B has 42 layers, and each layer receives an extra 256-number input associated with the current token. The token table is shaped 262,144 by 10,752: the vocabulary size by 42 × 256. A row is laid out as 42 adjacent slices of 256 numbers, one slice per layer. When the input is ordinary text, the row is selected by token ID. Media rows use the model's designated row zero.

There is also a learned projection of the ordinary token embedding. The embedding has width 2,560; the BF16 projection expands it to 10,752 values, which are likewise viewed as 42 slices of 256. For each slice, the engine RMS-normalises the projected part using a shared 256-value norm vector, adds the corresponding 256-value table slice, and scales the combination by one over the square root of two. The table component is scaled by the square root of 256 before the combination; the projection is scaled by one over the square root of 2,560 inside its normalisation path.

The two sources serve different purposes. The projected component starts from the token's ordinary embedding and gives a dense learned contribution to every layer. The table adds a token-specific contribution at each layer directly. Thus the same token ID has a distinct input slice at layer zero, layer one, and so on, even before the evolving residual stream reaches those layers.

What do the PLE scales do before the layer loop?

The table and the projection do not begin on equal footing. Table values are quantised and represent a token's direct layer-specific input; projection values are produced from the ordinary embedding. The engine scales the table values by the square root of the PLE width, scales the projected slice inside its RMS-normalisation path by the inverse square root of the model width, and then combines the two sources with a factor of one over the square root of two. Those are fixed dimension-based scalings, read alongside the widths in the GGUF.

The numbers are easy to sanity-check. The per-layer width is 256, so its square root is 16; the embedding width is 2,560, so its inverse square root is about 0.0198; the final combination factor is about 0.707. The shared norm has 256 learned values. These constants do not mean that every input value has a predictable size: the projected branch also passes through a per-slice RMS norm, and the learned norm weights alter individual coordinates. They do show that the two sources are deliberately adjusted before their sum.

A four-coordinate sketch helps keep the order straight. Pretend a table slice begins with [2, -1, 0, 3], and the projected slice begins with [1, 2, -2, 0]. Before adding them, the table values receive the width scale, while the projected values are normalised and weighted. The outputs are not simply added as raw quantised integers to raw projection numbers. After the combination factor, the resulting 256-value vector becomes the layer's saved PLE input. The real slice has 256 values; the four entries here illustrate the separate paths, not model output.

One norm vector is shared across the 42 projected slices. Sharing a 256-value vector saves parameters and gives every layer the same normalization rule for this part of the input, while the table and later gate/projection matrices still vary by token or layer. This arrangement is different from copying an entire token embedding into every layer: the ordinary embedding is projected once to a concatenated vector, and the large token table supplies its own per-token set of layer slices.

How does the engine find a row without loading the table into RAM?

At model load, `load_model` finds the PLE tensors and checks that the token-table row width equals layer count times the per-layer width. It opens a second file descriptor with `F_NOCACHE`, then marks the table region `MADV_RANDOM`. These choices tell macOS that the CPU will make scattered row reads and that it should not build a read-ahead cache around each row. The table is still part of the read-only mapped GGUF, but the loader does not copy it into an allocation or convert it into a GPU buffer.

PREPRINT

```
943      m.n_pl = (int)kv_num(g, "gemma4.embedding_length_per_layer_input",
         0);
944      const int n_kv_layers = m.n_layer - (int)kv_num(g, "gemma4.attention
         .shared_kv_layers", 0);  // layers with a cache
945      if (n_kv_layers < 1) die("no layers with a KV cache");
946      m.n_kv = n_kv_layers;
947      int fd = -1;
948      if (m.n_pl) {
949          if ((fd = open(g->path, O_RDONLY)) < 0) die("cannot open %s", g
         ->path);
950          fcntl(fd, F_NOCACHE, 1);
951          m.pl_tok = find_tensor(g, "per_layer_token_embd.weight", 1);
952          m.pl_model_proj = find_tensor(g, "per_layer_model_proj.weight",
         1);
953          m.pl_proj_norm = f32_data(find_tensor(g, "per_layer_proj_norm.
         weight", 1));
954          if (m.pl_tok->ne[0] != (int64_t)m.n_layer * m.n_pl || m.
         pl_model_proj->ne[1] != (int64_t)m.n_layer * m.n_pl ||
955              m.n_pl % 32)
956              die("unexpected per-layer embedding shapes");
957          // the token table is read a row (a token) at a time: no read-
         ahead of its neighbours
958          const size_t page = (size_t)getpagesize(), a = (uintptr_t)m.
         pl_tok->data / page * page;
959          madvise((void *)a, (uintptr_t)m.pl_tok->data + tensor_nbytes(m.
         pl_tok) - a, MADV_RANDOM);
960      }
```

Listing 13.3: gemma4.c:943–960

The row lookup happens before the layer loop. `pl_inputs` submits earlier GPU work, waits for the previous step that produces the token ID, reads and dequantises each selected table row on the CPU, and then combines it with the projected per-layer vectors. In decode, this ordering lets the CPU perform the irregular file lookup while the GPU is working on already-submitted computations. The wait is necessary because the newly sampled token ID is not known until the GPU completes the preceding step.

```
1955 // a->pl = the per-layer inputs of this call's tokens, from a->pl (the
         projection) and the token table. The CPU
1956 // needs the tokens: in a pipelined decode the previous step's GPU work
         (`prev`, all submitted before this call)
1957 // samples them, so this waits for it, while the GPU runs what this call
          has submitted so far.
1958 static void pl_inputs(const Model *m, Acts *a, int T, void *prev) {
1959      gpu_submit();
1960      gpu_wait(prev);
1961      PlJob job = {m, a};
1962      if (T >= 16) parallel_for(T, pl_tok_row, &job);
1963      else for (int t = 0; t < T; t++) pl_tok_row(&job, t, 0);
1964      PlCombineJob cj = {a->pl, a->pl_tok, m->pl_proj_norm, m->n_pl, m->
         eps, 1.0f / sqrtf((float)m->n_embd), sqrtf(0.5f)};
1965      STEP(metal_pl_combine(a->pl, a->pl_tok, m->pl_proj_norm, T * m->
         n_layer, m->n_pl, m->eps, cj.s_in, cj.s_out),
1966           T * m->n_layer, pl_combine_row, &cj);
1967 }
```

Listing 13.4: gemma4.c:1955–1967

This excerpt shows the sequence and combination parameters. For short decode calls the CPU handles rows directly; larger batches use the thread pool. The row reader scales the table values by the square

root of the PLE width; `metal_pl_combine` or its C reference then normalises and joins the projection slice with that table slice.

Why prepare all layer inputs before the layer loop?

The per-layer table lookup has an awkward dependency: its row is chosen by the token ID, but its 42 slices are consumed at different points in the forward pass. The engine resolves that dependency at the beginning. It dequantises the selected token row into a temporary activation, computes the ordinary embedding projection, then combines their corresponding layer slices. After preparation, the layer loop can use a direct pointer to the right 256-value slice.

For a batch of T token rows, this prepared data has $T \times 42 \times 256$ values. It is an activation buffer, not a new copy of the vocabulary table. The lookup result for one token is 10,752 floats before and during combination; the persistent model state remains dominated by the token table on disk. This distinction between model parameters and per-call activations matters when reading memory reports: the large table remains file-backed while the selected rows and projected vectors are ordinary working data.

During prefill, the batch can contain many prompt positions. The CPU must read one 7,392-byte row for each token ID, and the projection computes all 42 layer slices for each embedded row. This setup is more substantial than the one-row decode lookup, but it still avoids transferring the 1.94 GB table to GPU memory. The subsequent matrix operations over many rows can use the GPU's prefill kernels. The source's row threshold chooses whether table lookups are done serially or by the CPU thread pool; it does not change the meaning of the prepared vectors.

During decode, the batch is commonly one new token. The irregular row access remains on the CPU. At the same time, the GPU can run work already submitted for the previous token. The host then waits for that previous result because it contains the new token ID. The ordering is a pipeline boundary, not a second model pass: the CPU resolves the next row while the GPU begins the newly submitted work, and both sides meet before the layer sequence needs the prepared slice.

There is a similar boundary at the start of the model. The ordinary token embedding produces the residual stream and feeds the BF16 PLE projection. The table contributes an independent vector. The shared norm and scale factors make these sources comparable before they are added. Neither source is a replacement for the residual; they make a side input that each layer's PLE gate will use after the FFN.

Why is the layer slice a view into one prepared row?

A token's table row is stored contiguously. Slice zero is the first 256 values, slice one follows it, and the last slice completes the 10,752-value row. In the activation buffer, the next token's row begins after that full width. The layer loop therefore selects its view by adding the layer index times 256 to the token row's base. The stride between tokens is the whole 10,752 values; the visible vector passed into the PLE block is only 256 values.

This layout avoids gathering 42 scattered fragments from the large GGUF file. The CPU fetches one contiguous quantised row, dequantises it once, and stores a contiguous floating-point activation. The PLE gate at each layer then reads a compact 256-value slice from that activation. Layer five's slice is not found by searching tensor names at runtime and it is not a new file read; its position is determined by the fixed layer index and width.

The shared projected vector uses the same concatenated organization. One matrix operation maps a token embedding to 10,752 output values, then the combining operation works row by row and layer chunk by layer chunk. The norm vector has length 256 and is reused for every layer's projected chunk. It is not 42 separate norms. This shared normalization is another reason the table and projection are combined before the main layer loop.

PREPRINT

What happens between the file row and the layer slice?

The GGUF does not store each PLE table entry as a standalone five-bit number. Q5_K packs groups of 256 values together with the scales needed to reconstruct them. A token row is exactly 42 such blocks, because its 10,752 values divide evenly into 42 groups of 256. The W1 probe can calculate the row's encoded size from that block layout without decoding a single value; the engine's normal `dequant_row` path performs the actual conversion when a token needs the row.

The row reader begins at the token's tensor row and expands the selected quantised values into floating-point entries in the activation buffer. It does not ask the GPU to decode the whole vocabulary table, and it does not decode all 262,144 rows on load. After one row has been expanded, the engine multiplies its values by the table-side scale and combines them with the projected activation. The row's storage is temporary; its 42 slices remain available until the forward call has finished using them.

This separation also clarifies what a file-backed mapping means. The full table has an address inside the mapped GGUF, but accessing one row only touches the pages covering that row and whatever operating-system page granularity brings along. `MADV_RANDOM` discourages sequential read-ahead because the next token ID can point anywhere in the vocabulary. `F_NOCACHE` applies to the separate file descriptor used for the one-time F32-to-F16 matrix copies; it is not a request to throw away the token row before the layers use its activation copy.

A useful check is to compare the two sizes without confusing them. The encoded row is 7,392 bytes; after dequantisation, 10,752 float values occupy 43,008 bytes. That larger working row is still only about 42 KiB, while the original table is nearly 1.94 GB. The CPU pays a conversion cost for the row it actually needs, and the GPU can consume its compact 256-value slices while the rest of the table stays out of the GPU mapping.

Inside one layer's PLE block

When does the extra input meet the changing residual?

The prepared per-layer input waits until after the FFN. At this point `x` already contains the attention update and the FFN update. The PLE gate projects the current residual, applies GELU, and multiplies it elementwise by the saved 256-value slice for this layer. Its output has width 256. The second PLE matrix projects those 256 values back to width 2,560. The engine applies a learned post-PLE norm to that branch and adds it to the residual.

One detail is easy to miss: the PLE gate reads `xh`, the fp16 representation of the residual, and does not first use the FFN's normalised input. The FFN gate reads the normalised FFN input; the PLE gate does not. The final PLE residual add is the point where the layer's `out_scale` is applied. The earlier FFN residual update uses scale 1 when PLE exists, so the scale cannot accidentally move ahead of the PLE branch.

The first part of `pl_block` sets up the layer-specific slice and starts the gate projection. In a single-token GPU decode, `metal_matvec_gelu_mul` can do the PLE matrix-vector product, GELU, and multiplication with the saved slice in one operation. With multiple rows, or when that fast path declines, the code materialises the projection and invokes the ordinary PLE gate operation. Both paths produce the same 256-wide hidden vector for the next projection.

```
2021      const Tensor *G = L->pl_gate;
2022      if (T > 1 || !GPU(metal_matvec_gelu_mul(a->ph, a->xh, G->data,
          row_bytes(G), G->type, n, E, gj.pl))) {
2023          matmul(a->pg, a->xh, G, T, 0);  // (decode on the GPU: one step,
           h = gelu(xh · G)  pl)
2024          STEP(metal_pl_gate(a->ph, a->pg, gj.pl, T, n, gj.stride), T,
          pl_gate_row, &gj);
2025      }
```

Listing 13.5: `gemma4.c:2021–2025`

PREPRINT

After the 256-wide result is ready, `pl_proj` maps it back to the residual width. `add_norm` adds the normalised branch, applies the layer output scale, and can also prepare the next layer's normed input. This completes the branch shown in the opening diagram.

Why does PLE not add another vocabulary-sized GPU buffer?

A tempting implementation would upload the complete table once, then index it from a Metal kernel. That would make lookup convenient, but it would add almost 1.94 GB to the GPU-visible weight mapping. The engine instead treats the GGUF as a file-backed source and has the CPU resolve the ID to a row. Since the row is only 7,392 bytes, the transfer and conversion are small compared with making all vocabulary entries resident. The row is the only table data needed for the current input token.

The row is not reused across different token IDs, but its 42 slices are reused across the 42 layers for the same token. This is a useful form of locality: the CPU pays for one table lookup; the layer loop consumes its small activation piece by piece. If an input has repeated token IDs, the implementation still follows the per-token row path; it does not introduce a vocabulary-wide cache or change the model representation.

The model projection behaves differently from the table. It is a dense matrix with one set of weights shared by all token IDs, so the engine keeps it available as a GPU tensor and multiplies it by the ordinary embedding. The table carries token-by-layer values, while the projection maps an embedding into the entire layer-input space. Keeping these roles separate makes both the GGUF layout and the memory choice easier to understand.

How large is the PLE table in practical terms?

The GGUF table contains 2,818,572,288 values: 262,144 token rows times 10,752 values per row. The file encodes them in `Q5_K` blocks of 256 values and 176 bytes per block, for 1,937,768,448 bytes total (about 1.94 GB decimal).

The row cost is much smaller: 10,752 values make 42 blocks of 256 `Q5_K` values. At 176 bytes per block, that is 7,392 bytes per token row, or about 7.2 KiB. Each such row holds the 42 slices the model will use layer by layer; the engine does not reread one row for every layer. That is why an enormous parameter table can be affordable at decode time: only one compact row is fetched for a token, and it is fetched while the GPU has work to do.

The table is the largest PLE object, but it is not the only one. The shared BF16 model projection has shape `[2560, 10752]` and occupies 55,050,240 bytes. It produces all layers' projected contributions from the ordinary embedding in one matrix operation. Each layer also has two small per-layer matrices, stored F32 in the file and converted to F16 when loaded. Across 42 layers those 84 F32 tensors account for 220,200,960 bytes on disk; their F16 copies occupy 110,100,480 bytes. The projection and norm are small beside the token table, but their combined runtime storage still matters on a 16 GB machine.

What does one layer cost beyond the shared table?

The shared token table is large because it has one row per vocabulary entry and 10,752 columns. The layer-local PLE matrices are much smaller. A gate matrix maps 2,560 inputs to 256 outputs; a projection maps those 256 values back to 2,560. In each layer they hold 655,360 values apiece. Unlike the FFN matrices, which the GGUF stores quantised for this build, the PLE matrices are stored as F32 and then copied once to F16.

A single decode token reads the two F16 matrices for every layer as it moves through the model. The 42 layers' copies total about 110.1 MB; that is roughly 2.62 MB per layer. The corresponding F32 tensors in the file total 220.2 MB across all layers. This is why the loader performs a one-time conversion rather than leaving the GPU to read the larger format on every token. It also explains the loader's use of `pread`: it can make the compact shared copy without faulting all the original tensor pages into the normal file cache.

PREPRINT

The token row is smaller in traffic than the layer matrices as a whole. It takes 7,392 bytes once for each input token, not once for every layer. The prepared row slices are reused at their respective layer positions. On a decode step that means the CPU reads a few KiB from the file and the GPU reads the layer's small F16 gate and projection matrices along with its FFN and attention weights.

The output scale deserves a final pass through the dataflow. Without PLE, the FFN residual tail can apply the layer scale and prepare the next normalised input in one fused operation. With PLE, scaling there would change the value consumed by the PLE gate. The code therefore leaves the residual unscaled at the FFN tail, converts the current residual to the fp16 input expected by PLE, runs the PLE branch, and applies the scale at the final residual update. The next layer's norm can be prepared as part of that last step. A source listing is useful here because it makes the order explicit, while the shape table tells us the width of every crossing.

What stays on the GPU?

Can keeping a table off the GPU leave it waiting?

The token table remains in the mapped file and the CPU fetches one row. The model projection, the per-layer matrices, and the layer computation are available to Metal. The memory-saving choice is therefore selective: the table that would add about 1.94 GB is not wired as GPU weight memory, while the much smaller data needed for dense per-layer math is prepared for the GPU.

The runtime copies of the layer PLE matrices are F16. The loader reads the F32 source with `pread` rather than touching its mapped payload pages, requests `F_NOCACHE`, converts each value once, and puts the copy in shared storage usable by Metal. The original F32 pages need not enter the page cache. This halves the bytes the GPU reads per token from these 84 matrices, while retaining more accuracy than quantising them to `Q8_0`.

That conversion is implemented by `f16_copy` at `gemma4.c:904-924`. It returns the original tensor when it is not F32; otherwise it allocates a tensor descriptor and shared F16 destination, reads with `pread`, converts values, and points the descriptor at the copy. The loader applies it to both `inp_gate` and `proj` for each layer.

The PLE gate and projection matrices are used once per layer on each decode token, so the F16 copies trade about 110 MB of persistent memory for lower traffic than the original F32 form. The token table is different: mapping it and fetching a row does not require a 1.94 GB GPU buffer. The engine can overlap its CPU row lookup with the GPU's early work, leaving GPU utilization intact according to the project's documented memory design.

Experiment: inspect the shapes without inference

What does this exact GGUF say about one FFN and the PLE table? This is W1: a standard-library Python script reads the GGUF header, metadata and tensor descriptors. It stops after the tensor descriptors and does not read the payload region. It does not map the file, inspect tensor contents, start the engine, initialise Metal or run inference.

Run it from the book root, replacing the path if the model is stored elsewhere:

```
python3 code/ch13_gguf_probe.py "$MODEL"
```

The captured output below came from the local E4B `IQ4_XS` file. The script derives byte counts from the declared quantisation blocks and shape. For the PLE gate, its output uses the GGUF names `inp_gate.weight` and `proj.weight`; these are the per-layer matrices the loader assigns to `pl_gate` and `pl_proj`.

PREPRINT

Experiment W1

```
GGUF v3: 720 tensor descriptors; payloads not read
gemma4.block_count = 42
gemma4.embedding_length = 2560
gemma4.feed_forward_length = 10240
gemma4.embedding_length_per_layer_input = 256
layer-0 FFN and PLE descriptors:
blk.0.ffn_gate.weight: IQ4_XS ne=[2560, 10240] bytes=13926400
blk.0.ffn_up.weight: IQ4_XS ne=[2560, 10240] bytes=13926400
blk.0.ffn_down.weight: IQ4_XS ne=[10240, 2560] bytes=13926400
per_layer_token_embd.weight: Q5_K ne=[10752, 262144] bytes=1937768448
per_layer_model_proj.weight: BF16 ne=[2560, 10752] bytes=55050240
per_layer_proj_norm.weight: F32 ne=[256] bytes=1024
blk.0.inp_gate.weight: F32 ne=[2560, 256] bytes=2621440
blk.0.proj.weight: F32 ne=[256, 2560] bytes=2621440
```

The tensor directory identifies three layer-zero FFN matrices at 13,926,400 bytes each, and two F32 PLE matrices at 2,621,440 bytes each. It also shows the full token table and shared projection shapes. These are file descriptors and format-derived byte counts, not a report of resident or wired memory. The script is at `code/ch13_gguf_probe.py`; its full output is saved as `code/expected/ch13-w1.txt`.

PREPRINT

Experiment: measure the PLE step

The source can skip selected Metal kernels through `GEMMA4_SKIP`. Skipping a kernel deliberately makes the model output wrong, so this is only a timing probe. The following W2 comparison needs the model and a quiet GPU. Run the baseline and then the modified path with the same short prompt and token count, recording the printed wired-weight banner and end-to-end decode timing. The second command omits the PLE gate kernel; it does not remove the token table or its CPU read. Do not treat the changed text or logits as an accuracy result.

```
nice -n 10 code/build/gemma4 -m "$MODEL" \
  -p "A small example" -n 1
GEMMA4_SKIP=pl_gate nice -n 10 code/build/gemma4 -m "$MODEL" \
  -p "A small example" -n 1
```

The project outline mentions a roughly 1.9 ms PLE cost from a different engine. That is context only and is not a result for this snapshot. Try it on your machine (W2): compare baseline and PLE-gate-skipped decode in a scratch copy. Inspect per-token timing and the wired banner; skipping the kernel predicts lower measured time but invalid output. The roughly 1.9 ms figure belongs to a different engine and is not a reference for this snapshot. The table's 1.94 GB is a GGUF tensor size; the source does not provide a switch that wires or unwires it, so the memory saving is established from the selected mapping logic and file size, not a paired runtime toggle.

Pitfalls from the project history

Pitfall

The small matrices looked like a quantisation opportunity. The project's README records that the 84 `inp_gate` and `proj` matrices are F32 in the GGUF and total about 220 MB. Converting them once to F16 at load halves their per-token read traffic. An experiment with Q8_0 reduced those bytes by half again and made the engine about 2% faster, but mean KL divergence against llama.cpp rose from 0.00001 to 0.001 and the top token changed 0.7% of the time. The fix retained F16 copies. Source: snapshot `README.md`, "Speed and memory" and "Things that did not work"; the loader's `f16_copy` comment at `gemma4.c:904-906` records the same result.

A source-order check

The output scale is easy to put in the wrong place. The FFN residual update normally applies the layer scale. When PLE is active, the code sets the FFN update scale to 1, then `pl_block` performs the residual addition and output scale after its own projection and norm. Moving the scale earlier changes the residual that the PLE gate reads, even if the FFN branch itself still looks plausible. Source: the comments and `fj.out_scale` selection at `gemma4.c:1914-1920`, followed by the `add_norm` in `pl_block` at `gemma4.c:2030-2031`. This is a source-order check from the implementation, not a reported historical failure.

Exercises

Exercises

1. **Read.** For each of the three FFN matrices in the W1 table, state the input width and output width. Explain why the down matrix has its dimensions reversed relative to gate and up.
2. **Read.** Trace a single token's PLE table row. How many bytes does the row occupy, and how many 256-value layer slices does the engine obtain from it?
3. **Change.** Modify the W1 probe to report the combined on-disk bytes of all 84 layer PLE matrices, then calculate the size of their F16 copies. Keep it metadata-only.
4. **Change.** In a scratch copy of the snapshot, make the CPU GeGLU path write the gate output to a separate diagnostic array before multiplying by the up output. Compare the resulting hidden vector with the original loop on a small synthetic input.
5. **Measure.** As a W2 reader experiment, compare a normal decode with a scratch build that skips PLE. Record per-token timing and wired memory, and state that the skip changes the model output. Do not use the different engine's historical 1.9 ms estimate as a result for this snapshot.

PREPRINT

Rebuild checkpoint

Rebuild checkpoint

Milestone m6a: one layer's FFN and PLE, not a full forward pass. Implement `steps/m6a_ffn_ple.c` for layer zero and one token. Start with the CPU reference: read the relevant tensor rows, perform gate and up projections, GELU and elementwise multiplication, down projection and post-FFN residual update; then prepare the token's PLE input and run the PLE gate, projection, post norm, residual and output scale. Use the model's GGUF dimensions and exact tensor types. Compare the saved hidden vector at each branch boundary with an engine dump from the same token and weights. The acceptance check is agreement at the FFN output and after PLE within the chosen fp16 rounding tolerance; a matching final top token alone is too weak to diagnose a branch error.

PREPRINT

Chapter 14

One layer, then the whole model

The question

What happens, in order, to one token's vector, and which parts of that work can the engine avoid?

The idea

Can we follow one token without losing the surrounding rows?

Start with a concrete scale: the E4B text model has 42 layers, each carrying a 2,560-number residual vector for every token position. A matrix multiply changes one branch of that vector; a residual add puts the branch result back into the running vector. The next layer receives the updated vector. That running vector is the residual stream, and keeping track of it is the easiest way to read the forward pass.

For ordinary text, the first vector is a row from the token embedding table, scaled by the model's embedding scale. The model also prepares per-layer inputs (PLE): one row from a large token table and a projection of the ordinary embedding together make 42 slices of 256 values. The small slice for layer l waits in an activation buffer until that layer needs it. PLE is not an alternative residual stream; it is a learned side branch that joins the main stream near the end of each layer.

The diagram is deliberately one full page. It names the host calls that dispatch the GPU work and labels the important activation shapes. A box can represent several Metal kernels: for example, `glu` is one fused decode kernel for gate, up, GELU and multiply, while its prefill path is a matrix kernel over multiple rows. The arrows describe dependencies, not a claim that every arrow needs a separate command buffer.

PREPRINT

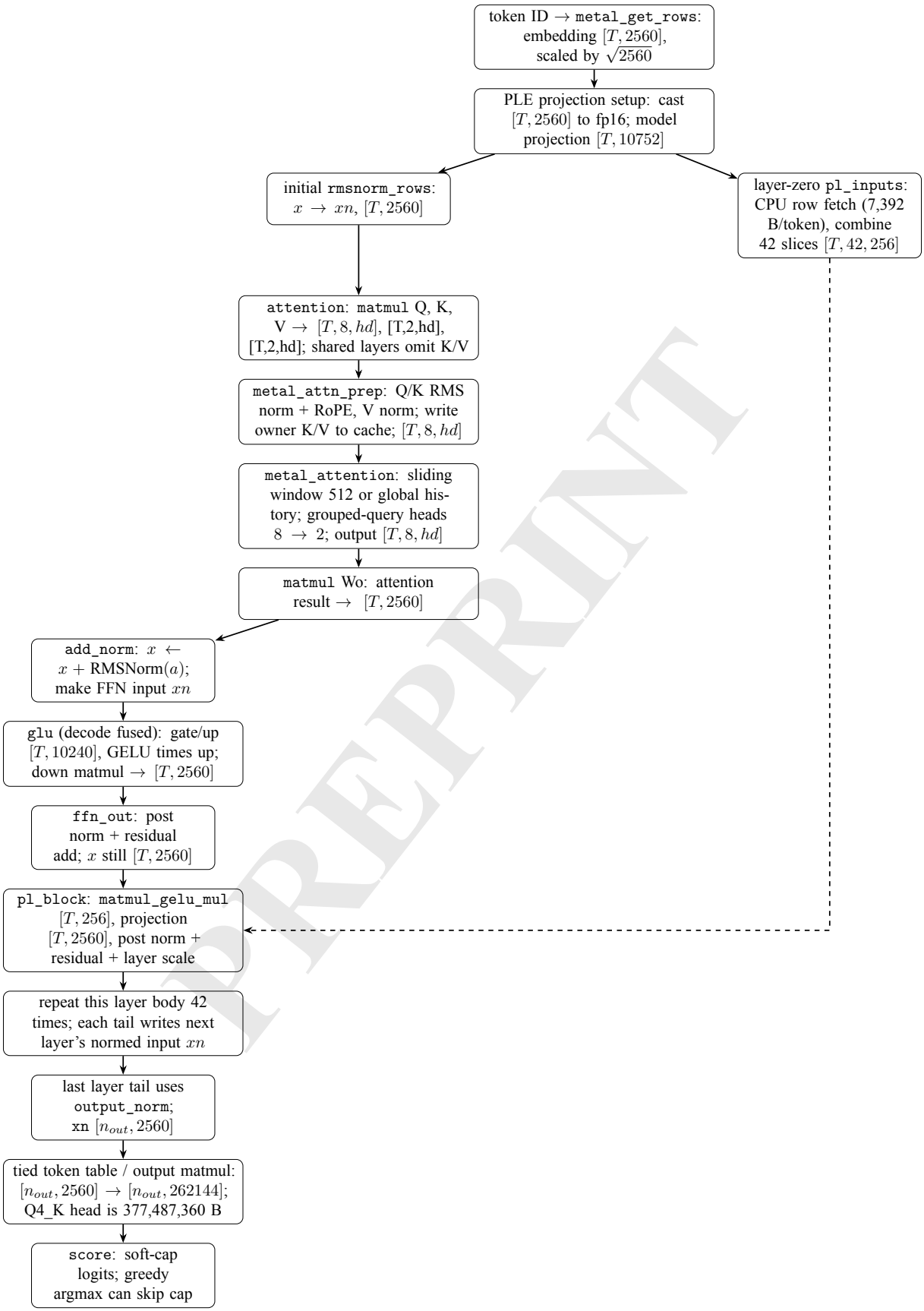


Figure 14.1. One token's route through a layer and the complete text forward. Here T is the current row count, $hd = 256$ for sliding layers and 512 for global layers. The prepared PLE slice joins only at that layer's PLE block. For a one-token decode, every $[T, \ldots]$ activation has $T = 1$.

PREPRINT

Where do the norms sit?

There are three useful boundaries. First, the embedding is RMS-normalised with layer zero's attention norm before its query projection. Each layer then has an attention branch: project Q/K/V from the normed input, prepare the heads and cache entries, run attention, and project the concatenated result back to width 2,560. The output projection is a branch value, not the residual itself.

The attention branch returns to the stream at `add_norm`. In plain words, the helper normalises the branch output, multiplies it by a learned post-attention vector, and adds it to the existing stream. The same helper also prepares the next normed input using the FFN norm weight. This is why the attention output can be added to `x` while the FFN reads `xn`; those buffers hold different stages of the same position, not separate model streams.

The FFN computes a branch and adds it after its own learned post norm. PLE then computes another branch and performs a post norm plus residual add. The layer output scale is applied at this PLE tail for this model. That scale is not an arbitrary factor folded into the first norm: source comments in `ffn` explain why the PLE case defers it until the PLE branch has joined. At the end of the final layer the tail prepares `xn` with `output_norm`, so the output projection receives the final normalised stream without a separate extra norm pass.

This ordering is easiest to verify in the host loop: `attention`; attention `add_norm`; `ffn`; optional `pl_block`; submit. The equations for RMS norm belong to Chapter 5; here the important thing is identifying which value is normalised and where it goes. The same layer skeleton also has a CPU implementation. Each `STEP` tries its Metal operation and runs the corresponding C row task if the GPU declines; the matrix wrappers similarly preserve the CPU reference.

What changes between sliding and global attention?

The attention sequence is stable, but its head width and visible history change. E4B has eight query heads and two K/V heads. A sliding layer uses head size 256 and looks back over a 512-position window; a global layer uses head size 512 and can read the full prior sequence. The metadata pattern marks global layers 5, 11, 17, 23, 29, 35 and 41 (zero based). The first 24 layers own caches: 20 sliding and 4 global. The final 18 use one of those earlier caches.

For an owner layer, Q/K/V projections are made from the current normed row. `attn_prep` normalises Q and K, rotates them for position, normalises V, and stores the new K and V at the cache row for that position. The attention kernel reads the proper window and produces one output per query head. For a shared layer, the query and output projection still run, and attention still reads the matching owner's K/V cache. Its own K/V projections and cache writes are omitted. Sharing changes which keys and values the query consults; it does not remove the layer's attention computation.

At prefill time the first 24 layers have to process all prompt rows because they produce cache entries that later decode steps may need. Once the loop reaches layer 24, remaining layers write no new cache. If the call asks for just the last row's logits, earlier rows cannot affect that requested result any more. The engine gathers the chosen residual, normed input, and PLE rows down into the front of their arrays and runs layers 24–41 on that smaller batch. For a 512-row prefill and one output row, it avoids 511 rows of work in those 18 layers. A bidirectional media path cannot use this shortcut because later prompt rows can affect earlier rows.

Why must the first 24 layers keep every row?

Imagine a prompt whose positions are numbered 0 through 511. In the first cache-owning layer, position 511's attention can read earlier keys and values, while the layer writes K and V for all 512 positions. The next generated token will need the cache entries for the prompt's visible positions. If the engine discarded rows 0 through 510 before those writes, a later decode step would not have the same context. The same is true through layer 23: each layer owns a distinct cache stream, and its prompt rows are persistent state for future calls.

PREPRINT

At layer 24 the situation changes. The layer's query still runs for each row that is needed, but its K/V pointer refers to an earlier layer's cache. It does not create a new cache record. In a causal sequence, an earlier position cannot affect the hidden state at a later position: attention only reads keys at or before the query's position. Therefore the last prompt row's result depends on its own residual path and earlier rows' cached K/V, not on the later-layer residual vectors for those earlier positions. If only the final logits are requested, the engine can gather that final row and continue with one row. If it wants logits for several positions, it gathers those positions instead.

The guard is deliberately narrower than "T is greater than one." It checks that the loop has reached exactly the first shared layer, that there are more rows than requested outputs, that the selected range is already at or beyond the output prefix, and that the call is not bidirectional. The in-place gather copies rows downward, so the source comment requires `first >= n_out`: destination rows must not overwrite a source that a later copy still needs. Once the copy is done, the host changes `Tl` to `n_out`, advances the first position by `first`, and resets the selected row offset to zero.

The arithmetic shows why the condition can matter without promising a speed number. For 512 input rows and one requested final row, the first 24 layers still process 12,288 layer-row pairs. The last 18 layers process 18 rows' worth of layer work after the gather instead of 9,216. Their matrix weights remain necessary, because that one surviving row still passes through every layer. What disappears is repeated work over the 511 unrequested positions in those layers. The saved amount depends on prompt length and requested output rows; the layer weights are still streamed for the surviving row. The outline records a historical prefill change from about 260 to about 440 tokens per second, but that is a past project result, not a measurement from this W0/W1 chapter.

Why does the call carry both `first` and `n_out`?

The input rows and the rows whose logits are needed are not always the same set. A call can process a prompt chunk to fill attention caches, yet ask for logits only at the chunk's end. In that case `T` is the number of rows entering the layer loop, while `first` and `n_out` select the output interval. After the gather, the code shifts the selected positions to row zero and resets `first`; the final matmul can then read the compacted interval as a contiguous block.

When every input position needs logits, as in a scoring pass, `n_out` can equal T. The gather condition requires `Tl > n_out`, so the loop keeps all rows. A chunked prefill may also make several calls: an earlier chunk fills the cache, then later calls continue from its past position. The forward function's `pos` begins at `st->n_past`; when the row gather drops a prefix, it advances by `first` so that the surviving rows still carry the correct absolute positions into RoPE and cache addressing.

For a single decode token, `T`, `first`, and `n_out` are typically all one, so no row reduction is needed. The same outer loop handles both decode and prefill, and the smaller gather case sits at a single safe boundary. This keeps cache-building rules, output selection and layer execution in one routine without making every caller allocate a second model implementation. To debug a surprising shape, print the three counts at the call site and again at layer 24: if `Tl` changes, the source has taken the row-selection path; if it does not, the later layers are processing all rows.

The kernels and their traffic

What runs inside one layer?

The call `attention()` in the source is a compact schedule over familiar operations. It enters a GPU group for Q/K/V matmuls; the K and V calls are conditional on this layer owning its cache. It leaves that group before `metal_attn_prep` and `metal_attention`, then computes the output projection. Chapter 10 explains the Q/K/V values and RoPE, Chapter 11 the cache layout, and Chapter 12 the attention kernels. Here we can read them as one chain: normed $[1, 2560]$ input, Q/K/V projections, prepared heads and cache, attended heads, then an output branch of width 2,560.

PREPRINT

The next helper fuses the attention residual boundary with production of the FFN input. In the C reference these are row operations; on Metal the matching operation can do the residual and norm together. The FFN's gate and up projections both consume $[1, 2560]$. A fused decode `glu` makes the 10,240-value hidden row after GELU and elementwise multiplication, then the down matmul returns $[1, 2560]$. The FFN output helper applies its post norm and adds to the stream.

The PLE path has already prepared a 256-wide slice per layer for this token. At layer l, the gate matrix maps the fp16 residual $[1, 2560]$ to $[1, 256]$; GELU and multiply combine it with that layer's prepared slice. The projection maps the result back to $[1, 2560]$, and the PLE post norm, residual add and output scale finish the layer. In decode, the gate matvec, GELU and multiply may be one `metal_matvec_gelu_mul`; the source falls back to matmul plus `metal_pl_gate` for other shapes. These dispatch choices preserve the same data flow.

Can we audit the shape at every boundary?

For one decode token, set $T = 1$ and read each leading dimension as one row. The head-width symbol is 256 in a sliding layer and 512 in a global layer. Since there are eight query heads and two K/V heads, the flattened Q width is 2,048 or 4,096, while flattened K and V each have width 512 or 1,024. The table walks from the normed residual through the exact host calls. It names activation shapes, so it remains meaningful even when one call is implemented by multiple Metal dispatches.

Step in source	Input for one token	Output / next consumer
Embedding lookup	ID; token row in Q4_K	`x` $[1, 2560]$, scaled
Cast + model projection	`x`, then fp16 `xh` $[1, 2560]$	PLE projection $[1, 10752] = [1, 42, 256]$
Initial RMS norm	`x` and layer-zero attention norm	`xn` $[1, 2560]$ to Q/K/V
Q/K/V matmuls	`xn`; Q $[2560, 2048/4096]$, owner K/V $[2560, 512/1024]$	Q, K, V projection rows
Attention preparation	Q/K/V rows, head norms, position, cache row	fp16 Q and owner cache writes; K/V skipped if shared
Attention	Q plus owner cache K/V (or shared source cache)	attended heads $[1, 2048/4096]$
Output projection	attended heads and W_o	attention branch `a` $[1, 2560]$
Attention residual + norm	`x`, `a`, post and FFN norms	updated `x`; FFN input `xn` $[1, 2560]$
FFN gate/up + down	`xn`; three matrices	gated hidden $[1, 10240]$, down branch $[1, 2560]$
FFN output update	residual `x`, down branch, post norm	updated residual `x` $[1, 2560]$
PLE gate + projection	fp16 residual, slice $[1, 256]$, two matrices	PLE branch $[1, 2560]$, then scaled residual `xn`

There are two details a shape table can hide. First, Q and K have been normalised and RoPE-rotated before attention sees them; a shared layer's K/V come from the earlier cache in their already-prepared form. Second, the PLE projection and token table are laid out for every layer, but the per-layer operation consumes only the slice at offset $l \times 256$ in that prepared row. That is why the input has width 10,752 before the loop but the gate at a single layer has width 256.

The prefill version uses the same logical widths with T rows. A `matmul` can switch from a one-row matvec kernel to a tiled matrix kernel; attention can switch from the decode split-K path to the prefill flash path. Residual and normalization kernels also process T rows. The layer widths and the order of dependencies do not change. Only the row count, cache range, and selected kernel implementations do.

What does the command-buffer timeline look like?

The C loop does not wait after every kernel. A Metal wrapper encodes a dispatch into the current concurrent compute encoder. When it creates a new encoder, the host has a command buffer ready; dependent steps get a buffer memory barrier between their writes and reads. The exception is a `gpu_group`: while

PREPRINT

the group is open, independent projections can be encoded without a barrier between them. The attention helper uses this around Q, K and V projections because all three read the same normalised input and write separate outputs. A shared layer simply has fewer calls in that group.

After the group closes, preparation must see the projection outputs, attention must see prepared Q and cache data, and the output projection must see the attention result. Those are dependent steps and the encoder inserts the necessary ordering. The same principle applies to the residual and FFN operations. The C source records operations in order; the Metal dispatch layer enforces the write-before-read edges. It does not create one command buffer for every box in the diagram.

At the bottom of each layer iteration, `gpu_submit()` commits the encoded work without waiting for completion. The queue runs submitted command buffers in order. The CPU can then encode the next layer while the GPU works on the current one. A call to `gpu_sync()` is stronger: it waits for all outstanding work and is used when the CPU fallback must read a result or a resource must be reused. The GPU timing counter adds elapsed GPU time from completed command buffers; it is not wall time and does not include the CPU encoding time.

The PLE row lookup is the notable CPU dependency. The token ID is needed by the CPU to fetch the per-layer table row. For PLE models, the start of `forward` submits prior work and records a fence. When layer zero reaches `pl_inputs`, it submits the projection and waits for that earlier fence before the CPU reads the token table. This permits already-submitted work to run during the wait. It also protects the token buffer from being overwritten while the GPU still reads it. The later PLE block is ordinary queued GPU work and does not force a full synchronization at each layer.

There is a similar distinction inside FFN dispatch. In a dense layer, gate/up and down operations can be encoded in order with the residual output after them. In an MoE model the router logits must return to the CPU to choose experts, so the code submits and fences around routing. That path creates a real dependency and can pause the CPU. E4B's text layers are dense, so the long dense chain is the forward path this chapter follows. This explains why one source loop can support both the E4B and the 26B model while their command timelines differ.

The intended mental picture is therefore a conveyor, not forty-two synchronous trips to the GPU. Host code describes layer L, submits it, and starts encoding L+1; Metal's queue preserves order while barriers preserve dependencies within each encoded sequence. The GPU may overlap independent dispatches only where the host has declared that they do not depend on each other. That rule is the reason the Q/K/V group is explicit and why the chapter's boxes should be read as a dependency map rather than as a stopwatch profile.

Which operations disappear into a fused kernel?

Fusion changes how many times an intermediate is written; it does not change the mathematical stages. For a one-row decode, Q/K/V projections are independent because they read the same `xn`; the group lets the GPU schedule their work together. The attention-preparation kernel then combines query and key normalization with RoPE, normalizes values, and writes the owner cache. This saves a separate launch for each small head operation. The attention kernel still has its own job because it must read a position-dependent amount of cache and reduce over keys. The output projection remains a matrix operation after attention.

The FFN has a stronger fusion opportunity. Separate gate and up matmuls would create two 10,240-value arrays, then an activation and elementwise multiply would read them again. The fused decode path keeps both accumulations until it can write the gated hidden row. The down projection still consumes that row as its own operation, and the output helper still needs to merge the branch with the residual. PLE repeats the pattern at width 256: for decode the projection may fuse its gate matvec with GELU and multiply, but the return projection and residual update remain dependent work afterward.

Prefill shifts the balance. A matrix kernel can reuse quantised weight blocks across several input rows, so a decode-specific single-row matvec fusion is not always the best schedule. The same `glu` wrapper can materialise or defer the norm and select an appropriate GPU route based on shape and support. The C fallback is intentionally more explicit and may use scratch arrays. These differences explain why a

PREPRINT

kernel name should be read with its call site: a fused decode implementation is not a universal promise that every prompt batch uses the same kernel.

There is also a useful limit to the word "fused." The residual stream cannot accept the FFN branch until the down projection has finished; the PLE branch cannot finish until its input slice and projection result exist; the final head cannot start until the last residual update and output norm are available. Those are real dependencies. Fusion can remove a temporary array or a launch between operations whose inputs are already at hand, but it cannot remove the chain of information from the token ID to the logits.

How many bytes of weights are read for one token?

The useful first estimate is not a speed result: it is the amount of encoded weight data the decode path must visit. The W1 probe in Chapter 13 establishes the FFN and PLE tensor sizes. This chapter's W1 descriptor inspection adds Q/K/V/output shapes for the two attention head widths. The table sums matrix weights that are used once for one token. It excludes small norm vectors, activations, cache reads, and cache writes, so it is a lower-bound inventory of layer weight bytes, not a complete memory-traffic measurement.

For a sliding attention owner, Q, K, V and output weights total 7,168,000 bytes: 2,785,280 + 696,320 + 901,120 + 2,785,280. A global owner doubles each head width, so its attention matrices total 14,336,000 bytes. The three FFN matrices contribute 41,779,200 bytes in either type. The two PLE matrices are stored F32 in the file but loaded as F16 copies; those active copies total 2,621,440 bytes for a layer. In shared layers, the K and V projection matrices are skipped, leaving Q plus output attention weights.

Layer kind	Count	Attention matrices	FFN + PLE matrices	Total / token
Sliding owner (cache writer)	20	7,168,000	44,400,640	51,568,640 B
Global owner (cache writer)	4	14,336,000	44,400,640	58,736,640 B
Sliding shared (query only)	15	5,570,560	44,400,640	49,971,200 B
Global shared (query only)	3	11,141,120	44,400,640	55,541,760 B

The count is split by both attention kind and cache ownership. There are 35 sliding layers in all, but only 20 own K/V; three of the seven global layers share K/V. The shared rows still include attention's query and output matrices, the FFN, and PLE. The per-layer token table is not in these totals: it is 1,937,768,448 bytes on disk, but one selected row is read once for the token and contains all 42 PLE slices. Chapter 13 measured that row's encoded size as 7,392 bytes. The embedding and LM head are also outside the layer rows.

Summing the four rows gives 2,182,512,640 bytes, about 2.18 GB, of layer matrix weights visited across one decode token (decimal bytes; norms and cache traffic are omitted). Add the 377,487,360-byte Q4_K output matrix, token embedding row, and selected PLE row to understand why the practical total is higher. The 1.94 GB table is not read in full for one token. The weights are file-backed, quantised blocks are decoded as the matvec proceeds, and the GPU streams them from unified memory. Chapter 7's bandwidth rule can turn a known byte count into a ceiling only when paired with an actual sustained-bandwidth measurement; this W1 chapter does not claim a token rate.

Why is the PLE table huge but the row small?

The table has 262,144 rows and 10,752 Q5_K values per row. One row holds 42 groups of 256 values. The row is not fetched once per layer: the CPU reads and expands the selected row, scales it, and the later layer loop uses the appropriate slice. Its 7,392 bytes are modest next to a layer's matrix weights. The full table stays in the mapped GGUF rather than being wired into the Metal weight map. Chapter 3 covers the mapping decision; the key point for this schedule is that the token lookup happens once before the 42-layer loop.

The shared model projection is another setup operation. The ordinary embedding is cast to fp16, multiplied by the BF16 projection into 10,752 values, and combined with the token-table slices by `pl_inputs`. This work is not repeated 42 times. Each layer later consumes one 256-wide slice. When T is small, the CPU must wait for previously submitted GPU work before it reads token IDs for the table; it does

PREPRINT

that while the GPU can continue the already-submitted projection and layer-zero work. The fence in `pl_inputs` is a coordination point, not evidence that every layer runs synchronously on the CPU.

The real code

Where does the forward pass start?

The first excerpt covers input validation, any expert prefetch, the ordinary embedding lookup, the PLE projection, and the first input norm. The branch for supplied image rows is visible because `forward` also serves multimodal callers, but a text token uses `m->tok_embd`. Each `STEP` first offers the matching Metal call and has a plain-C row callback ready if it declines. The excerpt is 35 lines from the frozen source snapshot.

```
2054 static void forward(const Model *m, Acts *a, State *st, int T, int first
         , int n_out, float *logits) {
2055     if (T > a->T_max || st->n_past + T > st->n_ctx) die("forward: %d
         tokens don't fit", T);
2056     if (st->ring < st->n_ctx && (T > st->ring - m->n_swa - 8 ||
         state_keep(m, st, st->n_past) != st->n_past))
2057         die("forward: the sliding-window cache can't hold this call");
2058     if (st->n_past == 0) st->n_high = 0;
2059     const int E = m->n_embd;
2060     void *prev = NULL;  // per-layer embeddings: all GPU work before
         this call (see pl_inputs)
2061     if (m->n_pl) {
2062         gpu_submit();
2063         prev = gpu_fence();
2064     }
2065     if (xcache && T >= PREFETCH_T) {  // a prefill: start reading the
         first MoE layer's missing experts
2066         gpu_sync();                   // (the staging slots may still be
          in use by the previous call)
2067         for (int l = 0; l < m->n_layer; l++)
2068             if (m->layers[l].router) { cache_prefetch(xcache, l); break;
          }
2069     }
2070     if (a->embd) {  // input rows given (an image): row t of a->embd,
         unscaled; a->tokens are the row indices
2071         Tensor rows = {.type = T_F32, .ne = {E, T, 1, 1}, .data = (const
          uint8_t *)a->embd};
2072         gpu_sync();  // (the GPU may not have read the previous call's
         tokens yet)
2073         for (int t = 0; t < T; t++) a->tokens[t] = t;
2074         EmbedJob ej = {&rows, a->tokens, a->x, 1.0f};
2075         STEP(metal_get_rows(a->x, a->embd, E * sizeof(float), T_F32, a->
         tokens, T, E, 1.0f), T, embed_row, &ej);
2076     } else {
2077         const Tensor *emb = m->tok_embd;
2078         EmbedJob ej = {emb, a->tokens, a->x, m->embd_scale};
2079         STEP(metal_get_rows(a->x, emb->data, row_bytes(emb), emb->type,
         a->tokens, T, E, m->embd_scale), T, embed_row, &ej);
2080     }
2081     if (m->n_pl) {  // the projection part of the per-layer inputs, from
          the embeddings
2082         CastJob cj = {a->xh, a->x, E};
2083         STEP(metal_cast_half(a->xh, a->x, T * E), T, cast_row, &cj);
2084         matmul(a->pl, a->xh, m->pl_model_proj, T, 0);
```

```
2085     }
2086
2087     // a->xn: the normed input of the next step. Each layer's last step
         makes the next layer's (and the last
2088     // layer's the output norm's, of all T rows), and its attention's
         add_norm makes the FFN's.
2089     rmsnorm_rows(a->xn, a->x, m->layers[0].attn_norm, T, E, m->eps, 0);
```

Listing 14.1: gemma4.c:2054–2089

Where do the 42 iterations change shape?

This second piece is the important loop. `Tl` begins as T. At `l == m->n_kv`, the conditional gather reduces it to the requested output rows, provided the call is causal and has more rows than it needs. Then each iteration calls attention, joins that branch to the residual and makes the FFN input, runs FFN, optionally runs PLE, and submits the work. Notice that `next_norm` is the next layer's norm, or the output norm for the final layer. This lets the current tail write the next step's normalised input.

```
2089     rmsnorm_rows(a->xn, a->x, m->layers[0].attn_norm, T, E, m->eps, 0);
2090     int Tl = T, pos = st->n_past;  // the layers' rows, and the position
         of the first
2091     for (int l = 0; l < m->n_layer; l++) {
2092         const Layer *L = &m->layers[l];
2093         const float *next_norm = l + 1 < m->n_layer ? m->layers[l + 1].
         attn_norm : m->output_norm;
2094         if (l == m->n_kv && Tl > n_out && first >= n_out && !a->bidir) {
2095             // The layers from here on write no KV cache, so the rows
         whose logits aren't wanted affect nothing
2096             // any more: only rows [first, first + n_out) go on (in a
         prefill, just the last). (Rows are moved
2097             // down, which needs first >= n_out so that no row is
         overwritten before it is read.)
2098             for (int t = 0; t < n_out; t++) a->rows[t] = first + t;
2099             const int P = m->n_layer * m->n_pl;
2100             Tensor xr = {.type = T_F32, .ne = {E, T, 1, 1}, .data = (
         const uint8_t *)a->x};
2101             Tensor pr = {.type = T_F32, .ne = {P, T, 1, 1}, .data = (
         const uint8_t *)a->pl};
2102             EmbedJob xj = {&xr, a->rows, a->x, 1.0f}, pj = {&pr, a->rows
         , a->pl, 1.0f};
2103             GatherJob gj = {a->xn, a->xn, a->rows, E};
2104             STEP(metal_get_rows(a->x, a->x, E * sizeof(float), T_F32, a
         ->rows, n_out, E, 1.0f), n_out, embed_row, &xj);
2105             STEP(metal_gather(a->xn, a->xn, a->rows, n_out, E), n_out,
         gather_row, &gj);
2106             if (P) STEP(metal_get_rows(a->pl, a->pl, P * sizeof(float),
         T_F32, a->rows, n_out, P, 1.0f), n_out, embed_row, &pj);
2107             Tl = n_out, pos += first, first = 0;
2108         }
2109         attention(m, L, l, a, st, Tl, pos);
2110         add_norm(a->x, a->a, L->post_attn_norm, Tl, E, m->eps, 1.0f, a->
         xn, L->ffn_norm);
2111         ffn(m, L, a, Tl, next_norm);
2112         if (m->n_pl) {
2113             if (l == 0) pl_inputs(m, a, Tl, prev);  // (the GPU runs
         layer 0 meanwhile)
2114             pl_block(m, L, l, a, Tl, next_norm);
2115         }
```

```
2116          gpu_submit();  // start the GPU on this layer while the next is
      encoded
2117      }
```

Listing 14.2: gemma4.c:2089–2117

What does the loop produce?

The final excerpt shows the layer tail and the output matrix. The matmul receives a->xn, already normalised by the final layer tail, and writes the requested logits. The update to cache position happens after the GPU work has been encoded; submission and synchronization are handled by the queue helpers described in Chapter 6.

```
2111          ffn(m, L, a, Tl, next_norm);
2112          if (m->n_pl) {
2113              if (l == 0) pl_inputs(m, a, Tl, prev);  // (the GPU runs
      layer 0 meanwhile)
2114              pl_block(m, L, l, a, Tl, next_norm);
2115          }
2116          gpu_submit();  // start the GPU on this layer while the next is
      encoded
2117      }
2118
2119      matmul(logits, a->xn + (size_t)first * E, m->output, n_out, 0);
2120      st->n_past += T;
2121      if (st->n_past > st->n_high) st->n_high = st->n_past;
2122  }
```

Listing 14.3: gemma4.c:2111–2122

Where does the CPU/GPU boundary sit in the FFN?

The router submission above this excerpt is conditional and applies to the MoE model; E4B's dense text layers take the shared MLP path directly. The excerpt begins at that path and continues through the optional expert wait and output/residual helper. glu itself chooses a fused Metal kernel where supported and otherwise follows the CPU reference. It connects this chapter's loop to Chapter 13's individual FFN operations.

```
1903      glu(a->h, a->xn, L->ffn_gate, L->ffn_up, T, 0, NULL);
1904      matmul(a->b, a->h, L->ffn_down, T, 0);
1905      if (L->router) {
1906          rmsnorm_rows(a->xn2, a->x, L->pre_ffn_norm_2, T, E, m->eps, 1);
1907          gpu_submit();
1908          const double t0 = now();
1909          gpu_wait(routed);
1910          t_wait_gpu += now() - t0;
1911          route(m, L, a, T);
1912          experts(m, L, a, T);
1913      }
1914      // with per-layer embeddings, the layer's output scale comes after
      their block, which reads x in fp16
1915      FfnOutJob fj = {m, L, a, m->n_pl ? 1.0f : L->out_scale, m->n_pl ? a
      ->xh : next_norm ? a->xn : NULL,
1916                      m->n_pl ? NULL : next_norm};
1917      if (fj.out == deferred_norm.out) deferred_norm.out = NULL;
1918      STEP(metal_ffn_out(a->x, fj.out, fj.next_w, a->b, a->ye, a->pos, a->
      cw, L->router ? L->post_ffn_norm_1 : NULL,
```

```
1919                            L->post_ffn_norm_2, L->post_ffn_norm, fj.
       out_scale, T, E, m->n_expert_used, m->eps),
1920            T, ffn_out_row, &fj);
1921    }
```

Listing 14.4: gemma4.c:1903–1921

Where does the final norm actually run?

The last norm is easy to look for in the wrong place. There is no standalone normalization call after the loop. The loop computes `next_norm` as the next layer's attention norm, except in layer 41 where it points to `m->output_norm`. The layer's tail writes the final `xn` with that weight. Then the output matrix reads `xn`. On a prefill that has kept only one output row, the matrix reads the selected row; on a decode call it reads the single row directly.

The table in the W1 experiment shows that `output_norm.weight` is a 2,560-element F32 vector, or 10,240 bytes. The output projection has a very different scale: 262,144 vocabulary rows by 2,560 inputs. In this file there is no separate `output.weight`; the loader sets the output matrix pointer to the token embedding matrix when that tensor is absent. That tied head is Q4_K and occupies 377,487,360 bytes. It is a large read for each decoded token, but it is not the 1.94 GB PLE token table. The latter has 10,752 columns (42 slices of 256) and serves the PLE branches; it is a separate GGUF tensor.

This distinction is useful when reading a memory report. The word "token table" can refer to the ordinary token embedding used for lookup and the tied output projection, or to the per-layer PLE table. The ordinary embedding/head is about 377 MB in this quantisation. The per-layer table is about 1.94 GB and is CPU-read by one selected row at a time. They have different shapes, formats and call sites even though both are indexed by token identity.

How does the score path cap the logits?

`forward` deliberately writes raw output-matrix values. The W1 metadata reports a final logit soft-cap value of 30.0. The scoring path applies that monotonic tanh cap before it computes probability statistics; the greedy argmax path only needs the ordering and can skip this transform. These C lines show the cap applied in the fallback scorer. The Metal scorer receives the same cap value as a parameter.

```
3721        if (job->softcap > 0)
3722            for (int j = 0; j < V; j++) lg[j] = job->softcap * tanhf(lg[j] /
        job->softcap);
3723
3724        float mx = lg[0], mn = lg[0];
```

Listing 14.5: gemma4.c:3721–3724

Experiment

Can we check the source build without loading weights?

This is W0. From the book root, build only the frozen snapshot and ask the resulting executable for help. The build goes under `code/build`; the help path exits before a model is opened. It checks the C and Metal sources compile and confirms that the command-line tool starts. It does not establish that the model's weights, output, or speed are correct.

```
nice -n 10 sh code/snapshot/build.sh
code/build/gemma4 --help
```

PREPRINT

The recorded W0 output is in `code/expected/ch14-w0-help.txt`. Inspect the usage text and the reported successful build. If the executable is named differently on a particular build, use the path printed by the build script; do not substitute the live engine binary.

Can we verify the dimensions without inference?

This is W1. The probe reads the GGUF header, metadata and tensor descriptors, then stops before the payload. For this chapter, the additional selected descriptors are the layer-zero Q, K, V and output matrices and the tied token table. To repeat the check, run the standard-library probe with the local E4B file:

```
python3 code/ch14_forward_probe.py "$MODEL"
```

The probe prints the FFN, PLE and global shapes, then the attention descriptors for layers 0 and 5. The captured read reports 42 layers, width 2,560, FFN 10,240, PLE width 256; layer-zero Q/K/V/output bytes are 2,785,280 / 696,320 / 901,120 / 2,785,280. Layer five reports 5,570,560 / 1,392,640 / 1,802,240 / 5,570,560. The tied output table is Q4_K, shape $[2560, 262144]$, 377,487,360 bytes. It is tied to the token embedding tensor, so the model does not need a second independent matrix for the head.

The directory says the output matrix is large, but that is not the same as reading 377 MB from disk in this experiment. It is only its descriptor and a byte count computed from shape and block format. No tensor payload was read, no Metal device was initialised, and no token was evaluated. The exact selected-descriptor output used to check the table is stored in `code/expected/ch14-w1.txt`. This W1 result establishes the file layout; it cannot show that the forward pass returns the right logits.

What remains for a weighted shortcut comparison?

The outline's W2 experiment compares prefill with the row gather enabled and disabled. For the reader experiment, copy the frozen snapshot into a scratch directory, comment out only the gather `if` block at `gemma4.c:2094--2108` in that copy, and build both variants there. Use a text file with at least 512 tokens and the same model, machine state and command for each run:

```
mkdir -p code/build/no-row-skip
cp -R code/snapshot code/build/no-row-skip/source
# In the scratch source, comment out the gather block at lines 2094-2108.
nice -n 10 sh code/build/no-row-skip/source/build.sh
GEMMA4_TIMING=1 nice -n 10 code/build/gemma4 \
  -m "$MODEL" -f code/snapshot/README.md --bench 512 64
GEMMA4_TIMING=1 nice -n 10 code/build/no-row-skip/build/gemma4 \
  -m "$MODEL" -f code/snapshot/README.md --bench 512 64
```

Compare prefill tokens per second and the printed logits value, then record the GPU-busy counter separately from wall time. The expected correctness check is identical logits; no speed value is predicted here. **Try it on your machine (W2):** compare prefill tokens/s and logits for both scratch variants. The correctness prediction is identical logits; speed may move either way because both row reduction and gather cost matter. No dated measurement for this exact toggle is cited.

PREPRINT

Pitfalls from the source history

Pitfall

The in-place gather can destroy a row it has not copied yet. The symptom would be wrong later-layer activations only on a prefill where the engine keeps fewer rows after the cache-owning layers. The cause is that source rows and destination rows share the same arrays: moving a selected row down too early can overwrite a later source row. The loop avoids that case by requiring `first >= n_out`; its comment at `gemma4.c:2094--2097` records the ordering condition. The safe fix is to preserve that precondition or gather through a separate scratch array. This is a source-documented hazard, not a claimed incident or a measured failure.

Exercises

Exercises

1. **Read.** List the layer's operations in execution order, from initial normed input through the PLE tail. Mark which decode work is fused in the common Metal path and which operations still have separate outputs.
2. **Read.** Use the four rows in the table to calculate matrix bytes for all 42 layers. Explain why the last 18 layers still read Q and output matrices even though they do not project or store new K/V.
3. **Read.** At what two places does the residual stream receive an attention or FFN update? Where does PLE add its branch, and which step applies the output scale?
4. **Change.** In a scratch copy of the snapshot, disable the gather block at layer 24. Keep the model and prompt fixed, then compare prefill speed and printed logits with the normal path. Record the change in rows processed and follow Chapter 21's quiet-GPU measurement rules. This is W2 and is not part of this chapter's experiment.
5. **Measure.** Extend the metadata-only probe to total the Q/K/V/output matrix bytes separately for sliding and global layers. Do not read the tensor payload. Compare your totals with the per-layer values derived here.
6. **Read.** How would the loop change for a model with no shared-KV layers? Which gather condition would never become true, and what would happen to `Tl`?

Rebuild checkpoint

Rebuild checkpoint

Milestone m6: one complete CPU forward pass. Implement `steps/m6_forward_cpu.c` for one text token and the E4B shared-KV layout. Reuse m3's decoders and m4/m5's norm, matvec, attention and cache pieces; add the FFN, per-layer inputs, layer loop, final norm and tied output projection. Keep the 42-layer PLE slices and the distinction between owner and shared layers explicit. Check the top-1 token against `gemma4_cpu` on ten fixed prompts. The check is intentionally top-1 rather than a claim that every intermediate float is bit-identical; record each prompt and compare logits too if the checkpoint needs debugging. The outline expects this CPU run to take minutes, so use a short fixed set and make the output reproducible.

Chapter 15

Choosing the next token

The question

Who decides the next token, and where should that decision happen?

15.1 A vocabulary-sized choice

At the end of Chapter 14, the model has turned its last hidden row into one score for every token in its vocabulary. For this E4B model that means 262,144 floating-point logits. A logit is a score: larger means the model prefers that token over a smaller-scoring one. The engine needs to turn that long list into one integer ID before it can continue.

This build makes one choice: greedy decoding. It selects the ID whose logit is greatest. If two IDs have the same greatest value, the smaller ID wins. That last rule may look fussy, but it makes the result reproducible. CPU and GPU implementations can see the same tie and still choose the same token.

The name "sampling" is often used loosely for choosing a token, but this path does not draw a random number. It has no temperature, top-k, or top-p option. Those controls reshape or filter the scores, then select randomly. The README calls out greedy generation as a limit; the source implements that limit directly. We will keep that distinction clear: here, token choice means argmax.

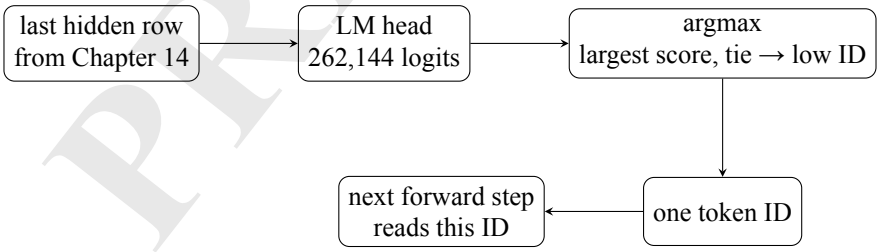


Figure 15.1: The output head gives scores; argmax turns them into the next input ID.

The diagram leaves out an important detail: the logit vector and the token buffer live in memory shared with Metal, and the GPU can write the chosen ID directly into the location used as the next step's input. The CPU does not have to read all 262,144 scores just to find one winner. This is both a data-movement decision and a scheduling decision.

15.1.1 What does "largest" mean when values tie?

Suppose a toy vocabulary has six candidates with scores `[0.2, 1.1, 1.1, -0.4, 0.7, 1.1]`. The largest score is 1.1, and it appears at IDs 1, 2, and 5. The tie rule picks ID 1. If the reduction's grouping or the number of GPU threadgroups changes, that answer stays the same.

PREPRINT

The CPU reference makes this behavior almost invisible: it starts at ID zero and only replaces the winner when a later score is strictly larger. An equal value does not replace it. On the GPU, work is split across lanes and then across SIMD groups. Each partial winner must carry both the score and its ID; comparing only scores would leave ties to the accident of reduction order.

The GPU helper called `better` encodes this pairwise rule. A candidate wins when its score is higher, or when scores match and its ID is lower. The same comparison is used first within a SIMD group and then across the group winners. That makes the parallel reduction behave like the simple CPU loop.

15.2 The reduction, first on the CPU

Why not copy the logits to the CPU and scan there? The snapshot has a CPU reference, and it is useful for correctness and fallback. For each candidate, it compares the score to the current winner. Because it scans IDs in ascending order and uses a strict greater-than comparison, the first ID at the maximum is retained. In this short loop, the tie rule follows from the traversal order.

```
3855 static int argmax(const float *x, int n) {
3856     int best = 0;
3857     for (int i = 1; i < n; i++)
3858         if (x[i] > x[best]) best = i;
3859     return best;
3860 }
```

Listing 15.1: `gemma4.c:3855–3860`

That reference is small, but the vector is not. The engine's logit allocation has one float per vocabulary ID. On the GPU path the same shared allocation is read by the argmax kernel, and the output token is written to shared integer storage. A CPU scan is therefore a useful reference, not the chosen fast path for pipelined decode.

15.2.1 Why keep the ID beside each partial score?

A parallel reduction cannot simply divide the vocabulary into pieces, take a maximum score from each piece, then compare the scores. It also needs the winning ID from every piece so it can apply the same tie-break to the final candidates. If only the scores survive the first stage, a later stage cannot know which equal maximum had the lower ID.

The Metal helper first reduces within each SIMD group. A lane compares its own score and ID with a partner lane's pair. The lane holding the preferred pair survives each comparison. One lane per SIMD group writes that group's result into threadgroup memory. After a barrier, the group leaders compare those partial results. The helper's comparison is what preserves deterministic tie handling at both levels.

```
42 static bool better(float v, int i, float v0, int i0) { return v > v0 ||
       (v == v0 && i < i0); }
43
44 static void block_argmax(thread float &v, thread int &i, threadgroup
       float *tv, threadgroup int *ti, uint tid, uint n) {
45     for (ushort off = 16; off > 0; off /= 2) {
46         const float v2 = simd_shuffle_xor(v, off);
47         const int i2 = simd_shuffle_xor(i, off);
48         if (better(v2, i2, v, i)) { v = v2; i = i2; }
49     }
50     if (tid % 32 == 0) { tv[tid / 32] = v; ti[tid / 32] = i; }
51     threadgroup_barrier(mem_flags::mem_threadgroup);
52     v = tv[0];
53     i = ti[0];
54     for (uint k = 1; k < n / 32; k++)
55         if (better(tv[k], ti[k], v, i)) { v = tv[k]; i = ti[k]; }
```

```
56      threadgroup_barrier(mem_flags::mem_threadgroup);
57  }
```

Listing 15.2: gemma4_metal.m:42–57

The helper accepts the current value and index by reference, plus scratch arrays for the partial winners. The first loop uses XOR shuffles to pair lanes at distances 16, 8, 4, 2, and 1. The reduction therefore stays within each 32-lane SIMD group. The leaders then publish one pair per group. The barrier makes those writes visible before the final comparison begins.

The second barrier protects reuse of the threadgroup scratch storage. All threads finish reading the published partial winners before any can leave the helper and reach another use of the same storage. This is a small synchronization boundary, but it is part of the correctness story: a reduction is not just arithmetic; it is also a protocol for handing intermediate values between lanes.

15.2.2 How does each lane cover the vocabulary?

The kernel assigns a strided sequence of IDs to each lane. Lane zero checks 0, then the threadgroup width, then twice the width; lane one checks 1 and the corresponding later IDs, and so on. Each lane keeps a local best pair. The threadgroup reduction combines those local answers. The launch wrapper chooses a threadgroup width no larger than 1,024 and rounds it to a multiple of 32, so the helper's SIMD-group assumptions hold.

```
1541 // ------------------------------------------------------------------
     greedy sampling
1542 // token = argmax(logits) (first index on ties), written where the next
     forward pass reads its input and
1543 // appended to the generated sequence
1544 kernel void argmax(device const float *logits [[buffer(0)]], device int
     *token [[buffer(1)]],
1545                     device int *out [[buffer(2)]], constant int &V [[
     buffer(3)]],
1546                     uint tid [[thread_index_in_threadgroup]], uint2 ntg
     [[threads_per_threadgroup]]) {
1547     threadgroup float tv[32];
1548     threadgroup int ti[32];
1549     float mx = -1e38f;
1550     int im = 0x7fffffff;
1551     for (int j = tid; j < V; j += ntg.x)
1552         if (logits[j] > mx) { mx = logits[j]; im = j; }
1553     block_argmax(mx, im, tv, ti, tid, ntg.x);
1554     if (tid == 0) { *token = im; *out = im; }
1555 }
```

Listing 15.3: gemma4_metal.m:1541–1555

The initial score is a very negative finite value and the initial ID is the largest signed integer. This gives every real candidate a chance to win. The strict comparison in the local scan means a lane retains its first occurrence. The later `better` comparisons extend that rule across lanes and groups. Lane zero finally writes the ID twice: once to `token`, where the next forward pass reads it, and once to `out`, where the generated sequence is collected.

A single kernel launch does this entire choice. There is no CPU reduction step after it, and no second GPU pass to copy the winner. For greedy decoding, that is the essential contract: after the kernel completes, the next token is already in the input slot.

15.3 The next step is already in flight

A transformer generates one token at a time: the hidden state for token t produces a score vector, the chosen ID becomes token $t+1$, and the model computes again. A naive implementation waits after every step. The CPU reads the winner, places it in the input array, then submits the next forward pass. During that pause, the GPU can sit idle while the CPU handles the handoff.

The pipelined path overlaps those jobs. The argmax kernel writes the winning ID into the next input slot. The CPU can submit the next step before it waits to inspect the token from the previous step. Since GPU command buffers are queued in order, the next forward pass sees the ID after the argmax has written it. Meanwhile, the CPU can receive an earlier token for printing or stopping generation. The generation loop is deliberately one token ahead of the CPU's view.

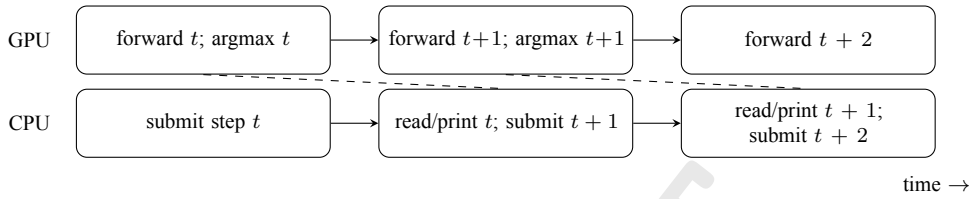


Figure 15.2: The CPU submits work ahead, then reads a token only when it needs that token for output or control.

The snapshot's `decode` function has two modes. In synchronous mode it finishes each step before moving to the next. In pipelined mode it submits the work, waits only for an older fence, and makes the next forward call before reading the newest generated ID. The callback therefore receives token `i-1` while the GPU may already be computing token `i`. At the end, the function waits for the final queued work and delivers the last token.

```
3877 typedef int (*token_fn)(void *ctx, int token);  // gets each generated
       token in order; returns 0 to stop
3878
3879 // Greedy decoding of up to N tokens after the tokens already in st; the
        sampled tokens go to out[0..N),
3880 // and to on_token (if not NULL) as soon as the CPU has them. Returns
       the number of tokens generated.
3881 // The GPU samples (argmax) and writes each token where the next step
       reads its input.
3882 //   sync = 1: the CPU waits for each token before it encodes the next
       step (as llama-bench and a plain
3883 //             interactive loop do), so the GPU idles while the CPU
       catches up;
3884 //   sync = 0: pipelined: step i+1 is submitted before the CPU waits for
        token i, so the GPU never waits
3885 //             for the CPU. The CPU still gets each token as soon as it
       exists (to stream it, or to stop;
3886 //             the step already queued behind a stop is discarded).
3887 static int decode(const Model *m, Acts *a, State *st, int first_token,
       int N, int sync, float *logits, int *out,
3888                   token_fn on_token, void *ctx) {
3889     a->tokens[0] = first_token;
3890     void *prev = NULL;  // pipelined: the previous step's work
3891     int n = 0, stop = 0;
3892     for (int i = 0; i < N && !stop; i++) {
3893         forward(m, a, st, 1, 0, 1, logits);
3894         ArgmaxJob job = {logits, a->tokens, out + i, m->n_vocab};
3895         STEP(metal_argmax(a->tokens, out + i, logits, m->n_vocab), 1,
       argmax_task, &job);
```

```
3896            if (sync) {
3897                gpu_sync();
3898            } else {
3899                gpu_submit();
3900                gpu_wait(prev);
3901                prev = gpu_fence();
3902            }
3903            const int ready = sync ? i : i - 1;  // the newest token the CPU
        can read
3904            if (ready >= 0) {
3905                n++;
3906                stop = on_token && !on_token(ctx, out[ready]);
3907            }
3908        }
3909        gpu_wait(prev);
3910        gpu_sync();
3911        if (!sync && !stop && N > 0) {  // pipelined: the last token
3912            n++;
3913            if (on_token) on_token(ctx, out[N - 1]);
3914        }
3915        return n;
```

Listing 15.4: `gemma4.c:3877–3915`

There are two details worth tracing. First, each iteration calls `forward`, then schedules argmax with the token buffer and output array. Second, `gpu_wait(prev)` waits on the fence from the earlier step, not the one just submitted. The CPU's callback can consequently lag the newest queued step by one token. If a callback asks to stop, the already queued step cannot be unsent; the comment notes that work beyond a stop is discarded.

The benchmark compares this mode with a fully synchronous decode. It reports tokens per second for both, checks that both generated sequences match, and can report GPU busy time alongside wall time when `GEMMA4_TIMING` is enabled. These are different views of the same run: token rate describes end-to-end throughput; GPU busy time describes the intervals recorded for completed command buffers. A close match is evidence that little wall time is being lost to gaps, not a universal guarantee for every prompt, machine load, or thermal state.

A useful way to read the benchmark is to draw a boundary around exactly what is timed. The prompt is prefilled first, the GPU is synchronized, and only then does the decode timer begin. The code runs a warm-up repetition with a negative repetition index so compilation and the first dispatch do not enter the recorded samples. It runs synchronous decode first and pipelined decode second, with the same prompt and first token. At the end it compares the generated integer arrays. The rate is computed from the requested number of tokens and elapsed wall time; it is not a kernel-only rate.

The optional timing report takes a second view of that interval. The host reads the cumulative GPU time before decode, runs the decode loop, and subtracts the starting counter afterward. It also clears the SSD-read and router-wait counters for that trial. This matters when interpreting the words "GPU busy": it is the sum of GPU execution intervals recorded for completed command buffers. A ratio near one token's worth of wall time says the queue was well occupied during that sample, but the CPU may still have done work concurrently, such as reading a per-layer row.

The warm-up and order are part of the protocol, not incidental details. If one mode ran first on a cold machine and the other after a long prefill, the comparison could mix scheduling with thermal behavior. The source repeats each mode several times, reports a mean and spread for the measured rates, and emits its own generated string as a quick check. For a careful external capture, keep the same depth and prompt, note whether the machine is idle, and compare the same reported statistic across runs. Do not turn one best rate into a general performance claim.

A practical trace through one iteration is: the CPU has already put the current token in the input; forward work is encoded; argmax is encoded after the logits are ready in queue order; the command

PREPRINT

Observation	What it tells you	What it cannot tell you alone
Pipelined tokens/s	End-to-end decode throughput for this prompt and run	Whether every individual token had the same latency
GPU ms/token	Recorded device execution time averaged over the interval	How long the CPU waited on a particular token
Sync / pipeline IDs match	Both modes selected the same sequence in this comparison	That all schedules or inputs are race-free
Generated text	A human-readable rendering of token IDs	Whether invisible probability shifts occurred
KLD over scored positions	Distribution changes across many candidate IDs	The exact cause of a discrepancy without a trace

Table 15.1: Use several observations because each sees a different part of the generation path.

buffer is submitted; the CPU waits only for the older fence; it may now use an earlier output token, while the newest one remains device-owned. The next call reaches per-layer input preparation, submits what it has built, and waits for the newest required token before reading that token's file-backed row. Once the row is prepared, the new GPU work continues. This is why "no CPU round trip for argmax" and "no CPU wait at all" are different claims.

15.3.1 A concrete pass through 262,144 candidates

The vocabulary is large enough that it helps to count the work. A float is four bytes, so the logit vector occupies 1,048,576 bytes: exactly 1 MiB before allocator and alignment details. The output decision is one 32-bit integer. A CPU sampler that needs every score would have to make about a megabyte of values available to its decision code; the greedy GPU path reads those values where they already are and writes only the winning ID.

For a concrete launch, suppose the device permits 1,024 threads in this threadgroup, the maximum used by the wrapper when available. The 262,144 IDs divide evenly: each lane visits 256 logits, spaced 1,024 IDs apart. Lane 0 examines 0, 1,024, 2,048, and so on; lane 1 examines 1, 1,025, 2,049, and so on. Each lane reduces its own 256 observations to one score-ID pair. Then 1,024 lanes form 32 SIMD groups. One lane in each group publishes its winner, and the 32 group winners are compared to produce the final ID.

This two-level count explains why the scratch arrays each have 32 entries. They store the per-SIMD-group candidates, not one slot per vocabulary item. The code can support a smaller multiple-of-32 threadgroup too: each lane then visits more IDs, while the number of partial results still matches the number of SIMD groups. The launch wrapper asks the compiled pipeline for its maximum thread count and caps it at 1,024.

Consider the six-score toy list from earlier. With six separate lanes, three lanes might see a tied 1.1. At the SIMD reduction, ID 1 defeats ID 2 because the scores match and 1 is lower. If ID 1 and ID 2 end up in different SIMD groups, the second-stage comparison applies exactly the same rule. The result does not depend on which subgroup happened to finish first. In the real vocabulary, the same reasoning applies even though most lanes only see a subset of the candidates.

The table also helps separate correctness from speed. The CPU reference is easy to audit and useful when a GPU call declines its arguments. The GPU path must preserve the same comparison rule while respecting lane communication and memory ordering. A matching result checks the choice; it does not by itself prove that the pipeline scheduled the next read safely.

15.3.2 What does the benchmark compare?

The inner decode function explains token ownership, while the benchmark checks whether overlapping the work changes the sequence or the measured rate. It resets the cache depth for each repetition, restores the same prompt IDs, and runs a fresh prompt forward pass to recover the first generated token. Then

PREPRINT

Step	CPU reference / control	GPU fast path
Read candidates	Walk every logit in ID order	Each lane walks a strided subset
Keep a winner	Replace only on a strictly larger score	Keep a score-ID pair; tie goes to lower ID
Combine partials	No partial results	SIMD shuffle, then threadgroup winners
Publish token	Store the selected integer	Store into next input and output slots
Wait / observe	Caller already has the result	Queue work; wait only for prior work when needed

Table 15.2: Both implementations choose the same ID; only the GPU version keeps the decision beside the logits.

it measures the requested number of continuation steps in each mode. The excerpt below shows the two-mode order, the sequence comparison, and the report labels. It is a compact view of the experiment protocol rather than a second implementation of argmax.

```
3975      for (int sync = 1; sync >= 0 && N > 0; sync--)
3976          for (int rep = -1; rep < reps; rep++) {
3977              st.n_past = D;
3978              memcpy(acts.tokens, prompt, P * sizeof(int));
3979              forward(m, &acts, &st, P, P - 1, 1, logits);
3980              gpu_sync();
3981              const double t0 = now();
3982              t_wait_ssd = t_wait_gpu = 0;
3983              const double busy0 = gpu_time();
3984              decode(m, &acts, &st, first, N, sync, logits, gen[sync],
      NULL, NULL);
3985              if (rep >= 0) tg[sync][rep] = N / (now() - t0);
3986              if (getenv("GEMMA4_TIMING"))
3987                  fprintf(stderr, "decode %.1f ms/token: GPU busy %.1f ms,
       waiting for expert reads %.1f ms, for the router %.1f ms\n",
3988                          1e3 * (now() - t0) / N, 1e3 * (gpu_time() -
      busy0) / N, 1e3 * t_wait_ssd / N, 1e3 * t_wait_gpu / N);
3989          }
3990
3991      char what[64], at[32] = "";
3992      if (D) snprintf(at, sizeof at, " @ depth %d", D);
3993      snprintf(what, sizeof what, "prefill  pp%d%s", P, at);
3994      report(what, pp, reps, "tokens/s");
3995      double sd, best = ttft[0];
3996      for (int i = 0; i < reps; i++) best = fmin(best, ttft[i]);
3997      const double mean = mean_sd(ttft, reps, &sd);
3998      printf("%-34s %8.1f ± %5.1f ms        (best %.1f)\n", "TTFT", 1e3 *
      mean, 1e3 * sd, 1e3 * best);
3999      if (N == 0) return;
4000      for (int sync = 1; sync >= 0; sync--) {
4001          snprintf(what, sizeof what, "decode   tg%d @ depth %d, %s", N, D
       + P, sync ? "sync" : "pipelined");
4002          report(what, tg[sync], reps, "tokens/s");
4003      }
4004      if (memcmp(gen[0], gen[1], N * sizeof(int))) fprintf(stderr, "
      warning: the two decode modes generated different tokens\n");
4005      fprintf(stderr, "generated: \"");
4006      for (int i = 0; i < N && i < 48; i++) {
4007          char buf[256];
4008          fwrite(buf, 1, detokenize(tk, gen[1][i], buf, sizeof buf),
      stderr);
```

```
4009        }
```

Listing 15.5: gemma4.c:3975–4009

Notice that the prompt forward and its synchronization occur before `t0`. The reported generation rate excludes prompt prefill; the benchmark reports prompt throughput and time-to-first-token separately. This keeps the decode comparison focused on the repeated one-token path. Each repetition starts from the same state depth, so one trial's KV cache does not silently become the next trial's longer context.

The synchronous loop is measured before the pipelined loop. The arrays `gen[0]` and `gen[1]` hold the two sequences, and the comparison checks every requested ID. The printed sample is detokenized from the pipelined array only after both measurements. That final string is convenient for a human, but the integer-array comparison is the stronger local check because different token IDs can sometimes render similarly.

15.3.3 When does a selected token become visible text?

Selection and printing are separate steps. A generated ID is first written by argmax into the output array. The callback receives it when the CPU is allowed to inspect that array. The printer checks the configured stop IDs before converting the token into bytes. If it is a stop token, the function returns immediately without writing that token's spelling to standard output. Otherwise it detokenizes the ID and flushes, so streaming text appears as each token becomes available.

```
4015 typedef struct { const Tokenizer *tk; const int *stop; int n_stop; }
        Printer;
4016
4017 static int print_token(void *ctx, int token) {
4018     const Printer *p = ctx;
4019     for (int i = 0; i < p->n_stop; i++)
4020         if (token == p->stop[i]) return 0;
4021     char buf[256];
4022     fwrite(buf, 1, detokenize(p->tk, token, buf, sizeof buf), stdout);
4023     fflush(stdout);
4024     return 1;
4025 }
```

Listing 15.6: gemma4.c:4015–4025

This distinction matters in an interactive response. The model can generate an end marker that controls the loop but is not part of the displayed answer. Similarly, a callback that asks to stop generation is a control decision, not a new sampling rule. Because pipelined decode may already have submitted the next step, stop handling can leave queued work that must finish before its buffers are reused; the source explicitly waits and describes the queued step as discarded from the visible result.

The printer has no vocabulary-ranking work. Its job starts after the winner exists: stop check, ID-to-text conversion, write, flush. Keeping these boundaries clear makes it easier to locate a delay. If the GPU's busy interval is large, inspect the forward pass and argmax path. If the model is done but text appears late, inspect when the callback can read the result and how detokenization and output flushing are scheduled.

15.3.4 A real dependency: per-layer embeddings

If the GPU can place the next ID for itself, why does the CPU ever wait? E4B has a per-layer embedding table. Chapter 13 showed that the table remains mapped as a file instead of being wired into GPU memory. To prepare the per-layer inputs for a call, the CPU must know the token IDs and read their corresponding rows.

The `pl_inputs` comment spells out this boundary. It submits the work already prepared for the current call, waits for the previous step represented by `prev`, and then reads the needed rows on the CPU.

While that wait completes, the GPU can run what the current call has already submitted. The wait is therefore placed after useful submission rather than before all work begins.

```
1955 // a->pl = the per-layer inputs of this call's tokens, from a->pl (the
       projection) and the token table. The CPU
1956 // needs the tokens: in a pipelined decode the previous step's GPU work
       (`prev`, all submitted before this call)
1957 // samples them, so this waits for it, while the GPU runs what this call
        has submitted so far.
1958 static void pl_inputs(const Model *m, Acts *a, int T, void *prev) {
1959     gpu_submit();
1960     gpu_wait(prev);
1961     PlJob job = {m, a};
1962     if (T >= 16) parallel_for(T, pl_tok_row, &job);
1963     else for (int t = 0; t < T; t++) pl_tok_row(&job, t, 0);
1964     PlCombineJob cj = {a->pl, a->pl_tok, m->pl_proj_norm, m->n_pl, m->
       eps, 1.0f / sqrtf((float)m->n_embd), sqrtf(0.5f)};
1965     STEP(metal_pl_combine(a->pl, a->pl_tok, m->pl_proj_norm, T * m->
       n_layer, m->n_pl, m->eps, cj.s_in, cj.s_out),
1966          T * m->n_layer, pl_combine_row, &cj);
```

Listing 15.7: `gemma4.c:1955–1966`

This is the pipeline's unavoidable handoff in this design. The GPU is able to produce the integer ID without a round trip through the CPU, but the mapped per-layer table makes the CPU a consumer of that ID before it can finish preparing the next call. The ordering matters: submit independent GPU work, wait for the preceding token to be ready, read the token-dependent row, then continue with the rest of the call.

The README describes pipelined decode as keeping GPU busy time close to wall time per token. Treat that as a measured behavior of the documented configuration, not a promise that any queue is always full. CPU scheduling, thermal throttling, memory pressure, a stop callback, and synchronous code elsewhere can all create gaps. We will check the snapshot's benchmark output rather than manufacture a rate here.

15.4 What random sampling would change

Greedy decoding is simple because the largest score decides. Temperature, top-k, and top-p sampling need extra work. Temperature changes the relative sharpness of the scores before random selection. Top-k removes all but the highest k candidates. Top-p keeps a set large enough to reach a chosen cumulative probability. These operations need a probability distribution, a random state, and a policy for how the filtered candidates are represented.

One possible experiment is a CPU sampler: make the logits available to the CPU, filter candidates, draw a random number, and return the chosen ID. But a CPU implementation would need to synchronize with the GPU and read the logits, not merely one integer. The current pipeline's advantage is that the GPU writes the token directly into the next input location. A CPU sampler would put a readback and a CPU decision between steps, so its cost includes lost overlap as well as the sampling arithmetic.

An on-GPU sampler could keep the decision near the logits and write the token to the same input slot. It would still need a random-number generator with a defined state, a way to compute or approximate the filtered distribution, deterministic rules for edge cases, and tests showing that CPU and GPU implementations agree when given the same random stream. It would also change the experiment contract: output tokens would vary with seed and sampling settings. The current argmax path has no random state and can be compared ID for ID.

This chapter does not add a sampler. The practical exercise is to design one and say which data must cross the CPU/GPU boundary. A safe first prototype can work on a scratch copy after a run has already produced logits; a production design would have to change the token-selection interface and the

benchmark's assumptions. In either case, report the synchronization and transfer cost separately from the random selection cost.

15.5 A rare race, found by a distribution test

Symptom. A decode-path perplexity check reported mean KLD 0.83 where the expected comparison was about 0.00001.

Cause. The CPU wrote tokens into a buffer while the GPU could still be reading those tokens. In the 26B model, MoE routing caused a CPU wait each layer and hid the timing window. The dense E4B had no such waits, so a later prompt chunk, segment, or scored token could overwrite data too early.

Fix. Wait for the GPU before those CPU writes. The project history records this under README.md, "Bugs fixed on the way," and the code comments document the ordering around per-layer inputs (`gemma4.c`, lines 1955–1957).

Why did this appear in a statistical check instead of an obvious crash? The buffer held plausible integer IDs. A race does not have to produce an invalid address or a visibly broken token; it can replace one valid token with another. The resulting forward pass still runs, but scores a different continuation. A check that compares only whether the top token matched might miss some distribution changes. KLD compares the full predicted distributions and exposes a large divergence even when the failure is intermittent.

The CPU and GPU sharing a buffer makes the lifetime rule concrete: the CPU may not overwrite a token slot until the GPU has finished reading it. The current path's fences and waits express that rule. The model architecture changed the timing enough to expose the defect, even though the buffer code itself had not become more complicated. A performance shortcut can remove an accidental wait that was masking a correctness bug.

When debugging a similar symptom, record the exact call sequence and the ownership interval for each shared buffer. Ask which command buffer last wrote a value, which one reads it next, and what event guarantees that the read has completed before a CPU overwrite. Then compare distributions or vectors, not only rendered text. The actual story is in the project's README, not a generic claim that "GPU races are hard": this specific race was found by the decode-path statistical test and fixed by waiting.

15.6 Experiment: inspect the token choice without weights

The snapshot's W0 build is safe because it compiles the code and prints command-line help without loading a model. From the book root, build the isolated snapshot at low priority and ask for help:

```
nice -n 10 sh code/snapshot/build.sh
code/build/gemma4 --help
```

The snapshot build completed, and the no-model help command printed usage text. This line is from the captured output:

```
  -r reps                 repetitions of each --bench test (default 5)
```

The full output is saved as `code/build/ch15-help.txt`. This W0 run confirms that the snapshot builds and its command-line interface starts without a model. It does not measure token throughput or validate GPU timing.

W1 metadata probe. The model file is not part of the snapshot, so this read-only probe checked the first 24 bytes of the local E4B GGUF and stopped there. A small Python standard-library script unpacked the magic, version, tensor count, and metadata count; it did not parse tensor data, read vocabulary strings, allocate model buffers, or run inference. The captured output was:

PREPRINT

```
file=gemma-4-E4B-it-IQ4_XS.gguf
magic=GGUF version=3 tensor_count=720 metadata_count=56
```

This confirms that the file has a GGUF v3 header and shows the declared counts. It does not confirm the tensor shapes, tokenizer entries, or that the file can be loaded; those require additional metadata parsing or a model load and are outside this probe. The script used for the check is in `code/build/ch15_gguf_header.py`.

W2 reader experiment: run the model benchmark with a prompt file containing at least 512 tokens and 128 decode tokens:

```
GEMMA4_TIMING=1 code/build/gemma4 -m $MODEL -f $PROMPT --bench 512 128
```

Try it on your machine (W2): run the command above in a scratch copy and inspect pipelined decode tokens/s beside GPU-busy milliseconds per token. Compare those views directly; record machine, date, and idle-gate result. No dated reference for this exact command is cited here.

A useful correctness check in that later session is that the synchronous and pipelined paths report the same generated sequence. The benchmark already compares their ID arrays and emits a warning if they differ. This check does not prove there is no race under every possible schedule, but it is a direct invariant for this pair of paths.

15.7 Exercises

1. A five-token toy vocabulary has logits `[0.4, 1.2, 1.2, -0.1, 0.8]`. Which ID does the CPU reference choose? Explain how a GPU reduction preserves the same answer.
2. Trace one iteration of `decode`. Which token slot does the GPU write, and when does the callback learn that token in synchronous and pipelined modes?
3. Design a top-k=40, temperature=0.8 sampler for the CPU. List the buffers it reads and writes, identify each required synchronization, and estimate the cost you would measure separately from model forward time.
4. Why might a top-token comparison fail to reveal the token-buffer race while a KLD measurement finds it? What additional check would you add to compare synchronous and pipelined generation?
5. Sketch an on-GPU sampling interface. What random state must persist between calls, how does it seed reproducibly, and how will the selected ID reach the next forward pass?

15.8 Rebuild checkpoint m6b

The smallest useful next component is `steps/m6b_generate.c`: a greedy generation loop built on the CPU forward checkpoint from m6. It needs a fixed prompt, a token array, an argmax routine with the same lowest-ID tie rule, and a loop that feeds each selected ID back as the next input. Test it against `gemma4_cpu` for 32 generated tokens, comparing IDs rather than only the printed text. The target is the same greedy sequence. This checkpoint adds no random sampler; sampling would be a separate extension with an explicit seed and a changed comparison test.

PREPRINT

Part IV

CHAT

PREPRINT

Chapter 16

The chat loop, templates and the cache prefix

PREPRINT

Part IV: Chat

Parts I and II built a loader and a fast path; Part III followed the model's work through attention, cache, layers, and token choice. Part IV turns that engine into a conversation. A chat program has to keep the visible history, render the model's exact prompt, and decide which already-computed positions are still valid. This chapter follows that loop from a user line to the next reply.

The question

How can a conversation cost only the new words when the model sees the whole conversation?

Reading map: 20 pages

This map allocates the outline's estimate before the detailed walkthrough. It includes diagrams, excerpts, experiment protocol, exercises, and the checkpoint.

Subsection	Pages
Part IV opener and question	1
Conversation state and the rendered template	4
Prefix matching, cache safety and rollback	5
Reply generation, context growth and reporting	3
Interactive commands, media prefill and interruption	3
W0/W1 probes and W2 chat experiment	1
Documented pitfalls	1
Exercises and Appendix F hints	1
Checkpoint m8	1
Total	**20**

The idea

What does the model remember?

The answer is not the transcript. The model is a function that receives token IDs and produces scores; it does not keep a conversational file in the weights. During a forward pass, each layer writes key and value rows into its KV cache. Later positions can attend to those earlier rows. If the engine wants to continue from an old position, the cache must contain exactly the rows that the newly rendered prompt expects at that position.

A transcript and a cache therefore have different jobs. The transcript is the editable source of truth: user messages, generated replies, and any attached media. The cache is a derived computation for one particular prefix of that transcript. When the user edits, undoes, or clears history, the engine can rebuild the affected suffix from the transcript while keeping an earlier matching prefix.

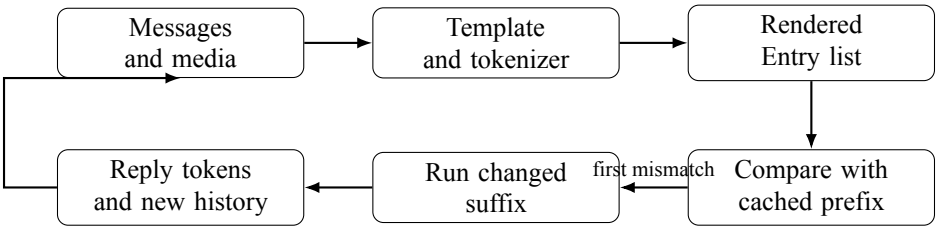


Figure 16.1: The transcript is rendered again each turn. Only the suffix after the reusable prefix needs another forward pass.

Suppose the previous prompt and generated answer together occupy 1,000 cache positions. The next conversation rendering contains those same first 1,000 entries, then 14 entries for the new user message, then the open model turn. A full rerun would compute all 1,015 prompt entries again before generation. Prefix reuse starts at position 1,000 and computes only the 15 new prompt entries; the reply itself still needs its own decode steps. These counts are an illustration of the mechanism, not a benchmark result.

The cache is not a memoized answer to a message. It is a sequence of intermediate states indexed by positions. If the first changed item is at position 650, entries after 650 cannot be trusted merely because their text happens to look familiar: their hidden states were produced after a different prefix. The rule is strict: reuse a contiguous prefix, and recompute from the first invalid position onward.

Why render the whole conversation again?

The transcript is simpler to edit than a partially advanced model state. Rendering every turn gives one canonical sequence to compare against the cache. The cost of walking and tokenizing the history is paid on the CPU, but the expensive matrix work is limited to the suffix that changed. It also means ‘/undo‘ can be described as a transcript edit followed by the same prefix check, instead of a second bespoke model-state mechanism.

Gemma’s chat prompt is not just the visible words. It contains control tokens that mark speakers and turn boundaries. In a simplified two-turn exchange, the sequence is:

```
<|turn>user\nHello<turn|>\n
<|turn>model\nHi.<turn|>\n
<|turn>user\nAgain?<turn|>\n
<|turn>model\n
```

The final open model turn is a generation prompt: the next selected ID continues from there. These spellings stand for vocabulary IDs, not strings that the engine asks the model to interpret as ordinary punctuation. The tokenizer recognizes the control token spellings when the template constructs the prompt.

There are two representations to keep distinct. A `Message` stores a role, text, and optional media parts. An `Entry` is one position in the rendered model input: either a token ID or a row of encoded media. This lets ordinary text and image/audio rows share the same sequence and prefix comparison. In the excerpt, the chat also stores the last rendered entries separately from the message history; that array is the description of what the cache currently represents.

What belongs in the chat state?

The engine’s `Chat` groups the model and tokenizer references with the system prompt, template options, editable messages, and the rendered prefix. It also owns generation buffers, KV state, and scratch space for calls that mix text tokens with media rows. The first excerpt focuses on those boundaries.

PREPRINT

```
4072 // a token, or row `row` of a part's rows (whole: the part is run in one
         call, its rows seeing each other)
4073 typedef struct { int tok; const f16 *rows; int row, whole; } Entry;
4074 typedef struct { int user; char *text; Parts media; } Message;
4075
4076 typedef struct {
4077     const Model *m;
4078     const Tokenizer *tk;
4079     const char *system;  // or NULL
4080     int think;           // thinking on: the template's <|think|> in the
         system turn, no empty thought
4081     int empty_thought;   // thinking off: the model's turn starts with
        an empty thought (the 26B's template; the
4082                          // E2B / E4B's doesn't, and those models then
        write their thinking after it)
4083     int bidir_images;    // an image's (or frame's) rows see each other
        (the 26B; not the E2B / E4B)
4084     Message *msg;
4085     int n_msg, cap_msg;
4086     Entry *cached;       // what the KV cache holds, position by
        position
4087     int n_cached, cap_cached;
4088     Acts acts;
4089     State st;
4090     float *logits;
4091     int *out, stop[2];
4092     int n_ctx, slack;    // the most positions (the KV cache starts
        smaller and grows: see chat_reply); its slack
4093     float *stage;        // [T_max][n_embd] a call's input rows when it
        mixes tokens and media
4094     int *ids;            // [T_max] their tokens (0 for media rows), for
         the per-layer embeddings
4095 } Chat;
```

Listing 16.1: `gemma4.c`:4072–4095

The `msg` array is the durable conversational history. `cached` is a second list, and its comment is a useful invariant: it records what the KV cache holds position by position. The count `n_cached` can be shorter than the latest rendering; that is how a changed suffix is represented. The prompt is allowed to evolve without pretending the old cache still matches it.

The `think` and `empty_thought` flags are separate because they mean different template choices. Thinking enabled places a control token in the system turn. Some model templates, including the 26B template discussed in the source comment, instead open a model reply with an empty thought channel when thinking is off. E2B and E4B do not. The engine reads the GGUF template and selects the right form; it must not copy the 26B convention into E4B just because both models use Gemma chat markers.

The `stage` and `ids` arrays are temporary input preparation. Text can use token IDs directly, while media has already become embedding rows. A mixed call needs a contiguous staged embedding input, and token IDs alongside it for per-layer embeddings. This is why `Entry` can point either to a token or to one row of a media tensor instead of forcing all input into a single token-only array.

How is the prompt rendered?

The renderer starts with an optional beginning-of-sequence token, emits the system turn when thinking or a system prompt requires one, then walks messages in order. User text is trimmed and tokenized; model replies are passed through the reply-specific text helper. Media markers and their embedding rows are

inserted at the beginning of their message. Finally the renderer appends the open model turn. Here is the source walkthrough; the exact ordering is part of the prompt contract.

```
4143 static void chat_render(const Chat *c, Entry **e, int *n) {
4144     const Tokenizer *tk = c->tk;
4145     int cap = 0;
4146     *e = NULL, *n = 0;
4147     if (tk->bos >= 0) push_entry(e, n, &cap, (Entry){tk->bos, NULL, 0,
     0});
4148     if (c->think || c->system) {
4149         push_str(tk, e, n, &cap, c->think ? "<|turn>system\n<|think|>\n"
     : "<|turn>system\n");
4150         if (c->system) {
4151             size_t len;
4152             const char *t = trimmed(c->system, &len);
4153             push_text(tk, e, n, &cap, t, len, 0);
4154         }
4155         push_str(tk, e, n, &cap, "<turn|>\n");
4156     }
4157     for (int i = 0; i < c->n_msg; i++) {
4158         const Message *msg = &c->msg[i];
4159         push_str(tk, e, n, &cap, msg->user ? "<|turn>user\n" : "<|turn>
    model\n");
4160         for (int k = 0; k < msg->media.n; k++) {  // at the start of the
     message, where llama.cpp's mtmd puts them
4161             const Part *pt = &msg->media.p[k];
4162             if (pt->kind == PART_FRAME) {  // "mm:ss <|image>...<image
    |>", frames separated by a space
4163                 char ts[24];
4164                 snprintf(ts, sizeof ts, "%s%s ", k && msg->media.p[k -
    1].kind == PART_FRAME ? " " : "", pt->time);
4165                 push_str(tk, e, n, &cap, ts);
4166             }
4167             push_str(tk, e, n, &cap, pt->kind == PART_AUDIO ? "<|audio>"
     : "<|image>");
4168             for (int r = 0; r < pt->n; r++)
4169                 push_entry(e, n, &cap, (Entry){0, pt->rows, r, pt->kind
    != PART_AUDIO && c->bidir_images});
4170             push_str(tk, e, n, &cap, pt->kind == PART_AUDIO ? "<audio|>"
     : "<image|>");
4171         }
4172         if (msg->user) {
4173             size_t len;
4174             const char *t = trimmed(msg->text, &len);
4175             push_text(tk, e, n, &cap, t, len, 0);
4176         } else {
4177             push_reply(tk, e, n, &cap, msg->text);
4178         }
4179         push_str(tk, e, n, &cap, "<turn|>\n");
4180     }
4181     push_str(tk, e, n, &cap, c->think || !c->empty_thought ? "<|turn>
    model\n" : "<|turn>model\n<|channel>thought\n<channel|>");
4182 }
```

Listing 16.2: `gemma4.c:4143–4182`

The template makes the speaker boundary explicit with `<|turn>user` or `<|turn>model`, and closes each completed message with `<turn|>`. The open model prefix at the end is deliberately not closed. Changing a newline or omitting a boundary would shift the token sequence and invalidate the cache from

that point, even if the visible conversation looked unchanged.

Media rows occupy entries too. For each attached part, the renderer adds an audio or image marker, one entry for every encoded row, and a closing marker. A video frame also gets its timestamp text. In E4B, image rows are marked as non-bidirectional here, reflecting the model layout: each causal input position sees the earlier positions, including its own, rather than all rows in the image seeing one another. The `whole` flag can keep a media part together when a cache restart requires that part to be run as a unit.

The conditional system turn illustrates why the rendering rule comes from the model's template rather than a generic role-tag recipe. When thinking is enabled, the renderer emits the system turn with `<|think|>` even if there is no separate system instruction. If a system string exists, it is trimmed and tokenized, then closed. When thinking is off and no system text exists, that entire turn is absent. The generation prompt at the end also depends on the template's empty-thought choice.

A compact checklist makes this template choice easier to audit. The following rows summarize the branches visible in the renderer and state comments; the exact opening comes from the loaded model's chat template settings.

Setting	System prefix	Open model prefix
Thinking enabled	System turn with the thinking marker	Model turn without an added empty thought
Thinking disabled, E4B	System turn only when a system string exists	Model turn with no empty thought
Thinking disabled, 26B template	Template-specific system choice	Model turn begins with its empty thought channel
Media in a user turn	Media marker and rows at message start	Same open model turn after completed turns

This table describes the template branch, not the content produced by the model. A template marker tells the model which channel or role is in effect. The model may then generate text in that channel, and the printer may treat some channel markers as control state instead of visible text. Because this pipeline has separate render, generate, and print steps, the chat history should be inspected at each boundary when a reply appears in the wrong channel.

The system prompt is not automatically copied into each user message. When one is present, the renderer trims it and places it in its own system turn. If thinking is on, the system turn is emitted even without a system sentence because the thinking marker itself belongs there. If both thinking and the system prompt are absent, the renderer skips the system turn entirely. That changes every following absolute position and therefore the full rendered prompt prefix.

For a manual check, write the sequence for a system instruction "Be brief", then user "2+2?", then model "4", then user "Why?", and finally the open model prefix. Mark each control-token boundary before tokenizing ordinary text. Compare with the engine's `--tokens` output only in an approved model session; this chapter's W1 probe does not load or tokenize the model vocabulary.

The prefix and the cache

Where does reuse stop?

The cache scan compares the new rendered list with the old cached list one position at a time. Its simple comparison function checks a token ID for text entries; for media entries it checks the row pointer and row number. The pointer check matters because two image rows with identical pixel values are not necessarily the same still-live object that the cache was computed from. Reuse is based on identity and position, not a guess that two media arrays look equivalent.

The first matching run in `chat_run` handles the prefix. The next excerpt includes the scan, the special treatment of a whole media part, the sliding-window guard, and the preparation of a bounded call. The `forward` call at line 4302 runs that batch and fills the logits; lines 4303–4307 clear the staged pointers, advance the input, synchronize, and return the reused count.

PREPRINT

```
4262 static int chat_run(Chat *c, const Entry *e, int n, int N, int logits) {
4263     if (n + N + 1 > c->st.n_ctx && c->st.n_ctx < c->n_ctx) {  // grow
     the cache: to twice its size, or to what it needs
4264         int want = 2 * c->st.n_ctx > n + N + 1 ? 2 * c->st.n_ctx : n + N
      + 1;
4265         state_grow(c->m, &c->st, want < c->n_ctx ? want : c->n_ctx, c->
     slack);
4266     }
4267     // reuse the cached prefix; a whole part is run whole, and the last
     position is always run (for its logits)
4268     int p = 0;
4269     while (p < n && p < c->n_cached && same_entry(e[p], c->cached[p])) p
     ++;
4270     if (p == n && !logits) return n;
4271     if (p == n) p--;
4272     if (e[p].rows && e[p].whole) p -= e[p].row;
4273     p = state_keep(c->m, &c->st, p);  // (0 if the sliding-window cache
     no longer holds what p needs)
4274     const int reused = p;
4275     c->st.n_past = p;
4276     const int E = c->m->n_embd;
4277     for (int i = p; i < n;) {
4278         int T = 0, media = 0;
4279         if (i > p) gpu_sync();  // the GPU reads a->tokens and c->stage:
      the previous call must be done with them
4280         if (e[i].rows && e[i].whole) {  // a part run whole
4281             while (i + T < n && e[i + T].rows == e[i].rows) T++;
4282             if (T > c->acts.T_max) die("image of %d tokens: more than %d
     ", T, c->acts.T_max);
4283             for (int t = 0; t < T; t++)
4284                 for (int k = 0; k < E; k++) c->stage[(size_t)t * E + k]
     = f16_table[e[i].rows[(size_t)(e[i].row + t) * E + k]];
4285             c->acts.embd = c->stage, c->acts.bidir = 1;
4286         } else {  // up to T_max tokens and media rows: a call that
     mixes them takes its input rows from c->stage
4287             while (i + T < n && T < c->acts.T_max && !(e[i + T].rows &&
     e[i + T].whole)) media |= e[i + T++].rows != NULL;
4288             for (int t = 0; t < T; t++) {
4289                 const Entry *x = &e[i + t];
4290                 if (!media) c->acts.tokens[t] = x->tok;
4291                 else if (x->rows) {
4292                     for (int k = 0; k < E; k++) c->stage[(size_t)t * E +
      k] = f16_table[x->rows[(size_t)x->row * E + k]];
4293                     c->ids[t] = 0;
4294                 }
4295                 else {
4296                     dequant_row(c->m->tok_embd, x->tok, c->stage + (
     size_t)t * E), c->ids[t] = x->tok;
4297                     for (int k = 0; k < E; k++) c->stage[(size_t)t * E +
      k] *= c->m->embd_scale;
4298                 }
4299             }
4300             if (media) c->acts.embd = c->stage, c->acts.pl_ids = c->ids;
4301         }
```

Listing 16.3: `gemma4.c:4262–4301`

The scan ends at the first unequal entry or when either list ends. If the new rendering exactly equals

Draft book, October 2026. Not peer reviewed.

the cached list and no logits are requested, there is nothing to run. But if logits are requested, as they are before a reply, the last position has to run again: that forward call produces the scores used to choose the first reply token. Reusing every prefix position would leave no fresh logits for the current prompt ending.

Media can make the exact mismatch position more subtle. A `whole` media entry, used by the bidirectional media mode, tells the loop that its rows should be passed together. E4B marks its image rows causal, so this rewind mainly protects the other template path. If a cache restart points into the middle of such a part, the start is rewound by the row offset. The batching code then converts those rows from the stored F16 table into staged activations. For ordinary token-only input, the token IDs can go directly into the model's token path. If a batch mixes media rows and ordinary IDs, the engine builds a shared staging array and fills in the per-layer token IDs as well.

The batch limit is `T_max`. The loop builds one forward call at a time and synchronizes before reusing the token and staging buffers from the previous call. That wait is a lifetime rule: the GPU must finish reading an input buffer before the CPU overwrites it for the next call. Chapter 15 described the token-buffer race that made this kind of ordering observable. Here, the same principle applies to chat's staging arrays.

The returned `reused` count is not simply the raw number of equal entries. `state_keep` may reduce the restart point to zero if a sliding-window ring has overwritten cache rows needed to continue at that position. The state helper says that when the ring is smaller than the full context, it can only continue if the required earliest window position remains within the retained range. In that case, a long matching transcript is not enough: the cache itself no longer has the needed state, so computation restarts from the beginning.

After the batches are prepared, the loop calls ‘forward‘ with ‘T-1‘ as the requested output row, so logits come from the final row at the current prompt boundary. It advances by the entries sent in that batch, synchronizes after the suffix is complete, and returns the reusable count. This is the handoff between the transcript algorithm and the forward path covered in Parts II and III.

This is why prefix reuse is a validity test, not an estimate of how many tokens are familiar. A full-context cache can continue at a matching boundary. A bounded sliding ring may reject that same boundary after enough positions have passed. A correct implementation must ask both questions: does the new sequence match, and does the cache still contain the state needed from that point?

What happens when the user edits history?

Consider a rendered prompt `A B C D E F` and a cache for those six entries. If the user changes the message that produced `D`, the next prompt might be `A B C X Y`. The first three positions are reusable; `D` no longer matches `X`, so the cache is reset to position 3 and `X Y` are run again. The old cached states for `D E F` cannot be spliced onto the new branch because they were computed from the old prefix.

Undo performs exactly this sort of branch change. The command removes the latest user/reply exchange from the message list, renders what remains, and counts the still-matching entries. It truncates the cache description to that common prefix. It need not explicitly zero every later KV value: the next forward pass overwrites the positions it will use, and `n_past` is set to the retained boundary before running. The cache's logical length is the relevant control; stale bytes beyond it are not part of the conversation.

A small but important exception appears when media are freed. A media entry stores a pointer to its row data. If the engine frees that memory before finding the last common prefix, an allocator could later reuse the same address for a different image. Pointer equality could then falsely claim a match. The code therefore renders and shortens the cached prefix before freeing discarded media rows. This is ordinary memory lifetime discipline applied to the cache key.

Why does the last position always run?

When a user's new message has been appended, the rendered sequence ends with the open model prefix. The cache may already contain much of the conversation, but the engine needs fresh logits for this current

PREPRINT

end position. Thus, if the prefix scan reaches the entire rendered list, it backs up by one before starting the forward pass. That final position is recomputed, after which argmax can pick the first reply token.

This behavior also explains the common phrase ”only the new words are prefetched”. It is a useful shorthand, but the actual count can include template markers, the open model prefix, and the final position that must run to make logits. The prompt report prints both total rendered prompt tokens and new prompt tokens. The difference is the reused count that survived the cache-validity check.

The engine has two distinct uses of its cache bookkeeping. During prompt rendering, `cached` describes the exact entries represented by KV state. During generation, newly selected reply token IDs are appended to that cached sequence, and the reply text is added to the transcript. On the next turn, rendering the transcript should reproduce those generated IDs under the template. The helpers for user text and reply text therefore need to agree with how output tokens are detokenized and later tokenized.

A cache ledger for three turns

It helps to write down what each list means after every transition. For a tiny example, let the first rendered user prompt be `B U h e l l o E M`, where `B` is an optional beginning marker, `U` starts a user turn, `E` ends it, and `M` opens the model turn. Assume the model emits two token IDs, `Y o`, and then the closing marker is included when the following rendering is built. The actual tokenizer may combine several of these characters into one ID; the labels here stand for entries, not claimed vocabulary IDs.

Moment	Transcript / new rendering	Cache after work
First prompt	`B U Hello E M`	`B U Hello E M Y o`
Second user line	`B U Hello E M Y o E U Again? E M`	Prefix through the prior reply; new suffix is the second user turn and open model prompt
After second reply	Prior rendering plus generated IDs	The second reply IDs are appended after the prompt entries
After undo	The transcript ends at the first user/reply exchange	Common prefix retained; second exchange tail is no longer logically cached

The table separates three moments that can be easy to blur. A prompt rendering ends at an open model marker and requests logits there. A completed reply adds generated IDs to the cache, even though the transcript stores a detokenized string for display and later re-rendering. Undo edits the transcript and marks the cache tail invalid. The next prompt call begins at that retained boundary and fills the new branch.

The source implementation uses real vocabulary IDs, not the labels in the table. The example also ignores the optional system turn, thinking marker, BOS convention, and media entries to make the state transitions visible. Those details change the rendered sequence, but not the prefix rule. Every entry before the first mismatch remains a candidate for reuse; every entry at or after the mismatch must be recomputed.

A diagnostic log can make this ledger concrete. After each event, print the message count, rendered entry count, cached entry count, chosen reuse boundary, and the number of new prompt entries. If those values are logged together, a report that says ”few new tokens” can be checked against a growing transcript. The engine’s normal report already gives total prompt tokens and new prompt tokens for one-shot chat; a standalone m8 exercise can expose the remaining internal counts.

What exactly changes at the prefix boundary?

The comparison rule is easiest to check with six entries. Let the old cached sequence be `a b c d e f`, and suppose a new rendering is `a b c x y z`. The comparison accepts `a`, `b`, and `c`; `d` and `x` differ, so the retained count is three. After this point, even an entry that happens to match later cannot be reused, because it was computed after the old `d`. Matching is contiguous from the beginning.

Now suppose the new rendering is exactly `a b c d e f`. If the caller only wants to synchronize the cache and does not need logits, it can report the full prefix as reusable. A reply caller does need logits, so

PREPRINT

it backs up to the final prompt position and runs it again. If `f` were the final open-model entry, the fresh result at that position becomes the first-token score vector. This is why a fully matching prompt can still perform one prompt position of work.

The sliding-window validity check can change the answer again. Assume the rendered prefix matches for 900 positions, but the ring cache has overwritten the first position needed to continue. The source's `state_keep` helper returns zero when its retained-window condition fails. The safe prefix then becomes empty, even though every token ID compares equal. This can look like lost reuse, but it is the right result: the current hidden state cannot be continued correctly from rows that no longer exist.

Here are four distinct cases and the correct conceptual action:

Situation	Rendered-list comparison	Cache action
Append a new turn	Existing transcript remains a prefix	Reuse to the previous end; run the new markers, text, and prompt end
Edit an old turn	First mismatch is inside earlier history	Keep only the earlier common prefix and recompute the rest
Undo the latest exchange	New rendering is shorter	Truncate to the new common prefix; later bytes become unreachable state
Ring lost required rows	Entries can all match, but state is incomplete	Reject continuation and restart from position zero

The source does not hash the whole transcript and hope for a match. It performs direct entry comparisons. That has a useful property: it identifies the exact earliest invalid position with no hash collision question. It also means the current work includes a linear scan of cached entries on the CPU. The outline's central cost claim is about model prefill work, which often dominates this scan; a benchmark should measure the scan separately if a very long transcript makes it visible.

Why are cache positions more fragile than message strings?

A message string is content, but a cache row is the result of a specific computation at a specific absolute position. Position affects rotary embeddings, causal visibility, and which earlier keys and values a token can see. If an earlier turn disappears, later text moves to different positions and has different preceding context. Its old cached state is not a portable representation of that text.

This explains why a policy that "drops the oldest turn" is not just deleting a string and shifting an index. The model's later states already incorporate that turn. To keep a correct state, the implementation must choose a valid restart point and recompute the retained suffix under the new positional sequence. A more advanced context manager could preserve a prefix before the dropped exchange, but it still must rebuild what follows. The simple implementation's '/clear' avoids this complication by setting the reusable length to zero.

For the same reason, the engine does not concatenate two independently computed conversations just because their visible words line up. The hidden rows for the second segment depend on the first segment's context. A valid cache prefix is a chain of computation from one exact beginning, not a bag of reusable sentence embeddings.

How does a reply become part of the next prompt?

The reply path renders the conversation and first checks whether the prompt alone fits. It runs the changed prompt suffix, takes argmax from the resulting logits, and clips the requested generation allowance to the cache space left after the prompt before decoding. It then detokenizes the emitted IDs to make a transcript message and appends those same IDs to the cached-entry list. This pairing is central: the human-readable history and the machine cache must describe the same generated sequence.

```
4312 static int chat_reply(Chat *c, int N, int verbose) {
4313     Entry *e;
4314     int n;
```

DRAFT

```
4315      chat_render(c, &e, &n);
4316      if (n + 1 >= c->n_ctx) {
4317          free(e);
4318          fprintf(stderr, "the conversation (%d tokens) doesn't fit in the
          context (%d); /clear starts over\n", n, c->n_ctx);
4319          return 0;
4320      }
4321      chat_stopped = 0, chat_generating = 1;
4322      const double t0 = now();
4323      const int reused = chat_run(c, e, n, N, 1);
4324      gpu_sync();
4325      const int first = argmax(c->logits, c->m->n_vocab);
4326      const double t1 = now();
4327
4328      if (N > c->st.n_ctx - n) N = c->st.n_ctx - n;
4329      ChatPrinter pr = {c->tk, c->stop, xmalloc((N + 1) * sizeof(int)), 0,
          N, isatty(STDOUT_FILENO),
4330                        token_id(c->tk, "<|channel>", 10), token_id(c->tk,
          "<channel|>", 10), 0};
4331      if (N > 0 && chat_print(&pr, first)) decode(c->m, &c->acts, &c->st,
         first, N - 1, 0, c->logits, c->out, chat_print, &pr);
4332      chat_generating = 0;
4333      const double t2 = now();
4334      if (pr.thought) fputs("\033[0m", stdout);  // (the reply ended while
          thinking)
4335      fputc('\n', stdout);
4336      fflush(stdout);
4337
4338      // the cache now holds the rendered conversation and the reply's
         tokens
4339      char *text = xmalloc((size_t)pr.n_reply * 64 + 1);
4340      size_t len = 0;
4341      for (int i = 0; i < pr.n_reply; i++) len += detokenize(c->tk, pr.
         reply[i], text + len, 64);
4342      text[len] = 0;
4343      chat_add(c, 0, text, (Parts){0});
4344      free(text);
4345      free(c->cached);
4346      c->cached = e, c->n_cached = n, c->cap_cached = n;
4347      for (int i = 0; i < pr.n_reply; i++) push_entry(&c->cached, &c->
         n_cached, &c->cap_cached, (Entry){pr.reply[i], NULL, 0, 0});
4348      c->st.n_past = c->n_cached;
```

Listing 16.4: `gemma4.c:4312–4348`

The code computes `reused` before reporting prompt work. After caching the reply, lines 4349–4356 print either the verbose one-shot report (total prompt tokens, new prompt tokens, time to first token, and generation rate after the first token) or the compact interactive report. The interactive loop reports new prompt tokens and elapsed time for that prefill. These are not all equivalent measures: a new-token count says how much prompt computation was done, while elapsed time also reflects the machine and current load.

The first generated token comes from the logits produced by `chat_run`. Remaining tokens go through the decode helper, which calls the printer as each token is chosen. The printer handles stop tokens and visible text; the chat layer collects emitted token IDs. At the end, detokenization creates text for future rendering, while the IDs become the cache tail directly. This avoids requiring the next turn to infer what the model said from its own printed text alone.

There is a finite context bound. If the rendered prompt alone does not fit (`n + 1 >= c->n_ctx`), the

reply path prints an explanatory message and returns failure. If only the requested generation allowance overflows, it clips `N` to `st.n_ctx - n` (`gemma4.c:4328`) instead of failing. On prompt-fit failure, the interactive loop forgets the just-added message and its media. It does not silently clip the oldest part of the conversation; a conversation exceeding the prompt limit is not shortened automatically.

The cache begins smaller than the maximum context and grows when a call needs more room. Its target is the larger of twice the current size and the immediate requirement, capped at the configured maximum. Doubling avoids resizing on every small increase, while the cap respects the user's context setting. The new state also accounts for slack needed by the sliding-window implementation and the turn's rollback room. This is storage management, not prompt truncation: the logical history remains intact until the configured bound is reached.

Why does a second turn often rerun a little more than the user typed?

The template contains more than the literal message text. It adds the user turn marker, the ending marker and newline, then the open model turn. The final position is recomputed for logits. So a report of, for example, "12 new prompt tokens" could include those structural tokens as well as the user's words. The exact count depends on tokenization and the conversation's rendered suffix; do not equate it with the word count.

The template may also render a previous model reply differently from its original output if special channel markers were streamed as control tokens. The printer recognizes thought-channel markers, while ordinary reply content is detokenized. The source's helpers define how a reply is represented for a later render. For debugging, compare the actual tokenized rendering and the cached IDs, not just the text printed in the terminal.

A warm conversation can still have a large total rendered size even while new prompt work is small. The total prompt count describes the current sequence length. The new count describes the suffix sent through forward. Generation cost is a third quantity: the number of reply tokens and the work of repeated decode. Keeping these three measures apart prevents the misleading conclusion that an ever-growing transcript is free simply because most of its prefill is reused.

The interactive loop

What do /undo and /clear really change?

The loop reads one line at a time. Ordinary input is appended as a user message and passed to the reply path. Commands can instead edit the transcript, attach media, prefill an attachment, or end the session. The command excerpt shows the state changes for undo, clear, and prefill.

```
4419          if (!strncmp(t, "/undo", 5)) {  // drops the attached media, or
      else the last exchange
4420              gpu_sync();
4421              if (media.n) chat_forget_pending(c), parts_free(&media),
      fprintf(stderr, "(media dropped)\n");
4422              else if (c->n_msg >= 2 && !c->msg[c->n_msg - 1].user && c->
      msg[c->n_msg - 2].user) {
4423                  // the cache keeps what the conversation without them
      still starts with (cut before their media's
4424                  // rows are freed, whose addresses may come back for new
       ones)
4425                  c->n_msg -= 2;
4426                  Entry *e;
4427                  int n, p = 0;
4428                  chat_render(c, &e, &n);
4429                  while (p < n && p < c->n_cached && same_entry(e[p], c->
      cached[p])) p++;
```

PREPRINT

```
4430                c->n_cached = p;
4431                free(e);
4432                for (int k = 0; k < 2; k++) parts_free(&c->msg[c->n_msg
    + k].media), free(c->msg[c->n_msg + k].text);
4433                fprintf(stderr, "(undone)\n");
4434            } else fprintf(stderr, "(nothing to undo)\n");
4435            continue;
4436        }
4437        if (!strncmp(t, "/clear", 6)) {
4438            gpu_sync();
4439            for (int i = 0; i < c->n_msg; i++) parts_free(&c->msg[i].
    media), free(c->msg[i].text);
4440            parts_free(&media);
4441            c->n_msg = 0;
4442            c->n_cached = 0;  // (freed rows' addresses may come back
    for new ones: nothing cached may match)
4443            fprintf(stderr, "(new conversation)\n");
4444            continue;
4445        }
4446        if (len == 8 && !strncmp(t, "/prefill", 8)) {
4447            chat_prefill(c, media);
4448            continue;
```

Listing 16.5: `gemma4.c:4419–4448`

Undo first synchronizes outstanding GPU work. It either drops currently attached, unsent media or removes the last user/reply pair. In the latter case it renders the remaining conversation and shortens `n_cached` to the common prefix. A new question can then branch from that earlier state. The cache is not rolled back by subtracting a guessed number of tokens; the template is run again and compared position by position.

Clear also synchronizes before it frees message media and text. It sets both the message count and cache-prefix count to zero. The underlying state buffers can keep their allocation, but no old positions are considered valid. This distinction between allocation and logical contents is common in high-performance code: keeping memory is cheap when it is reusable, provided its old meaning is no longer trusted.

The line-oriented interface has a practical boundary: one message per line. This is documented in the README. Newlines inside a message are therefore not accepted as a multiline prompt through the ordinary terminal path. The ‘/image‘, ‘/audio‘, ‘/video‘, and ‘/frame‘ commands attach media to the next message; an empty line can send attached media alone. These options are a small shell-like command language around the same transcript and renderer.

What does cache growth protect?

The configured context is a maximum, not necessarily the initial allocation. A new call asks whether the current cache can hold the rendered prompt, requested output allowance, and an extra position needed by the bookkeeping. If not, it grows. The target doubles the existing size unless the immediate requirement is larger, then clips that target to the configured maximum. For example, if the current allocation has 1,024 positions and the call needs 1,300, doubling to 2,048 is enough unless the configured maximum is smaller. This is an arithmetic illustration of the growth rule, not a measured allocation size.

Why double instead of allocate exactly what is needed every time? Conversations usually grow over multiple turns. Exact growth on every small increase would repeatedly replace or extend state structures. Doubling creates headroom for the next turns and makes growth less frequent. The code also asks for slack related to the sliding-window call size and rollback allowance. The ring cache is a storage layout; the transcript and configured limit remain the logical conversation policy.

Growth does not solve a prompt that is too large. If the rendered prompt alone cannot fit even at

PREPRINT

the configured maximum, the reply fails; excess generation allowance is clipped, not failed. On prompt-fit failure, the command loop removes the message that was just attempted and frees its media. The previous exchange remains. This is easier to reason about than silent truncation, because the user can decide whether to clear or edit the conversation.

A useful design exercise is to distinguish four lengths: total rendered prompt, valid cached prefix, prompt suffix to run, and requested reply allowance. The first two are not interchangeable. The suffix is what must be prefetched now, while the allowance is future capacity required for generation. A prompt can be short in new work but too long in total context because it has a long reused prefix.

Can the engine work while the user is still talking?

For live video, encoding frames can take time. The ‘/prefill‘ command places currently attached media at the start of a pending user message, renders that partial message, and runs through the last media row while leaving the text unsent. It then saves the rendered entries and cache length. When the user sends the text, the prefix scan can reuse the media work and run only what follows it.

The important word is ”pending”: media are attached to the next message, not added as a completed exchange. The prefill path temporarily constructs a user message so the regular renderer places its media markers and rows correctly. It removes that temporary message from the transcript, then cuts the rendered entry list just after the media. The resulting cache describes work already done, while the attached media object remains available for the eventual message.

This is a latency optimization around encoding and prefill, not an answer generated before the user is finished. It is also another reason to treat the transcript and cache as separate objects. The transcript can be incomplete while the cache has valid rows for the beginning of its next message. When the actual text arrives, the new rendering must still match that cached prefix for reuse to be correct.

The implementation of prefill makes the temporary-message trick explicit. It invokes the normal add and render helpers, then removes the temporary transcript entry while retaining the rendered media prefix that has already been run.

```
4374 static void chat_prefill(Chat *c, Parts media) {
4375     if (!media.n) { fprintf(stderr, "(nothing to prefill)\n"); return; }
4376     chat_add(c, 1, "", media);
4377     Entry *e;
4378     int n;
4379     chat_render(c, &e, &n);
4380     c->n_msg--;  // (the message isn't sent: its media stay attached)
4381     free(c->msg[c->n_msg].text);
4382     int cut = n;  // up to the last media row: the rest of the message
        isn't known yet
4383     while (cut > 0 && !e[cut - 1].rows) cut--;
4384     if (cut + 1 >= c->n_ctx) { free(e); fprintf(stderr, "(doesn't fit in
         the context)\n"); return; }
4385     const double t0 = now();
4386     const int reused = chat_run(c, e, cut, 0, 0);
4387     free(c->cached);
4388     c->cached = e, c->n_cached = cut, c->cap_cached = n;
4389     c->st.n_past = cut;
4390     fprintf(stderr, "(prefilled %d tokens in %.0f ms)\n", cut - reused,
        1e3 * (now() - t0));
4391 }
```

Listing 16.6: `gemma4.c:4374–4391`

The cut point walks backward from the rendered end to the last entry that contains rows. The cut keeps everything through that last media row entry, including the opening media marker and all earlier rows, and omits the closing media marker plus any following text or open model prompt that is not yet known to belong to the user’s message. If the required media prefix is too large for context, it reports that

fact and leaves the attachments pending. Otherwise it passes the cut prefix through the same reuse-aware runner, replaces the cached-entry description, records the resulting cache length, and reports how many entries were actually new.

The `reused` result matters when prefill is repeated or the earlier conversation already contains part of the same pending sequence. The report is the cut length minus that retained prefix. This is a count of entries processed, not a count of video frames or seconds. For real-time use, the README gives examples of frames being run while a person is still speaking, but those timing numbers are historical examples and are not re-measured by this writing task.

One implementation detail is worth noticing: the temporary message's text allocation is freed after the message is removed, while its media parts remain owned by the caller's pending-media structure. The cached entries point to the media rows, so those rows must stay alive through the eventual send or cancellation. This ownership split is why the function takes a copy of the media descriptor while the main loop retains the original pending parts.

The media lifetime rule applies here too. If the user removes the pending media or clears the conversation, the engine first shortens the cached prefix while the row pointers still identify the old data, then frees the rows after GPU work is complete. An asynchronous kernel reading rows while they are freed would be a use-after-free even if the message list itself had already been edited.

What does Ctrl-C leave behind?

The chat installs a SIGINT handler that marks generation as stopped. It does not attempt to perform complex GPU or allocator work inside the signal handler. The reply printer checks the stop state as tokens arrive; the normal reply path then finishes bookkeeping for the emitted prefix. The source describes interruption as stopping the reply while keeping what was generated. That means a partial answer becomes part of the transcript and cache, and the next user turn can continue from it.

This behavior is different from undo. Ctrl-C accepts the generated prefix so far. '/undo' removes the most recent exchange, including the reply. A user can therefore interrupt a long answer and ask a follow-up about the partial response, or use undo if the partial answer should be discarded. Both operations preserve the same core invariant: before continuing, the cached list must be a prefix of the newly rendered transcript.

Signal handling deserves special care because it can arrive while a GPU command is queued. The chat loop's ordinary control path and synchronization points are where buffers and cache lengths are reconciled. The handler only sets flags; it does not free media, change the cached count, or print a new prompt from signal context. That narrow job avoids re-entering the rest of the program at an arbitrary point.

Experiment: observe a reusable prefix

W0: start the snapshot without weights

The snapshot build and help output check the executable path without opening a model. Run from the book root:

```
nice -n 10 sh code/snapshot/build.sh
code/build/gemma4 --help > code/build/ch16-help.txt 2>&1
```

The build completed successfully during this writing session. The help command printed usage text and did not read model weights. The captured help lists the `--chat` option and `-c n_ctx` context argument. The full output is in `code/build/ch16-help.txt`; it shows the interface without treating this as a model run. This is W0: it confirms the binary and interface start, not chat correctness or generation speed.

PREPRINT

W1: read only the GGUF header

A standard-library Python probe read exactly the first 24 bytes from the local E4B GGUF: magic, version, tensor count, and metadata count. It did not load the model, parse vocabulary strings, allocate engine state, or run inference. The captured result was:

```
file=gemma-4-E4B-it-IQ4_XS.gguf
magic=GGUF version=3 tensor_count=720 metadata_count=56
```

This is header evidence only. It establishes the container version and declared counts, not the contents of any tensor or the model's selected template. A later metadata-only parser could inspect the template key, but this chapter's W1 probe intentionally stops at the fixed header. The header reader and captured result are saved in `code/build/ch16_gguf_header.py` and `code/build/ch16-gguf-header.txt`; no weights were mapped by this check.

W2: interactive cache-prefix check (not run)

Try it on your machine (W2): in a scratch copy, start chat and make two turns, undo, and branch with a different question:

```
code/build/gemma4 -m $MODEL --chat -c 4096
> What is 2 plus 2?
> Why?
> /undo
> Explain it with a small example.
> /exit
```

Try it on your machine (W2): record prompt length, new prompt tokens, and prefill time for each turn. The shared second turn should reuse its prefix; after undo, reuse should stop at the first changed entry. Record run conditions; counts alone are not performance measurements.

For a separate media run, attach a video and issue '/prefill' before sending the text. Compare the first message's new prompt count and elapsed prefill time with a run that sends it without early prefill. This is also W2 and is a reader experiment. Media encoding requires its associated encoder file. The experiment should show reuse in the new-token count; it does not prove that encoding and prefill overlap perfectly with real-time video capture.

How should the capture be read?

A useful chat log has more than one number. The total prompt length describes the rendered sequence for this turn. The new-prompt count describes entries that the runner actually sent after cache matching and any sliding-window reset. Time to first token includes the work needed to reach the first generated ID. Generation rate after that first ID describes repeated decode steps. These figures answer different questions, so a report that lists only tokens per second cannot tell whether a slow turn spent its time rebuilding prompt state or decoding the answer.

Observed item	Meaning in this chat path	Common misreading
Total prompt tokens	Current rendered history, markers, and open model prefix	Work repeated this turn
New prompt tokens	Suffix sent through forward after reuse checks	Number of user words
Time to first token	Prompt run plus first selection	Pure prefill kernel time
Generated tokens	IDs emitted by the printer before stop or limit	All IDs requested by the limit
Decode tokens/s	Rate after the initial token in the report	Whole-turn latency

PREPRINT

The W2 protocol should record the lines printed to stderr for each turn and the exact sequence of actions that produced them. First send one short question. Then append a follow-up without editing history. Next undo and send a different follow-up. This gives a baseline, an append case, and an edited-branch case in the same process. It is more informative than running three unrelated prompts because the cache's starting state is explicit at each transition.

For each case, keep the prompt text fixed across repeated captures and change one factor at a time. If comparing normal send with '/prefill', keep the attached media and final message identical. Record whether the new-token count changes, then compare wall time under the same quiet-GPU conditions. A reduction in new prompt entries demonstrates that work was reused; a shorter wall time is a separate observation that can vary with GPU contention, file paging, thermal state, and encoder time.

The chat output's reported prefill interval is bounded by the code's timer around prompt execution and first-token selection. It is not a stopwatch for the user's entire conversation turn, and it does not include time spent thinking of a question. Likewise, generated rate excludes the first token because the prefill logits produce it. When comparing designs, preserve these boundaries instead of recomputing one overall rate from unrelated intervals.

A cache-prefix test is first a correctness experiment. After undo, confirm that the alternate reply is produced from a valid prompt path; compare it with a fresh conversation replayed from the beginning. Only after the cached and fresh paths agree should their timings be interpreted as a speed difference.

A template mismatch and two lifetime traps

Pitfall

Symptom. Thinking-off chat could start with the wrong model-turn prefix, changing how later text was interpreted.
Cause. The 26B chat template opens a model turn with an empty thought channel, but E2B/E4B do not; when that empty thought is supplied to E4B, the model writes its thinking after it.
Fix. Read the GGUF's template choice and render the corresponding prefix. The snapshot README records this under "The model file" (`README.md`, line 209).

A template mismatch is not merely decorative. The control tokens are part of the model input. If the renderer makes a wrong empty thought, every later position is shifted or changed, and cached hidden states from the previous rendering cannot be reused past the mismatch. This is why the template option is stored in chat state and applied in the renderer, rather than appended as a terminal-only flourish.

The snapshot README "/undo" note (lines 153–155) records a separate media-lifetime constraint: before freed media are discarded, cut the cache at the point where their rows start. Row addresses can be reused by a later allocation, so leaving the old cached entries around could make pointer-based matching accept unrelated data. The fix is to compute the surviving prefix before freeing the media. This ordering is visible in the comments on `chat_forget_pending` and the undo path.

A third documented limit is that the interface accepts one line per message and does not shorten a conversation whose rendered prompt cannot fit. The reply function reports that prompt overflow and the loop forgets the unsent message and media; an oversized generation request is clipped to the available space. This is an explicit behavior rather than a hidden truncation policy. A future context manager would need to define which turns to drop and then rebuild a valid prefix from the retained transcript.

Exercises

Exercises

1. Render a two-turn text exchange by hand. Mark the role, end-of-turn and open-model markers before tokenizing ordinary text. Which entries are cached after the first reply?
2. Add a diagnostic command that prints the current number of cached positions. Decide whether it should count the open model prefix and explain how you would keep the count consistent after undo and clear.
3. Design a policy that drops the oldest complete user/reply exchange when the context is full. Explain why positions after the removed exchange cannot simply retain their old cache states.
4. For a transcript with media rows, list the safe order for shortening the cached prefix, synchronizing GPU work, and freeing the media. What can go wrong if the allocator reuses a row address too early?
5. Compare a second-turn prompt's total length, new-token count, and reply-token count in a W2 run. Explain what each count includes and record wall time separately from GPU time when available.

Rebuild checkpoint m8

How would a small test catch a wrong reuse boundary?

A useful unit test needs only entry arrays and a deterministic fake state. Give the renderer a fixed token list for a first turn and assert that the fake forward function sees every prompt entry. Record the cached entries including two generated IDs. Render a second turn with that exact history plus a new user suffix. The comparison should stop at the old cache length; the fake forward function should see only the suffix and the prompt end needed for logits.

Then mutate one old token near the middle. This is stronger than only testing append: a faulty algorithm that compares the new list length but ignores contents would pass the append case and incorrectly reuse the mutated branch. Assert that the retained count equals the number of entries before the mutation, and that every subsequent fake state is recomputed. Also place a matching token after the mutation to verify that matching later content does not restart reuse.

Undo is a third case. Remove the last two messages, render again, and assert that the cached count is no larger than the resulting rendered list. A no-logits operation may keep the full matching list, while a reply operation must rerun the final prompt entry. Those should be separate test cases because the same rendered sequence has different requirements depending on whether the caller needs a next-token score.

Media rows deserve two synthetic cases. First, represent a media part by a marker followed by three row entries and then a closing marker. Force the mismatch at the second row while the part is marked whole; the runner should back up to the beginning of that part before batching. Second, make two entries point to different row objects with the same fake values; direct identity comparison must reject them. This checks the engine's chosen equality rule without loading an encoder or image file.

Finally, simulate a bounded sliding cache whose state-keep predicate rejects a positive matching prefix. The test should prove the cache runner obeys the state result, not just the token comparison. A test can also track a fake absolute position so that recomputing after an earlier edit produces a different state than reusing stale downstream values. That catches errors which a final string comparison could conceal. Build a small standalone chat-prefix program around the idea in this chapter. It does not need a model or a tokenizer: represent each rendered text token as an integer `Entry`, render two short turns from fixed arrays, and compare the new sequence with the cached sequence until the first mismatch. Save only that common prefix, then run a fake forward function over the suffix. For the fake model, let the ”logit” at a position be a deterministic hash of the input token and prior fake state so that a mistaken suffix reuse changes the check value.

PREPRINT

The checkpoint is `steps/m8_chat_prefix.c`, described in PLAN.md §4.4. Test that the first turn computes its full prompt, an unchanged second-turn prefix costs only its new entries, and undo followed by a changed question recomputes exactly from the first changed entry. Include the open model marker in both rendered prompts; verify that the final prompt position is run when logits are needed. Then test a bounded-cache case that rejects an otherwise matching prefix and restarts from zero. The target is a testable cache-validity rule, not a miniature transformer.

What this chapter established

Chat history is the source used to build a model prompt; the KV cache is reusable only while it represents the same prefix. The engine renders the conversation again, compares entries, checks whether a sliding-window cache still holds the needed state, recomputes the suffix, and appends generated IDs and text in step. Undo, clear, interruption, and media prefill all become easier to reason about when that invariant is kept explicit. Chapter 17 leaves this text-only route for the optional vision encoder; the chat loop remains the same consumer of the resulting media rows.

PREPRINT

Part V

EYES AND EARS (optional; needs the encoder file; 56 pp.)

PREPRINT

Chapter 17

Seeing: the vision encoder

Part V: Eyes and ears (optional)

The first four parts followed text from its prompt to its reply. Part V follows two other ways of making prompt rows: pictures and sound. It is optional; the text engine in the earlier parts stands on its own. This chapter stays with one image and the encoder that turns it into rows the language model can read.

The question

How does a picture become 280 rows of the prompt?

Reading map: 20 pages

This first pass assigns the estimate before adding the detailed walkthrough. The map includes the diagrams, code, experiments, pitfalls, exercises, and a short bridge back to the prompt representation from Chapter 16.

Section	Pages
Part V opener, question, and image-to-row map	1
Resize, borders, pixels, and patch layout	4
Position information and two-axis rotation	3
One vision block: projections, attention, and MLP	4
Pooling, projection, and the 280-row example	3
Source walkthrough and clippable linears	2
W0/W1 experiments and the W3 reader experiment	1
Documented pitfalls and accuracy comparisons	1
Exercises, hints, and rebuild checkpoint	1
Total	**20**

The idea

What does the encoder hand to the chat loop?

The image path does not turn a picture into a caption. It turns the pixels into a sequence of vectors. Chapter 16 called each such vector an entry in the rendered prompt: an image marker, a run of image rows, and a closing marker. Here we open the encoder side of that boundary. The vision tower reads patches, updates their vectors through 16 transformer blocks, combines each 3-by-3 group, and projects each combined vector to the language model's width.

For the default budget, a prepared image has at most 280 output rows. The encoder can produce between 70 and 280. The exact count depends on the resized grid, and each row has width 2,560 when

PREPRINT

it reaches the E4B language model. Those are shapes, not a promise that every image always makes 280 rows.

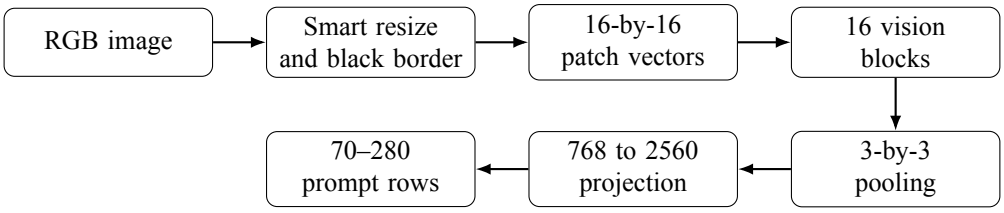


Figure 17.1: The vision encoder reduces a patch grid to a shorter sequence of language-model rows.

Why change the size before making patches?

The encoder needs a grid whose dimensions fit both its 16-pixel patches and its 3-by-3 merge. A side must therefore be a multiple of 48 pixels. The resize also limits work: the default output budget corresponds to between 70 and 280 merged rows. Later sections will count patches from a real shape and follow one row through the encoder.

The unit of this budget is a row after pooling, not a patch and not a source pixel. One output row represents nine patches. Since each patch covers 16 by 16 pixels, each row begins with a 48-by-48 region of the prepared image. The pixels within that region are still processed separately by the vision blocks before those nine patch vectors are combined. Pooling is late in the pipeline; it does not mean the encoder can skip eight patches out of every nine.

The maximum grid dimensions also have a second constraint. The learned x and y position tables contain 10,240 positions per axis. The loader checks that the column and row counts fit those tables before it starts the blocks. The default pixel budget is much smaller than that theoretical table limit, but the check protects non-default budgets and malformed metadata. The model file supplies most shape settings; the engine sets its 70-to-280 default budget and allows the command line to raise the maximum.

What happens to a wide photograph?

Imagine a 1,024 by 768 image. Its aspect ratio is 4 to 3, but neither side is initially a multiple of 48. The engine first rounds each side to an aligned candidate and checks the pixel-area budget. That candidate is too large for 280 output rows, so it scales down and rounds the sides down to aligned values. The actual W0 calculation later in this chapter gives 912 by 672. This is close to the original proportions and fits in the budget.

That fit is not achieved by stretching the photograph into an arbitrary rectangle. The engine computes one scale limited by the target width and target height, resamples to the resulting inner dimensions, and centers the result on a black canvas. If the target grid is slightly wider or taller than the scaled image, the leftover pixels become borders. Video frames take a different path and omit borders; this chapter is about still images.

The resampler is separable: it does a horizontal pass and a vertical pass. For each output coordinate it precomputes the source interval and a normalized bicubic weight row. The weights use 22-bit fixed-point integers, and the pixel accumulation rounds before shifting back to bytes. This makes the engine's pixels reproducible against the chosen reference implementation instead of depending on a platform image library's filter defaults. Resizing is CPU work; the transformer and projection use the GPU when available.

For large downscales, a source pixel can influence several neighboring output pixels. A nearest-neighbor resize would discard much of that information, while an ordinary single-pixel bilinear sample can alias fine patterns. The bicubic support widens with the downscale factor, so more source samples contribute to each destination. This matters for small print: the larger image-token budgets can preserve

PREPRINT

more detail because they retain a denser patch grid, but no resize can recover characters that the original image did not resolve.

How does one pixel become one patch coordinate?

After resizing, the image is a top-to-bottom RGB byte array. The patch grid is row-major. Patch number 0 is at the top left; moving one patch right increments its column, and moving past the last column starts the next patch row. Within a patch, the input vector groups the red, green, and blue planes, and within each channel visits rows and columns. That channel-row-column order is the order expected by the patch-embedding tensor.

Each 16-by-16 RGB patch contains 768 byte values. The model's patch matrix maps those values to a vector of width 768. The file stores this patch embedding in F16 for E4B, one of the model-specific details the generic linear helper cannot assume. Before the multiply, each channel byte is converted to a unit interval using the configured mean and standard deviation, then shifted and scaled into the interval from minus one to one. In this E4B encoder the mean is zero and the standard deviation is one, so the channel conversion does not add a learned color correction.

The resulting patch vectors form the residual stream. The engine keeps that stream in float for much of the CPU/GPU interface, with half-precision buffers for several projections and intermediates. That mixture is deliberate: it matches the existing matmul and attention helpers and limits some large activation arrays to half precision. The `VActs` structure makes the live buffers visible: patches, residual vectors, normalized input, q/k/v projections, prepared heads, attention output, MLP hidden values, pooled vectors, and final language-model rows.

For a 912 by 672 prepared image, dividing by 16 gives a 57-column by 42-row patch grid. That is 2,394 patch vectors entering the vision tower. Dividing each grid side into groups of three gives 19 groups across and 14 down, or 266 output rows. The image dimensions are not themselves the number of prompt positions. The model sees the shorter 266-row sequence after the tower and projection.

Why are there two kinds of position information?

The patch embedding alone says what colors and local patterns appeared, but the same patch contents at the top and bottom of a picture should not be treated as identical. The encoder adds one learned vector for the patch column and another for its row. These tables contribute absolute grid position to every channel of the patch vector.

Attention also needs a way to compare positions. In each 64-value head, q and k receive a two-axis rotary transform. The first 32 dimensions turn according to the column coordinate; the other 32 turn according to the row. Each half is rotated in 16 pairs. The base is 100, as loaded by the engine, and the angle for each pair changes at a different rate. Values are transformed in q and k; v receives RMS normalization but no rotation.

The two mechanisms do different jobs. The additive x/y vectors make an absolute location part of the residual stream. The rotary transform makes the attention score sensitive to relative offsets along both axes. If two patches move together by the same grid offset, their relative coordinate difference is unchanged, although their learned absolute position vectors can differ. This is an axial two-dimensional extension of the one-dimensional RoPE discussed in the model chapters.

There are 12 heads, each 64 values wide, for the 768-value stream. The Q and K projections are normalized per head using learned weights before rotation. V is normalized per head without weights. Attention has no extra inverse-square-root multiplier in this path: the implementation uses scale one, with the q/k normalization providing the intended score range. Scores compare every query patch with every key patch in the image. That full patch-to-patch attention belongs to the vision tower; after projection, the E4B language model consumes the resulting image rows in its ordinary causal prompt order.

PREPRINT

What does one vision block do?

The block begins with an RMS-normalized residual stream. Three learned projections produce query, key, and value vectors. The encoder prepares those vectors per head, computes full attention across the patch grid, projects the concatenated result, normalizes that branch, and adds it back to the residual. A second normalization feeds the gate and up projections. The gate uses the model's tanh GELU, multiplies element by element with the up branch, and passes the product through the down projection. A final post-branch norm and residual add complete the block.

The order is important. In particular, post-attention and post-FFN norms are on the branch output before it is added to the residual. This resembles the language model's pre- and post-norm structure, but the vision block has its own dimensions, positions, clamp metadata, and activation selection. It is safer to read the actual call order than to infer it from a generic "ViT block" diagram.

The same pattern repeats 16 times. Between blocks, the newly updated residual is normalized for the next block. The GPU path groups q/k/v work so the shared clamped input can be prepared once; the CPU reference follows the same mathematical sequence through `STEP` dispatch. `STEP` tries the Metal implementation and falls back to the CPU task if that GPU operation is unavailable. This is useful both for portability and for understanding which stage a comparison is testing.

At this point the number of sequence positions is still P, the number of patches. Full attention compares P queries against P keys for each head, so its work grows faster than the linear projections as images get larger. The row budget controls that cost indirectly: increasing the output budget raises the patch count by nine for each additional pooled row, and raises the attention matrix by the square of the patch count. The largest setting is useful for small print, but can cost substantially more encode work than the default.

Why clamp a projection if half precision has room?

The E4B encoder attaches input and output minimum/maximum values to most block projections. Before a matrix multiply, the input is limited to its stored interval; after the multiply, each output is limited to its own interval. These are parameters from the encoder file, not global constants guessed by the C code. Q, K, and V share an input interval, as do gate and up; the loader checks those pairs agree before allowing the engine to run.

The numeric ranges are far below the maximum finite F16 value, 65,504. That rules out the tempting explanation that the clamp exists just to prevent half overflow. The limits instead bound outliers to the ranges stored with the trained encoder. This makes the saved metadata part of the operator: a matrix with the same weight bytes but missing or misread clamp scalars is a different computation.

The engine represents an unclamped side with the largest finite float sentinels. That lets `load_linear` support ordinary projections and clippable projections with the same structure. The `clamp_in` helper applies the input range where needed; output clamps are incorporated in the matmul store path. In the GeGLU pair the gate and up results have distinct output ranges and are clamped separately before activation and multiplication.

How do nine patches become one language-model row?

When the 16 blocks finish, a pooled output position corresponds to one non-overlapping 3-by-3 patch group. The engine sums the nine vectors, scales their average by the square root of the hidden width, applies optional standardization values, and performs an unweighted RMS normalization. E4B has no vision standardization tensors, so the loader supplies a zero bias and unit scale. The RMS-normalized 768-value row then goes through the final projection to width 2,560.

The scale used before RMS normalization is a positive scalar shared by all channels. With the small epsilon used here, RMS normalization largely cancels it; the implementation still follows the common encoder path and keeps the scale explicit. The standardization tensors remain optional because the loader also supports another Gemma 4 vision configuration. For E4B, absence means the identity transform, not a fatal file error.

PREPRINT

The final projection output is the media embedding that Chapter 16 places between image markers. It is not a vocabulary token and is not retokenized from a textual description. The chat renderer treats each projected vector as one position, and the language model's token embedding path is bypassed for that row. As with other media, later cache reuse requires the same row data to remain the same live input; the chat layer owns that lifetime rule.

Consider the W0 example again. It makes 2,394 patch vectors for the vision blocks, then 266 pooled vectors for the projection. The final prompt contribution is 266 positions between the image markers. At width 2,560, those rows contain 266 times 2,560 scalar values. That count explains why a moderate-looking picture can use hundreds of prompt positions even though a human treats it as one attachment.

Stage	Shape in the example	What the count means
Prepared RGB	912 by 672	aligned image with aspect ratio retained
Patch grid	57 by 42	2,394 positions enter the vision tower
Grouped grid	19 by 14	each location represents nine processed patches
Projected rows	266 by 2,560	positions and width passed to the language model

What changes when the row budget changes?

The flag `--image-tokens` changes the maximum number of pooled rows. It is easy to mistake this for a direct request to create exactly that many rows. In the loader, the default is a minimum of 70 and a maximum of 280. When a user supplies a larger maximum, the lower bound stays at 70; a value below 70 becomes both the lower and upper bound. The geometry and aspect-ratio alignment then determine the final grid. Even when the maximum is fixed, a particular aspect ratio may not land exactly on the boundary.

Each pooled row requires a group of nine patches. A budget of 70 therefore starts from at least 630 patches, while a ceiling of 280 permits as many as 2,520. The 1,024-by-768 example produces 266 output rows because the aligned dimensions give 19 groups across and 14 down. These are different kinds of "maximum": pixel area constrains the resize, whole patch groups constrain the grid, and a command-line value limits the output row count. The implementation stores the budget in `Vision.min_tokens` and `Vision.max_tokens`, and `smart_size` receives pixel bounds derived from those values.

The largest supported setting is useful when tiny letters dominate the task. It is also a significant compute choice. The patch projections and feed-forward matrix operations grow in proportion to the number of patches. Attention compares every query against every key; doubling the patch count therefore makes the attention score grid about four times as large. The final row count is a useful user-facing budget, but the encoder performs most of its work before pooling, on nine times as many positions.

This creates a practical reason to choose the smallest budget that preserves the information needed. A scene-description question may be answerable at the default budget. Reading a small receipt or a distant sign may benefit from more image rows. The extra rows also become positions in the language model prompt, which means more prefill work after the encoder finishes. Media is not "free" just because it arrives as vectors rather than words.

Which image-file details does the loader actually trust?

The vision loader checks that the GGUF describes a CLIP-family file with the Gemma 4 vision projector identifier and that the encoder is present. It reads the hidden width, number of blocks, head count, epsilon, patch size, and projection width from metadata, applying defaults only where the format permits them. It rejects shapes outside the engine's supported limits. It then resolves tensors by name and checks their types and dimensions before constructing the layer array.

For E4B, the patch embedding has a special storage expectation: the loader accepts F16 or F32 and checks the output width. The book's W1 reader does not inspect that tensor descriptor, so its success cannot prove the patch matrix is present or correctly typed. That is a deliberate separation of checks.

PREPRINT

Header metadata answers "what architecture and dimensions does this file declare?" A tensor-table check answers "are the named tensors and their shapes available?" An encoder run answers "do the bytes produce the intended rows?" Each test is stronger and more expensive than the one before it.

The loader also treats the x and y position vectors as separate tables. It verifies the position tensor dimensions and binds the appropriate slices by axis. For every layer it loads q, k, v, output, gate, up, and down projections, together with the associated norms. Clamp ranges are optional fields beside many matrix weights. The loader checks that paired operations share their required input intervals. This prevents a malformed or incompatible file from silently turning a shared input buffer into an inconsistent operation.

That metadata-driven design keeps the C data structures compact. A `Vision` contains the dimensions, limits, normalization values, activation choice, position tables, projection, and layer array. A `VLayer` carries the block weights and norm vectors. The `Linear` structure wraps a tensor pointer and its four clamp bounds. These structures do not duplicate the matrix payload; they point into the mapped GGUF. This is the same no-copy principle introduced for the language model file, now applied to the media projector.

Why is the 12-megapixel picture not a tiny task?

A full-resolution photograph may be reduced to a grid of only a few thousand patches, but each patch is processed by wide matrix layers. At the 2,394-patch example size, the vision tower runs 16 blocks across width 768, with a 3,072-wide feed-forward hidden layer. Its full attention also compares all patch pairs. ENGINE-MATH's arithmetic for this example totals about 1,008 GFLOP in the vision tower, with about 723 GFLOP from the block matrix work and 282 GFLOP from attention. The final projection contributes comparatively little.

Those values are an analytical operation count in the snapshot's ENGINE-MATH notes, not a new benchmark. The README's historical measurement for a 12-megapixel image gives 266 rows, 0.56 seconds of encoding, and about 0.70 seconds of language-model prefill on an M3 Air with warm kernels. That table makes another important distinction: encode latency and prompt prefill are separate. The image encoder turns pixels into rows; then the language model reads all those rows through its own layers.

For context, one 266-row image also contributes a sizable sequence to the language model. The model's per-row compute is different from the vision encoder's per-patch work, and its KV cache grows with the input positions. Chapter 16 already described how these rows join the message and become cache entries. This chapter does not need to repeat the cache algorithm; it is enough to see that choosing a larger image budget increases both the encoder's work and the prompt's length.

Attention's quadratic cost gives a useful mental check. If a wide image is resized to twice as many patches in each dimension, its patch count becomes four times larger, and the attention score matrix becomes sixteen times larger. The actual smart resize does not always scale both dimensions by the same exact integer, but the square-law warning remains. A page with small text can justify that cost; an ordinary scene often cannot.

How does a block preserve useful information?

The residual stream gives every block a path to carry its previous representation forward. Attention adds a branch built from other patches. The feed-forward branch then applies a learned nonlinear transformation at each patch position. Post-branch normalization controls the scale before each residual addition. This sequence lets a block add a correction without replacing the entire vector with an unrelated new value.

The attention part begins with an input normalized by the block's first norm. The q, k, and v matrices each map from width 768 to width 768. The three projections are applied to all patches as a batch, not one patch at a time. This is where the same matrix-matrix machinery from the prefill chapters is useful: the encoder has many rows available, even though an individual image is a separate request.

For each query patch and head, the encoder forms one score for every key patch in that image. A stable softmax turns those scores into weights and the weighted values produce an output head. The CPU reference's attention task allocates a temporary score array for the patch positions and does the score and

DRAFT

value passes explicitly. The Metal path uses the engine's general attention kernels. Chapter 12 covered the general kernel's online softmax; the vision path uses a 64-dimensional head, which fits the older general kernel shape noted in the source history.

The concatenated head output goes through the output matrix and the attention post-norm. The result is added to the residual. The block then normalizes again and computes two feed-forward projections: a gate and an up branch. The gate's activation and the up branch combine elementwise, so each patch's hidden vector becomes a nonlinear feature vector before the down projection returns it to width 768. A final post-norm and residual add completes the block.

The activation deserves care because the file configuration matters. Gemma 4's vision encoder records the tanh GELU variant. The engine's `gelu_quick` field is an explicit compatibility option for a comparison implementation that defaults differently. The default is therefore not an arbitrary speed choice: it follows the model configuration. When an external converter drops that information, matching its output may require switching both programs to the same alternate activation, then clearly labeling that comparison.

What do the saved clamp ranges look like?

The E4B ranges vary by layer and operation. The recorded input limits for q/k/v range from about plus or minus 5.5 to plus or minus 20.8 across the 16 blocks. Their output ranges span about plus or minus 10 to plus or minus 25. The down projection is especially broad: its input range reaches about plus or minus 91, and its output range reaches about plus or minus 88. These values are summarized from ENGINE-MATH's inspection of the encoder metadata; the exact bound for a given matrix is stored with that matrix in the file.

The table below gives representative ranges to show scale and variation. It does not replace the per-layer values, and it does not claim that each row has the same bound in every block.

Projection group	Observed input range	Observed output range
q, k, v	roughly −20.8 to +20.8 at widest	roughly −25 to +25
attention output	roughly −3.5 to +3.5	roughly −21.4 to +21.4
gate and up	roughly −12.8 to +12.8	roughly −17.4 to +17.4
down projection	roughly −91 to +91	roughly −88 to +88

The ranges are symmetric in the summary because the recorded scalar minima and maxima were close to opposite values; the tensor remains the authoritative value. The q/k/v and gate/up sharing rules let the engine clamp one input array before multiple matrix multiplies. Output ranges can differ even when the input is shared, so they are applied to each result separately.

Do not infer that the limits are a generic numerical safeguard from the fact that the activation buffers use F16. F16 can represent magnitudes much larger than these ranges. The actual behavior comes from the weights and auxiliary scalar tensors in this encoder file. If a future conversion loses those scalar tensors, a loader default of "no clamp" may still allow a program to run while producing a different embedding. That kind of metadata loss is exactly why a comparison should include intermediate rows and not only the final generated answer.

What does `STEP` mean for an image?

The vision path uses the same CPU-reference/GPU-dispatch pattern as the rest of the engine. For operations with an explicit CPU task function, `STEP` asks the Metal wrapper to enqueue the work; if it cannot handle the operation, the CPU task runs over the required row count. The `vision_encode` layer loop combines these dispatches with common matrix operations. That means not all code inside the encoder is duplicated as a Metal kernel, and a CPU fallback is not necessarily a separate lower-quality algorithm.

The main difference between the reference and Metal preparation paths is representation and parallel assignment. CPU code iterates over every head in one patch task and stores each result as half. Metal gives a thread to a patch, head, and kind and computes the normalization and rotations in one kernel. Both

PREPRINT

Draft book, October 2026. Not peer reviewed.

paths have to agree on the dimensions, axis split, norm weights, rotary base, and half conversion. A bug in any of those can create a consistent-looking but wrong image embedding.

The fallback also makes source reading more informative than merely searching for kernel names. The CPU implementation tells the reader the operation's intended ordering in ordinary C. The Metal kernel then shows the parallel ownership and memory access. The layer loop reveals how input clamps and matrix dispatch are arranged around those kernels. These three views answer different questions: what math is intended, how threads divide it, and how data moves between stages.

Why did the image path once use so much memory?

The engine's README records a vision-specific attention problem. The general attention kernel looped over 32 output blocks with a bound that was known only at run time. The compiler kept those accumulators in per-thread memory, and the driver allocated that spill storage for the whole grid. During an image, the extra allocation reached 206 MB. After the loop bounds became compile-time constants, the 64-dimensional head needed only eight blocks, which stayed in registers; the reported image peak fell from 300 MB to 99 MB, and the result stayed bit-identical.

This is a concrete GPU lesson: a short local array or loop can multiply into a large allocation when the kernel runs across many threads. The source-level accumulator may look small, but the compiler's register allocation and spill strategy determine the real footprint. The fix was not to reduce image resolution or change the model; it was to make the kernel's actual head-size limits visible to the compiler. Source: snapshot `README.md`, "Memory", item "Vision attention without spills". These are historical project results, not a fresh measurement here.

Where does the encoder's temporary memory go?

`vision_encode` allocates its activation arrays for the patch count rounded up to a multiple of 32. The padding makes the row batches align with the engine's matmul tile. It allocates patch input as half values, the residual stream as floats, normalized inputs as half values, q/k/v projections as floats, prepared q/k/v and attention output as half values, and MLP hidden values as half. It also allocates pooled half rows and the final float output rows.

The scratch arrays are shared with Metal so the same buffers can be consumed by the GPU. After the final projection, the function synchronizes GPU work before freeing scratch. That synchronization is not optional housekeeping: freeing a shared allocation before its final command buffer reads it would turn a correct computation into a lifetime race. The returned embedding array is excluded from the scratch list, because the caller needs it to insert the image rows into the message.

Rounding P up to a multiple of 32 can add some unused tail capacity, but it doesn't change the logical patch count or row count. The loops guard against `P` and output lengths separately. The GPU preparation kernel checks whether the computed patch index exceeds P; the host allocation reserves padded space only to make batching convenient. Padding is capacity, not extra image content.

The README also reports that the general attention spill was the high-water event for an image. This illustrates why total model memory and transient activation memory need separate accounting. The encoder file's mapped weights are one category; scratch buffers are another; output rows retained by chat are a third. Chapter 19 will return to the lifetime of mapped media weights, while Chapter 16 already covered how retained rows affect cache state.

How should a row comparison be designed?

The command in the W3 box produces one answer, but it is not a complete encoder correctness test. A useful comparison dumps rows before the language model can obscure differences behind its own attention and vocabulary projection. The engine's `GEMMA4_DUMP_EMBD` format begins with the row count and width, followed by float values. An external path should produce the same input image, the same resize, the same row ordering, and the same activation before values are compared.

PREPRINT

First check the header values in the dump. If the row count or width differs, cosine comparisons by flat offset are meaningless. Then compare each row against its counterpart, recording cosine similarity and relative error. Inspect the worst row as well as the mean: an average can hide one badly corrupted location. For a quantization comparison, use the same encoder architecture and image, then state which file is the reference. For an implementation comparison, state the activation and preprocessing conventions explicitly.

Only after the rows agree should the end-to-end question be asked: does the model answer the same thing with the image? This is a separate check because near-tied next-token scores can turn small embedding differences into a later change of wording. Conversely, an unchanged caption does not prove the rows are equal. The layers of evidence are useful for different failures: metadata and tensor shape checks localize file issues; embedding comparison localizes encoder issues; a final response checks user-visible behavior.

The default E4B activation should be used when asking whether the engine follows the model configuration. The quick-GELU setting should be used only when the reference path also uses quick GELU. Mixing these goals in one test makes the conclusion hard to interpret. The README's cosine values are explicitly for the latter compatibility comparison; the book does not claim they are the default tanh-GELU comparison result.

Which facts are safe to take from the W1 probe?

The W1 result supports a narrow set of statements: the file identifies itself as a `clip` architecture, names the `gemma4v` projector, declares a vision encoder, and reports 16 blocks, width 768, 12 heads, and patch size 16. The reported tensor count is also in the header. These facts make it a useful first gate when an experiment fails before image processing begins.

The probe does not validate the tensor names, shapes, offsets, alignment, type codes, or payload. It does not read the image, initialize Metal, or compare a single embedding. The actual engine's model loader does more work, and only a successful encode exercises the complete path. Keeping the boundary explicit avoids turning a small metadata check into a claim about the 559 MB file's contents.

The reader itself is deliberately simple: it decodes GGUF strings and primitive metadata values using `struct`, recursively handles arrays, and skips no unknown field by seeking; it reads and materializes metadata values as it walks them. That is practical here because the encoder metadata is small and typed. It then stops as soon as the metadata table ends, before the tensor descriptor table. If a future GGUF adds a metadata type, the script should fail visibly rather than guessing its byte length.

What can go wrong at the image boundaries?

Several assumptions sit at the edges of this pipeline. Input files decode to RGB bytes in top-to-bottom order. The resize helper expects positive dimensions. The output canvas is zero-initialized, which is what makes padding black. The patch loop converts bytes into the expected channel order and handles each patch in the same coordinate convention used by the position lookup. The row count calculation assumes complete groups of three, which smart sizing ensures through the 48-pixel alignment.

A transposed width and height can still produce a plausible image because both are legal integers. It changes which patches are considered neighbors and changes both position coordinates. An incorrect channel order can preserve broad colors while changing the first patch projection. A missing border fill can leave allocator contents in padded pixels. Each bug may still make a vector of the correct width and a plausible caption, so shape checks should be followed by intermediate comparisons when diagnosing an image path.

The `stb_image` loader is asked for three channels even if a source file is grayscale or contains alpha. That keeps the later code's layout stable. The returned source channel count is recorded but the requested output layout is RGB. GIF input is another potential surprise: a decoder may choose one frame or a default frame, while the encoder receives only one RGB image. Multi-frame video is a separate media path handled in Chapter 19, not a hidden property of this still-image function.

PREPRINT

If an image is already exactly the target size, `image_prepare` copies it into the output buffer and returns without resampling. The output object still has its own allocation, so later cleanup is consistent. If only one dimension changes, only one resampling pass is necessary. If both change, the horizontal result becomes the input to the vertical pass and the temporary buffer is freed after the second pass. These lifetime details are mundane, but they prevent leaks and accidental reuse of a freed intermediate.

Why does full attention not contradict causal image rows?

There are two stages with two different attention masks. The vision tower uses full attention across the patch grid: every patch's representation can incorporate information from every other patch. This lets a patch-level feature use the whole image before the tower pools it. The output rows then enter the E4B language model as positions in a prompt. That language model reads the prompt causally, so each row can use earlier image rows but not future ones.

The phrase "image tokens are causal" describes the language-model prompt treatment, not the vision tower's internal attention. The 26B sibling has a different bidirectional image-row behavior in the language model, which affects how a media part is batched and cached. E4B does not use that path. Chapter 16 showed the `Entry` sequence and prefix reuse at the chat boundary; this distinction explains why the encoder can use full image context without changing the E4B chat cache contract.

This separation also clarifies why one image is not a single token. The vision transformer first makes patch states with global image context. Pooling merges adjacent groups into a shorter row sequence. The projection maps each row into the language model's width. The chat path then places all of these rows between marker entries. Attention within the encoder, pooling within the encoder, and causal consumption by the language model are separate operations with separate sequence lengths.

Can we trace one position through both axes?

Take patch index 100 in the 57-column grid from the W0 example. The quotient and remainder give row 1 and column 43. In one 64-value attention head, dimensions 0 through 31 use that column value; dimensions 32 through 63 use the row value. Inside each 32-value half, values are paired 16 apart. The engine therefore rotates the same head vector twice, with different position numbers and disjoint dimensions.

Patch position	Coordinate	Rotary part for each head
Index 100	column 43, row 1	first 32 dimensions use 43; last 32 use 1
One patch to the right	column 44, row 1	column rotation changes; row rotation stays the same
One patch below	column 43, row 2	column rotation stays the same; row rotation changes

This local example shows why both axis positions are carried through the prep kernel. If the column and row were swapped, a landscape image would not necessarily fail a shape check; it would simply be interpreted with the wrong spatial relationships. If only the first half were rotated, vertical neighbors would lose their positional signal in attention. The additive x and y tables provide a separate position signal over the full 768-value patch vector, so the rotary split is not the only location information in the block.

For two nearby patches in the same row, the row-dependent rotation uses the same coordinate in both q and k, while their column-dependent rotations differ. For a pair in the same column, the reverse is true. This is the useful intuition behind axial RoPE: each axis can express its own offset without flattening a two-dimensional grid into one long coordinate. The actual head vector contains learned values, so this picture explains where the position enters; it does not predict the resulting attention score by itself.

What does input normalization do to color bytes?

The E4B metadata gives zero channel means and unit standard deviations, so the image bytes are first treated as normalized channel values and then mapped into the model's signed input interval. A few byte

PREPRINT

values make the endpoints concrete. The mapping is applied independently to red, green, and blue before values are placed in the channel-row-column patch vector.

RGB byte	Unit value	Encoder input	Interpretation
0	0.000	−1.000	black channel value
64	0.251	−0.498	dark channel value
128	0.502	0.004	near the middle
255	1.000	1.000	maximum channel value

The small nonzero middle value comes from 128 being one step above the exact midpoint of the 8-bit range. This table is a direct conversion of sample byte values, not an encoder measurement. It also shows why storing the patch values as half precision is adequate for representing the input range: the source itself had only 256 possible values per channel. Later operations create richer floating-point activations.

The channel order matters just as much as the numeric range. A channel-first vector puts all 256 red values before the 256 green values and then blue. A pixel-interleaved RGB vector would alternate red, green, and blue for every pixel. Both contain the same byte count and produce the same apparent image in a generic viewer, but only one order matches the rows and columns in the saved patch matrix. The engine explicitly builds the channel-first layout expected by that tensor.

Can the activation arrays explain the image's memory peak?

We can estimate the base activation reservation for the 2,394-patch example from the dimensions in `vision_encode`. The count is padded from 2,394 to 2,400 so matrix batches align to 32 rows. A float array with 2,400 rows of width 768 uses about 7.03 MiB. The corresponding half array uses about 3.52 MiB. The feed-forward hidden array has width 3,072, so one half buffer for all padded patches takes about 14.06 MiB.

Allocation class	Count	Approximate bytes
Patch input, half, width 768	1	3.52 MiB
Residual/projection arrays, float, width 768	5	35.16 MiB
Normed and prepared head arrays, half, width 768	5	17.58 MiB
Feed-forward hidden, half, width 3,072	1	14.06 MiB
Pooled rows and projected output	1 each	3.02 MiB
Subtotal from these named buffers		**about 73.3 MiB**

The subtotal uses the host allocations visible in `VActs`, rounds the patch count to the actual multiple used by the code, and includes the final output rows. It is an estimate of allocated buffer sizes, not a measured process peak. It leaves out small per-task score arrays, allocator metadata, mapped weights, Metal's own bookkeeping, and any compiler spill allocation. That last item matters: the historical 206 MB spill could exceed the ordinary image activation arrays, even though it was created by a local attention-kernel register problem.

Some entries in the table are grouped for readability. The five float arrays of width 768 are the residual `x`, `q`, `k`, `v`, and attention/MLP projection scratch `a`. The five patch-count half arrays of that width are `xn`, `qh`, `kh`, `vh`, and attention output `o`. Pooled storage is rounded by output rows rather than patches, so the last row combines it with the final projected buffer. The exact allocation expression in the source is the authority if the structure changes.

This estimate helps explain why the attention spill reduction mattered even though the 559 MB `Q8_0` file is much larger than any one activation buffer. The mapped encoder weights are released when encoding is done, while the activation allocations exist during the call and the projected rows can remain with conversation history. Peak resident memory depends on which of these lifetimes overlap. Chapter 19's encoder mapping discussion will describe that boundary in more detail.

PREPRINT

What does a useful shape table reveal?

The W0 calculator can use the same input dimensions in either orientation, as well as a full-resolution photograph. The table is a geometry experiment only. Each example fits the default 280-row maximum; it helps show that the source's megapixel count is a poor proxy for the prompt's row count once smart resize applies.

Source dimensions	Prepared dimensions	Patch grid	Rows
1,024 by 768	912 by 672	57 by 42	266
768 by 1,024	672 by 912	42 by 57	266
4,000 by 3,000	912 by 672	57 by 42	266
960 by 720	912 by 672	57 by 42	266

These captured W0 outputs are in `code/expected/ch17_w0.txt`. The portrait case transposes the prepared grid while retaining the same number of groups. The other three inputs land on the same prepared dimensions because they share a 4-to-3 aspect ratio and exceed the same default maximum. The aligned 960-by-720 candidate is still larger than the allowed pixel area, so it is reduced. That last result is a reminder that already aligned sides do not bypass the budget check.

The 4,000-by-3,000 case has about twelve million source pixels but creates the same number of vision positions as the smaller inputs after the default resize. This is why the README's 12-megapixel example can be summarized with 266 rows. If the user raises the image-token ceiling, the prepared grid can grow, and both the encode cost and language-model prefill change. The W0 table cannot predict latency because it does not run a resampler, a matrix kernel, or an encoder.

How can source reading catch a wrong dimensional assumption?

The input patch vector width is 768, and the patch matrix returns another 768 values. That symmetry can tempt a reader to assume the patch operation is just a reshape or an identity. It is a learned projection: every output channel can combine all 768 input color samples. The row count remains P while width remains E. These are different axes in the activation arrays, and confusing them can make a buffer seem far smaller than it really is.

The head dimension is 64 because the 768-wide stream is divided among 12 heads. The prep kernel's output index is derived from the patch, kind, and head. One thread handles one such tuple. Its offset first chooses the patch's 768-wide row and then the head's 64-value slice. A source audit can check that the same offset is used for q, k, and v and that the selected norm vector has 64 values. A transposed or wrong-stride address can remain within the allocation and still mix heads silently.

Likewise, the position lookup uses the patch column as a modulo by the grid width and the patch row as an integer division by that width. Those expressions appear in both CPU and Metal code. Matching them is more important than matching variable names. The simplest review is to select three coordinates: top-left, one to the right, and one below; then verify that the additive table and rotary component change on the expected axis for each.

The output projection changes width from 768 to 2,560. That output width comes from the language model side and is why the final matrix is not another vision block. The rows are now shaped exactly like the ordinary language-model residual stream. The input path can therefore place them at positions where text-token embedding lookup would otherwise have produced vectors of the same width.

What has this chapter established?

We have followed the representation through four distinct resolutions. The source image is an RGB raster. The aligned canvas becomes a grid of 16-pixel patches. The vision transformer keeps one 768-value state per patch through 16 full-attention blocks. Non-overlapping groups of nine become a shorter sequence, and the final projection expands its width to 2,560 so it fits the E4B prompt.

The important boundaries are testable. W0 can check the geometry without weights. W1 can confirm the architecture metadata without reading tensor payloads. The loader's tensor checks can reject the

PREPRINT

wrong projector or dimensions. A W3 row dump can compare the encoder itself. The final chat answer is a useful end-to-end check, but it cannot substitute for inspecting the intermediate representation when the goal is to locate a mismatch.

The next chapter follows sound through its own front end, convolution layers, and Conformer blocks. It shares the final idea from this chapter: an encoder produces rows at the language-model width. The features and temporal structure are different, but once those rows join a prompt, the chat layer works with the same kind of media entry and cache lifetime described in Chapter 16.

The real code

The walkthrough will follow the same order as the data: choose a size, resample, make patches, process a block, then pool and project. The CPU reference is shown before the Metal companion. Source ranges in this section are checked against the frozen snapshot.

The first function is small enough to show the whole size decision. `align` is the patch side multiplied by the pooling width, so the output canvas fits complete 3-by-3 groups. The initial rounding avoids unnecessary resizes when an image is already close to an aligned size. The area checks then scale only when the candidate misses the minimum or maximum pixel budget. This is a CPU-only choice; no image tensor has been allocated yet.

```
2400 // The size to resize to ("smart resize"): the aspect ratio kept, both
       sides multiples of `align` (a patch ×
2401 // the pooling), the pixel count between min_px and max_px.
2402 static void smart_size(int w, int h, int align, int min_px, int max_px,
       int *tw, int *th) {
2403     int wb = (int)roundf((float)w / align) * align, hb = (int)roundf((
       float)h / align) * align;
2404     if (wb < align) wb = align;
2405     if (hb < align) hb = align;
2406     if (wb * hb > max_px) {
2407         const float beta = sqrtf((float)h * w / max_px);
2408         hb = (int)floorf(h / beta / align) * align, wb = (int)floorf(w /
       beta / align) * align;
2409         if (hb < align) hb = align;
2410         if (wb < align) wb = align;
2411     } else if (wb * hb < min_px) {
2412         const float beta = sqrtf((float)min_px / ((float)h * w));
2413         hb = (int)ceilf(h * beta / align) * align, wb = (int)ceilf(w *
       beta / align) * align;
2414     }
2415     *tw = wb, *th = hb;
```

Listing 17.1: `gemma4.c`:2400–2415

The output dimensions are rounded to aligned values, but preserving the aspect ratio can leave a narrow gap. `image_prepare` chooses a uniform scale, resamples the inner image, and copies it into a zero-filled RGB destination at a centered offset. The source comment says this implementation follows the image preprocessing used by the reference path. It avoids delegating a model-visible operation to a library whose default filter or rounding might change pixels.

```
2492 static Image image_prepare(const Vision *V, const Image *img) {
2493     const int align = V->patch * V->merge, area = align * align;
2494     int tw, th;
2495     smart_size(img->w, img->h, align, V->min_tokens * area, V->
       max_tokens * area, &tw, &th);
2496     Image out = {tw, th, xcalloc((size_t)tw * th * 3, 1)};
2497     if (tw == img->w && th == img->h) {
2498         memcpy(out.rgb, img->rgb, (size_t)tw * th * 3);
```

```
2499         return out;
2500     }
2501     const float scale = fminf((float)tw / img->w, (float)th / img->h);
2502     int nw = (int)ceilf(img->w * scale), nh = (int)ceilf(img->h * scale)
         ;
2503     if (nw > tw) nw = tw;
2504     if (nh > th) nh = th;
2505     uint8_t *p = img->rgb, *t = NULL;
2506     if (nw != img->w) p = t = resample(p, img->w, img->h, nw, 1);
2507     if (nh != img->h) {
2508         uint8_t *u = resample(p, nw, img->h, nh, 0);
2509         free(t);
2510         p = t = u;
2511     }
2512     const int ox = (tw - nw) / 2, oy = (th - nh) / 2;
2513     for (int y = 0; y < nh; y++) memcpy(out.rgb + ((size_t)(y + oy) * tw
         + ox) * 3, p + (size_t)y * nw * 3, (size_t)nw * 3);
2514     free(t);
2515     return out;
2516 }
```

Listing 17.2: gemma4.c:2492–2516

This row loop is the core of one separable resampling pass. The precomputed table gives each output coordinate its first source sample, the number of contributing samples, and fixed-point weights. The same loop handles horizontal and vertical passes by selecting which coordinate comes from the output row and which comes from the inner pixel loop. The round bit is added before the fixed-point shift; then the result is clamped to an 8-bit channel.

```
2460 static void resample_row(void *ctx, int y, int tid) {  // output row y
2461     (void)tid;
2462     const ResampleJob *j = ctx;
2463     for (int x = 0; x < j->ow; x++) {
2464         const int o = j->horizontal ? x : y, first = j->bounds[2 * o],
         cnt = j->bounds[2 * o + 1];
2465         const int32_t *k = j->wt + (size_t)o * j->ksize;
2466         for (int c = 0; c < 3; c++) {
2467             int32_t acc = 1 << (RS_BITS - 1);
2468             for (int i = 0; i < cnt; i++) {
2469                 const int sx = j->horizontal ? first + i : x, sy = j->
         horizontal ? y : first + i;
2470                 acc += j->src[((size_t)sy * j->in_w + sx) * 3 + c] * k[i
         ];
2471             }
2472             j->dst[((size_t)y * j->ow + x) * 3 + c] = clip8(acc >>
         RS_BITS);
2473         }
2474     }
2475 }
```

Listing 17.3: gemma4.c:2460–2475

Once the RGB canvas is ready, `vision_encode` packs each patch as channel, row, column, converts normalized values into the model's input range, and multiplies by the patch matrix. Position tables are added before the first block. The CPU helper below shows exactly how a patch index becomes a column and row and how each head is prepared. This is the CPU reference path; the Metal kernel later mirrors the same split.

```
2543 // patch p: per head, q and k RMS-normed (with weights) and turned (2-D
         RoPE), v RMS-normed
```

```
2544 static void vit_prep_row(void *ctx, int p, int tid) {
2545     (void)tid;
2546     const VActs *a = ctx;
2547     const Vision *V = a->V;
2548     const int E = V->n_embd, hd = V->hd, col = p % a->cols, row = p / a
     ->cols;
2549     float x[128];
2550     for (int h = 0; h < V->n_head; h++) {
2551         const size_t o = (size_t)p * E + h * hd;
2552         rmsnorm(x, a->q + o, a->L->q_norm, hd, V->eps);
2553         rope(x, col, hd / 2, V->rope_base, NULL), rope(x + hd / 2, row,
     hd / 2, V->rope_base, NULL);
2554         for (int i = 0; i < hd; i++) a->qh[o + i] = float_to_half(x[i]);
2555         rmsnorm(x, a->k + o, a->L->k_norm, hd, V->eps);
2556         rope(x, col, hd / 2, V->rope_base, NULL), rope(x + hd / 2, row,
     hd / 2, V->rope_base, NULL);
2557         for (int i = 0; i < hd; i++) a->kh[o + i] = float_to_half(x[i]);
2558         rmsnorm(x, a->v + o, NULL, hd, V->eps);
2559         for (int i = 0; i < hd; i++) a->vh[o + i] = float_to_half(x[i]);
2560     }
```

Listing 17.4: gemma4.c:2543–2560

For a head, the CPU path normalizes q and k with their learned per-head weights, then calls the existing rotary helper twice: once for the column half and once for the row half. It normalizes v without a weight. The three prepared arrays are stored as half values, which is the form consumed by the attention implementation. `hd / 2` is the size assigned to each axis; within each half the RoPE helper uses the usual paired coordinates.

The surrounding loop is longer than a useful listing, so this excerpt starts after image preparation and shows a complete per-layer handoff. It is the CPU/GPU bridge rather than an independent kernel: calls use `STEP`, projections use the clippable linear wrapper, and the vision attention call asks for all P keys. The residual and normalization helpers keep the operation ordering visible.

```
2631     matmul(a.x, a.patches, &V->patch_w, a.P, 0);
2632     STEP(metal_vit_pos(a.x, V->pos_x, V->pos_y, a.P, E, a.cols), a.P,
     vit_pos_row, &a);
2633     rmsnorm_rows(a.xn, a.x, V->layers[0].ln1, a.P, E, V->eps, 0);  // (
     later layers' come from the previous one)
2634     for (int l = 0; l < V->n_layer; l++) {
2635         const VLayer *L = a.L = &V->layers[l];
2636         clamp_in(&L->wq, a.xn, a.P);  // (k's and v's input clamps are q
     's)
2637         gpu_group(1);
2638         matmul_out(a.q, a.xn, &L->wq, a.P);
2639         matmul_out(a.k, a.xn, &L->wk, a.P);
2640         matmul_out(a.v, a.xn, &L->wv, a.P);
2641         gpu_group(0);
2642         STEP(metal_vit_prep(a.qh, a.kh, a.vh, a.q, a.k, a.v, L->q_norm,
     L->k_norm, a.P, V->n_head, V->hd, a.cols,
2643                             V->rope_base, V->eps),
2644              a.P, vit_prep_row, &a);
2645         STEP(metal_attention(a.o, a.qh, a.kh, a.vh, NULL, NULL, a.P, 0,
     V->n_head, V->n_head, V->hd, 0, 1, a.P),
2646              V->n_head * ((a.P + ATT_QB - 1) / ATT_QB), vit_attn_task, &
     a);
2647         linear(a.a, a.o, &L->wo, a.P);
2648         add_norm(a.x, a.a, L->attn_post_norm, a.P, E, V->eps, 1.0f, a.xn
     , L->ln2);
```

```
2649          linear_glu(a.h, a.xn, &L->gate, &L->up, a.P, V->gelu_quick);
```

Listing 17.5: gemma4.c:2631–2649

Reading this excerpt from top to bottom gives the layer boundary. The patch matrix is followed by additive positions, then the first pre-norm. The q input clamp is applied once because the loader has already checked that q, k, and v share the interval. A GPU group surrounds the three output-clamped matrix multiplies. Preparation and full attention can fall back to the CPU implementations. The attention output projection is followed by its post-norm residual update; the gate/up pair is followed by the down projection and the other residual update in the next source lines.

The corresponding Metal excerpt shows how one thread maps to a patch, head, and one of q/k/v. The first two output branches use learned norm weights and two rotations; the value branch takes the normalized path and returns early. The kernel uses the same column and row coordinates derived from the row-major patch index. In contrast with the CPU loop over all heads for a patch, the grid exposes each patch/head/kind combination to the GPU.

```
1487 // one thread per (patch, q | k | v, head): RMS norm (with weights for q
     , k), then for q and k 2-D NeoX RoPE:
1488 // dims [0, hd/2) turn with the column, [hd/2, hd) with the row, pairs (
     i, i + hd/4) within each half
1489 kernel void vit_prep(device half *q [[buffer(0)]], device half *k [[
     buffer(1)]], device half *v [[buffer(2)]],
1490                       device const float *q_new [[buffer(3)]], device
     const float *k_new [[buffer(4)]],
1491                       device const float *v_new [[buffer(5)]], device
     const float *q_norm [[buffer(6)]],
1492                       device const float *k_norm [[buffer(7)]], constant
     VitParams &p [[buffer(8)]],
1493                       uint gid [[thread_position_in_grid]]) {
1494     const int H = p.n_head, h = gid % H, kind = gid / H % 3, pt = gid /
     (3 * H);
1495     if (pt >= p.P) return;
1496     const ulong o = (ulong)pt * p.n + h * p.hd;
1497     device const float *src = (kind == 0 ? q_new : kind == 1 ? k_new :
     v_new) + o;
1498     device half *dst = (kind == 0 ? q : kind == 1 ? k : v) + o;
1499     float ss = 0;
1500     for (int i = 0; i < p.hd; i++) ss += src[i] * src[i];
1501     const float scale = rsqrt(ss / p.hd + p.eps);
1502     if (kind == 2) {
1503         for (int i = 0; i < p.hd; i++) dst[i] = half(src[i] * scale);
1504         return;
1505     }
1506     device const float *w = kind == 0 ? q_norm : k_norm;
1507     const int half_d = p.hd / 2, quarter = p.hd / 4;
1508     for (int part = 0; part < 2; part++) {
1509         const int pos = part == 0 ? pt % p.cols : pt / p.cols, b = part
     * half_d;
1510         for (int i = 0; i < quarter; i++) {
1511             const float theta = pos * precise::pow(p.rope_base, -2.0f *
     i / half_d);
1512             const float c = precise::cos(theta), sn = precise::sin(theta
     );
1513             const float x0 = src[b + i] * scale * w[b + i], x1 = src[b +
      i + quarter] * scale * w[b + i + quarter];
1514             dst[b + i] = half(x0 * c - x1 * sn);
1515             dst[b + i + quarter] = half(x0 * sn + x1 * c);
1516         }
```

```
        }
    }
```

Listing 17.6: `gemma4_metal.m`:1487–1518

What do the wrappers do with clamp ranges?

The source makes a useful distinction between loading a linear operation and executing it. Loading records four optional scalar bounds next to the matrix tensor. Missing bounds use open-ended sentinels, so ordinary matrices pass through the same helper. During execution, input clamping is a separate operation; output clamping is handled by the matmul store path. This arrangement avoids changing the underlying generic matmul kernel for every encoder projection.

The abbreviated loader excerpt is intentionally not a sixth listing. Its source is `gemma4.c` lines 2129–2151: `Linear` stores the matrix and its four bounds, `load_linear` looks up the scalar tensors, and `same_input_clamp` lets `load_vision` reject an inconsistent q/k/v or gate/up file. The reader can follow those lines directly in the snapshot. The check is a data-integrity guard: if the shared input were clamped differently for each of q, k, and v, one shared input buffer could not represent all three operations.

What can the Metal pool kernel teach us?

The pooling kernel is a helpful contrast even though its body is not repeated as a listing. One threadgroup owns one output row. Its grid coordinate is converted to the output column and row, and each thread handles a stride through the 768 channels. For every channel it accumulates the nine patch values from the corresponding 3-by-3 neighborhood. It then applies the model's optional bias and scale, accumulates the squared values for RMS normalization, reduces those partial sums across the threadgroup, and writes the normalized half row.

The kernel declares a 4,096-float shared array, enough for the supported channel width, and a small scratch array for the reduction. Each channel value is written once before the reduction result is used for normalization. The output is half precision because the next step is a batched projection, and the row crosses into the language-model input staging path.

The CPU twin is `vit_pool_row` in `gemma4.c`, lines 2587–2604. The Metal version is `vit_pool` in `gemma4_metal.m`, lines 1520–1539. Compare their 3-by-3 loops and the shared scale: both average a patch group and then apply the RMS normalization. The GPU layout assigns a threadgroup to each pooled position, rather than a group to each patch.

Operation	Reference implementation	GPU counterpart
Size and pixels	`smart_size`, `resample`, `image_prepare` on CPU	none; image stays ordinary RGB input
Patch projection	`matmul` and CPU row preparation	shared matrix routines and `vit_prep`
Vision blocks	`vit_prep_row`, `vit_attn_task`, `vit_pool_row`	`vit_prep`, attention, `vit_pool`
Final projection	`matmul` on pooled rows	shared batched matrix routine

This table also prevents a common reading error: not every operation in an image encode is a Metal kernel. File decoding, resize, border filling, and patch packing happen on the CPU. The engine dispatches the large projections and supported kernels to Metal, with CPU implementations retained for the `STEP` operations. The final matrix helper is shared with the rest of the engine.

PREPRINT

Experiment: inspect shapes without running the encoder

Experiment W0

Run the weight-free shape calculator:

```
nice -n 10 python3 code/examples/ch17_shape.py 1024 768
```

It copies the size policy from `smart_size` and reports the prepared dimensions, patch grid, and pooled row count. It uses only four integer inputs and the standard library; it opens no model or image. On this machine, the captured result is in `code/expected/ch17_w0.txt`: the prepared size is 912 by 672, the patch grid is 57 by 42 (2,394 patches), and pooling yields 266 output rows. The capture was made 2026-10-22 on a Mac15,12; the idle-GPU gate is not applicable because no GPU operation runs. This is a geometry check, not a resize-pixel comparison.
For W1, set `MM` to the path of a local `Q8_0` or F16 encoder GGUF and run:

```
nice -n 10 python3 code/examples/ch17_vision_header.py "$MM"
```

The small Python reader uses only the standard library. It reads the GGUF header and typed metadata, then stops before tensor descriptors and payload. The captured result for the local `Q8_0` file is in `code/expected/ch17_w1.txt`: it reports `clip` / `gemma4v`, a vision encoder, 16 blocks, width 768, 12 heads, and patch size 16. The header also says the file contains 1,411 tensors; that count comes from the GGUF header, not a scan of their data. This is a metadata observation, not a test that the file's tensors are valid or that the encoder output is correct. It was captured on the same date and machine; the GPU idle gate is not applicable.
The outlined image experiment is W3. Try it on your machine with this command:

```
nice -n 10 code/build/gemma4 -m "$MODEL" --mmproj "$MM" \
  --image photo.jpg -p "What is in this picture?"
```

Repeat with `--image-tokens 70` and `--image-tokens 1120` using a page with small print, and save the image rows with `GEMMA4_DUMP_EMBD=...`. **Try it on your machine (W3):** inspect image row count, encode time, prefill time, and saved rows for both resolutions. The image-token setting predicts more rows at 1,120 than at 70. Any README values are historical reference points, not fresh results.

Pitfall

Pitfall

Symptom. The vision rows have mean cosine similarity 0.992 against llama.cpp.
Cause. The model selects tanh GELU, but the converter drops that setting and llama.cpp uses quick GELU.
Fix. For this compatibility comparison, set `GEMMA4_VISION_GELU_QUICK=1`. The README reports per-row cosine at least 0.9997 and 0.4% relative error. Source: snapshot `README.md`, "Images, audio and video" / "Accuracy".

The switch matches the alternate implementation; it does not change which activation the model configuration specifies.

The README also records a speed change in the same preprocessing path: a 12-megapixel resize fell from 113 ms to about 20 ms when the work was distributed across cores. The single-core comment beside `resample` is a reminder that a source-level loop's apparent cost is not necessarily the shipped path's wall time. These are historical README values, not measurements made for this chapter. Source:

snapshot `README.md`, “Speed”, M3 Air, warm encoder kernels.

Exercises

Try these without changing the live engine. The first three are shape and data-flow checks; the final two ask how to validate compatibility and memory behavior.

Exercises

1. For a 1,024 by 768 source image, use the W0 calculator to find the prepared size, patch grid, and output row count. Explain why the output is not simply 280.
2. A patch row is numbered in row-major order. For a grid 57 patches wide, find the row and column of patch index 100. Which coordinate controls each half of its 2-D rotary transform?
3. Each output row combines a 3-by-3 group. If an aligned patch grid is 21 wide by 15 high, how many patches enter the blocks and how many pooled rows leave them?
4. The clamp ranges are far smaller than the largest F16 value. What does that tell you about their role? Trace where an input clamp and an output clamp enter the projection path.
5. Design a W3 comparison of F16 and `Q8_0` image embeddings. State what file, image, activation setting, row count, and per-row statistic you would record. Explain why matching the final caption alone is a weaker check than comparing the encoder rows.

The matching hints follow Chapter 16 in Appendix F.

Rebuild checkpoint

The agreed rebuild sequence ends at m8, chat prefix reuse; Chapter 17 adds no invented m9. For a small image-side exercise, implement the aligned size calculation in `steps/` only if the reader wants an optional extension, then compare it against the W0 probe on several wide, tall, and already aligned dimensions. Keep pixel resampling and encoder numerics as separate follow-on work: a matching output size does not establish matching pixels or matching embeddings. The formal checkpoint remains m8 from Chapter 16.

PREPRINT

Chapter 18

Hearing: the audio encoder

The question

How does sound become 25 rows a second that the E4B language model can read?

Reading map: 22 pages

This first pass assigns the outline's 22-page estimate before the detailed walkthrough. The estimate includes the diagram, source excerpts, worked shape counts, experiment protocol, documented pitfall, exercises, and rebuild checkpoint.

Section	Pages
Question and audio-to-row map	1
Samples, framing, FFT, mel bands, and two convolutions	4
Conformer block: half-step FFNs, local attention, and causal convolution	6
Output rows, chunk boundaries, and the chat prompt	3
CPU/GPU work split and activation memory	2
Source walkthrough and Metal companion	2
W0/W1 experiments and W3 reader transcription experiment	1
Documented pitfalls and real accuracy context	1
Exercises and Appendix F hints	1
Rebuild checkpoint m5b	1
Total	**22**

The idea

What does the language model receive?

The audio encoder does not hand the chat loop a transcript. It hands it a sequence of numeric rows, in the same width the language model expects from its token embeddings. The model then reads those rows as positions in its prompt. Any transcription is the language model's reply to those positions, shaped by the prompt and the model's learned behavior.

The path begins with mono samples at 16,000 per second. It turns overlapping windows into log-mel feature rows, reduces time with two convolutions, processes the remaining rows with a 12-block Conformer, and projects each result to width 2,560 (via a 1,536-wide intermediate). The sequence length after the two stride-2 convolutions is one row for every 40 milliseconds of audio: 25 rows per second.

The row count depends on the chunk's sample count and the exact frame formula, especially for very short clips and the last partial chunk. For ordinary durations, the useful mental conversion is simple: ten seconds gives about 250 rows, 17.5 seconds about 438, and one full 30-second chunk 750. These are encoder rows; the user prompt also contains text and control markers.

PREPRINT

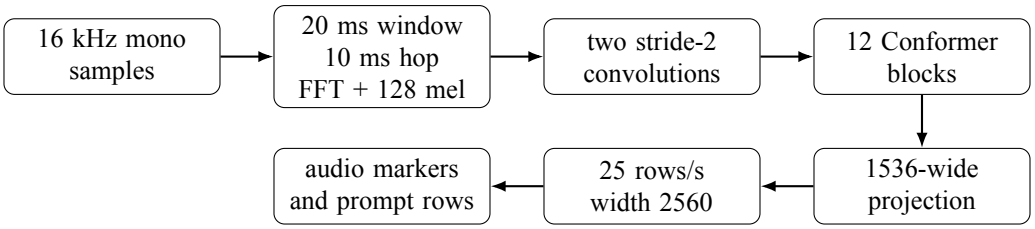


Figure 18.1: Audio becomes a lower-rate sequence of language-model-width rows.

Why frame the samples with overlap?

Speech changes continuously. A 20 ms window is short enough to capture local sound structure, while a hop of 10 ms means neighboring frames share half their samples. At 16 kHz, those durations are 320 samples per window and 160 samples per hop. The engine pads the beginning so the first analysis window is centered according to the chosen feature-extractor convention.

The window is periodic Hann. Its weights taper the beginning and end of the window, reducing the sharp cut that would otherwise spread energy across FFT bins. The 512-point transform has room for the 320 weighted samples plus zero padding. It produces 257 nonnegative-frequency magnitudes. Each mel band combines those magnitudes with a triangular filter; the implementation uses the HTK mel spacing, no area normalization, and a natural logarithm with a floor of 0.001. The floor gives a quiet or empty band a finite feature value rather than negative infinity.

Do not confuse the analysis frame rate with the final row rate. The feature extractor advances 100 times a second. The first convolution halves time, and the second halves it again, so the Conformer sees roughly 25 positions per second. The encoder is not dropping arbitrary words at this stage; each convolution has a learned 3-by-3 filter over time and mel band and combines neighboring features.

What exactly does the FFT do here?

The FFT is an in-place radix-2 transform over 512 complex values. Before the butterfly stages, the code permutes the values into bit-reversed order. Each stage doubles the butterfly span: first pairs, then groups of four, then eight, until the full transform is covered. The twiddle factors are read from cosine and sine tables prepared when the audio encoder is loaded.

For a small mental check, a constant signal has energy concentrated at frequency zero; a sinusoid aligned to one FFT bin concentrates energy near that bin. The implementation then takes the magnitude, not a power spectrum, before applying the mel weights. That distinction matters because replacing magnitude with squared magnitude changes every downstream feature value. The mel filters and their normalization are part of the model's input contract.

The 128 filterbank outputs describe coarse frequency regions rather than exact FFT bins. Neighboring triangular filters overlap, so one frequency can contribute to two adjacent bands. In the source, each filter is precomputed for bins from zero to the Nyquist frequency, then each frame performs a weighted sum. The log floor is applied after summation. This is a deterministic CPU front end and does not need encoder weights at inference time once the filter tables have been constructed.

Here is a small numerical way to think about the filter stage without pretending to know the model's trained weights. Suppose one mel triangle assigns weights 0.25, 0.50, and 0.25 to three adjacent FFT magnitudes, and those magnitudes are 2, 4, and 6. Its weighted magnitude is 4; the feature stored for that band is the natural log of 4. In the actual encoder, each of 128 bands has a longer precomputed weight row over 257 magnitudes, and the sum is floored at 0.001 before logging. The toy values explain the reduction only; they are not an output measurement from a model file.

At 16 kHz, the frequency spacing between adjacent FFT bins is 31.25 Hz. This is a property of the transform size and sample rate, not of the mel filters. The mel filters combine bins into perceptual bands, with more closely spaced centers at low frequencies and wider spacing toward the upper end. A voice's

PREPRINT

Draft book, October 2026. Not peer reviewed.

fundamental and harmonics therefore do not map one-to-one to mel bands. The convolution learns useful patterns from this coarser representation instead of receiving raw samples.

The feature array is frame-major: all 128 bands for frame zero, followed by all 128 bands for frame one, and so on. Each output frame is computed independently, so the engine parallelizes that outer frame loop. The temporary real, imaginary, and magnitude arrays belong to one worker's frame. This makes the feature stage's temporary memory proportional to the number of workers times the small FFT buffers, while the saved mel array grows with the clip length.

How much do the convolutions shorten the sequence?

Each convolution uses a 3-by-3 receptive field and stride two with padding one. The output length along an axis is the ceiling of half the input length. Time and frequency both shrink, while the channel count grows. The first convolution begins with one feature channel and produces a learned number of channels; the second consumes those channels and produces 32. Each result is normalized across its channels, passed through ReLU, and stored as half precision before the next stage.

Take a 17.5-second clip represented by exactly 280,000 samples. The source frame rule yields 1,749 frames; the two stride-2 stages leave 438 rows. The historical performance table in ENGINE-MATH reports 437 rows for its 17.5-second sample. That one-row difference is a reminder that the exact sample count and chunk alignment matter; a duration rounded to one decimal place is not enough to predict the row count.

After the second convolution, the remaining frequency positions and channels are rearranged into the input width expected by the learned projection. This is a reshape and ordering contract as much as it is a matrix multiply: changing whether channel or band varies fastest would feed the weights a different vector. The loader checks convolution shapes and tensor types before encoding, and the row builder follows the same band-major layout used by the input projection.

What makes a Conformer different from an ordinary transformer?

Each block combines attention with a local convolution module. The two feed-forward networks sit on either side of attention and convolution, and each contributes half of its normalized branch before the residual stream is normalized for the next block. The half-step scaling is part of the block definition, not a tuning constant to omit when porting the model.

In outline, one block does this: half an FFN branch; local attention; a causal depthwise convolution with a gated linear unit; another half FFN branch; then output normalization. The attention sees at most the current frame and its 11 predecessors. A learned relative-position term is added to the query/key score, and the score is soft-capped at 50. The convolution has a five-frame kernel and is causal: its taps reach backward through the current frame and never peek at a future frame.

The two mechanisms have different reach. Attention can select among the latest 12 frames, with a learned distance signal. The convolution mixes nearby channel values over five positions after a GLU. Since both are causal, a position's representation does not use later audio positions in the same chunk. This local design bounds attention work and gives the model a short-range signal path that does not depend on attention weights.

Block step	Input and operation	Residual effect
FFN 1	normalized row; up, SiLU, down	add half of the post-normalized branch
Attention	q/k/v projections; 12-position causal window and relative term	add normalized attention output
Convolution	pointwise GLU; five-tap causal depthwise convolution; SiLU	add projected convolution branch
FFN 2	normalized row; up, SiLU, down	add half of the post-normalized branch
Block tail	RMS normalization with learned weights	normalize the updated stream

PREPRINT

The operations are described in the source's sequence rather than compressed into one generic transformer block. The Conformer uses per-dimension q and k scales, a learned relative key for each distance, and a soft cap. Its attention is not interchangeable with the text model's GQA attention or the vision encoder's full patch attention. Read the audio-specific task alongside its Metal twin when following the score.

How does sound reach the E4B prompt?

After the Conformer blocks, a projection first maps each row to an intermediate width of 1,536. An unweighted RMS normalization follows, and a final projection maps to the language model's width of 2,560. The returned buffer contains one 2,560-value row per encoded position. The chat layer wraps those rows between the audio markers and places them in the rendered prompt just as it does image rows.

This boundary explains why the encoder does not need to know about tokens, vocabulary IDs, or text decoding. It produces representations, not words. The 42-layer language model processes those rows and can answer with a transcription, a summary, or another response requested by the user. The same encoder rows can support different text replies when the instruction changes.

For ten seconds of aligned audio, budget about 250 media positions. At 2,560 values per row, that is 640,000 scalar values before the row storage format is considered. The prompt markers add a small number of entries, while the prefill computation must process all media rows. This is why a brief voice note can still be a substantial prompt even though its visible transcript might contain only a few dozen words.

Why split audio at 30 seconds?

The engine processes each chunk independently, up to 30 seconds, then appends the resulting rows. A full chunk has 480,000 samples and about 750 output positions. A 70-second file becomes three encoder calls: 30, 30, and 10 seconds, with about 1,750 rows overall. The last chunk can be shorter; if it is too short to make even one frame, the chunk helper returns no rows.

Separate encoding limits the activation arrays for one call and matches the behavior documented by the project. It also means convolution and attention context restart at the chunk boundary. The final frame of one chunk does not attend to the first frame of the next. This is a real boundary in the computation, even though the rows are concatenated into one prompt sequence afterward.

Audio	Samples at 16 kHz	Chunks	Rows, approx.	What to remember
10 s	160,000	1	250	approximate count; framing sets the exact count
17.5 s	280,000	1	437–438	source's alignment can shift a rounded estimate
30 s	480,000	1	750	maximum ordinary chunk
70 s	1,120,000	3	1,750	attention context restarts at each boundary

Which stages use the CPU and which can use Metal?

The CPU reads and converts the audio to 16 kHz mono, calculates log-mel features, computes the first convolution row by row, and performs the convolution LayerNorm and ReLU steps. The second convolution is batched through the shared matrix helper, then the Conformer projections and supported kernels can use Metal. If a Metal helper is unavailable, the C reference path runs the corresponding operation. This is the same pattern used elsewhere in the engine: the GPU path accelerates the main tensor work while the CPU implementation remains a readable reference and fallback.

The first convolution's row-at-a-time design has an explicit memory payoff. A 30-second feature sequence has about 3,000 frames and 128 bands, but the stride-two first convolution emits about half as many time positions and half as many bands. A full F32 output plane therefore depends on both reduced dimensions and the model's declared output-channel count. The project README documents a 49 MB

PREPRINT

intermediate avoided by this rowwise path. The implementation computes one output row, normalizes it, and immediately stores the result in F16. The important invariant is that the row is complete before it is converted; it need not be retained as a full F32 tensor.

The Metal attention kernel assigns a SIMD group to a frame and head, which makes its short 12-key window local to one task. The Metal convolution kernel handles the GLU, causal depthwise convolution, normalization, and activation. The host still controls dispatch order and synchronizes where the CPU needs results. A command buffer may queue GPU work, but the CPU cannot safely consume its outputs before synchronization.

What does the activation layout tell us about memory?

The activation structure names the intermediate arrays: mel features, the first convolution's half output, the second convolution's input columns and result, residual rows, normalized half rows, q/k/v, attention output, pointwise hidden values, GLU output, and final embeddings. Reading the declarations before the block loop helps distinguish persistent weights from per-chunk working memory.

For a 30-second chunk, the sequence after subsampling is roughly 750 rows. A single F32 array of shape 750 by 1,024 occupies 3,072,000 bytes, about 2.93 MiB. Three such q/k/v arrays use about 8.8 MiB before padded rows and other scratch. The activation peak is larger because several arrays coexist, and the two convolution stages also have their own temporaries. These are shape calculations from the declared data types, not a measured process peak.

Weights have a different lifetime. The encoder tensors are mapped for GPU use when the media is encoded, then unmapped; ordinary text chat need not keep the encoder permanently wired. The chapter's memory arithmetic therefore describes the chunk's scratch, not the process's whole footprint or the file's resident pages. Chapter 19 follows the wider media lifetime and memory budget.

Which choices define the feature contract?

Several details look replaceable but alter the values presented to learned weights: mono mixing, resampling, sample rate, beginning padding, periodic Hann window, magnitude rather than power, 512-point FFT, HTK mel spacing, no area normalization, natural log, the 0.001 floor, and the frame alignment. A change can preserve the same row count while changing every feature value.

The same caution applies after the features. Convolution padding determines edge rows; LayerNorm operates over output channels and has no bias; ReLU is applied before F16 storage. The depthwise convolution is causal and uses a five-frame kernel. Relative-position terms depend on distance, and the score cap changes extreme values. A shape match alone does not establish numerical equivalence.

What is a half-step feed-forward branch?

The block has two feed-forward branches, one before attention and one after the convolution module. Each branch first normalizes the current residual row, projects it to the feed-forward width, applies SiLU, projects it back to the encoder width, and applies a learned post-normalization scale. Its contribution is halved before it is added. This lets each block use two feed-forward updates without giving their combined residual effect the same size as two full branches.

The source folds the half factor into the post-normalization weights while loading them. That detail matters when following the runtime loop: the loop can call its usual normalized residual-add helper and does not need a separate multiply-by-one-half at each row. When reading a tensor dump or writing a reference implementation, however, it is easy to apply the factor twice. The correct question is whether the weights already include it at that point in the data path.

The two FFNs are separately parameterized. The first runs before the attention and convolution pair; the second runs after it. In the C loop, the attention-and-convolution section occurs only during the second sub-iteration, between the two FFNs. The loop shape can look surprising until the order is written out:

PREPRINT

FFN 1, attention, convolution, FFN 2, block output normalization. A generic encoder that puts one FFN after attention would not match this structure.

Each FFN's SiLU is evaluated in the Metal helper for supported shapes. The CPU fallback visits every element and stores the activation as half precision. The down projection then reads that hidden buffer. The half conversion is a deliberate boundary between a wide F32 accumulator and the next matrix multiply; it should be preserved in a comparison that aims to match engine numerics rather than just mathematical intent.

How is the relative-position score assembled?

The loader builds a table of sinusoidal vectors for the 12 possible local distances, then projects them with the learned relative-key matrix. During attention, the current frame's query is scaled per dimension. The key is separately scaled per dimension. Each distance score is the key dot product plus a dot product between the scaled query and that distance's learned vector. Only after both terms are combined does the implementation apply the soft cap.

The cap is a smooth saturation, not a hard clamp. Scores far beyond the cap approach a bounded magnitude rather than being abruptly cut to a constant. The softmax then subtracts the largest capped score, exponentiates differences, and normalizes across the visible window. At frame zero there is one score; at frame five there are six; after the window fills there are twelve. Early frames therefore have shorter attention work and a different normalization count.

The Metal kernel distributes one frame and one head to each SIMD group. Its lanes divide the head dimensions into small slices, reduce the partial dot products, and cooperate on the score and value passes. The output is written as half precision. The CPU reference is organized by one task for the same frame/head pair and performs its dot products directly. Different reduction orders can produce tiny floating-point differences, which is why a row comparison should use a stated tolerance rather than assume bit identity without checking.

The key and value arrays are still full sequences for the current chunk, even though each query reads at most 12 positions. This makes simple row addressing possible and lets all projected keys and values be available before the attention tasks run. Local attention bounds the score work; it does not mean the engine stores only 12 rows of activations. Text attention's persistent KV cache is a different object with a different lifetime and layout.

What does the causal convolution add?

The convolution module begins with a pointwise projection twice as wide as the encoder stream. The first half is the candidate signal and the second half controls a sigmoid gate. Multiplying the candidate by that gate gives a stream with width equal to the residual. Each channel then has its own five learned tap weights. At time t, the taps cover t-4 through t; negative positions at the beginning contribute zero.

After the depthwise sum, the engine computes an RMS scale across the channels for that row, multiplies by the learned post-normalization weights, applies SiLU, and stores a half-precision hidden row. A second pointwise projection returns this result to the residual stream. The convolution is depthwise because each output channel uses its own taps and does not mix channels with neighboring channels; cross-channel mixing happened in the pointwise projections around it.

Consider the very first frame. Only the last tap (the one aligned with the current frame) can contribute because all earlier frame indices are negative. At frame three, four of the five taps have real inputs; the earliest tap still reaches before the start and contributes zero. By frame four, the full five-frame history is available. This boundary behavior is not zero-padding the entire sequence after the end; the model is causal and does not calculate future outputs from padded future frames.

The Metal kernel uses one threadgroup per output time row and spreads channel indices among its threads. It accumulates each channel's tap sum, keeps those sums in threadgroup memory, reduces their squares to get the row scale, and writes activated half outputs. The source declares scratch sized for up

PREPRINT

to 4,096 channels. The C fallback uses a row loop and an explicit local accumulator. Both paths read the same causal tap order, but their parallel reduction details differ.

What exactly shrinks in a two-dimensional convolution?

The feature tensor has a time axis, a mel-band axis, and a channel axis. The first convolution sees one channel and uses three time positions by three neighboring bands to create many channels. The second convolution sees those channels, again gathers a 3-by-3 area, and emits 32 channels. Both time and mel-band counts shrink with stride two; channel count changes according to the learned tensor shape.

The source flattens each receptive field into a column whose order is channel, time offset, band offset. For the first convolution, channel has only one value, so each output channel consumes nine weights. For the second, every old channel contributes its own nine-value patch, and the flattened width is nine times the input-channel count. The matrix multiply treats each output location as a row and each flattened patch as its input vector.

At the edge of either axis, a receptive-field coordinate outside the valid range is zero. This applies both at the first and last feature frames and at the low/high mel boundaries. The output length uses the convolution's padding convention, which keeps a ceiling-half count for odd or even input lengths. Replacing that with floor division can lose a row whenever the preceding length is odd, and the later output count will then also be wrong.

The first convolution is special in where its result is stored. It computes one row of output channels in F32, applies channel LayerNorm, then immediately applies ReLU and converts to F16. The second convolution needs columns from those half rows, so `im2_row` unpacks half values while assembling its matrix input. This sequence trades a small conversion for a much smaller retained activation tensor.

Can we audit the row count with a concrete sample?

The chunk helper derives the number of feature frames directly from the integer sample count. With 160,000 samples, that expression gives 999 frames. The first stride-two stage has 500 time positions; the next has 250. At the standard output width of 2,560, that clip therefore contributes 640,000 scalar values to the media-row buffer. This is a worked source calculation, not an encoder output dump.

For 280,000 samples, the source produces 1,749 frames. The two convolution lengths are 875 and 438. With 1,536 values in the intermediate projection, that stage contains 672,768 scalar values. The final projection changes the width to 2,560, but the row count stays 438. Keeping width and sequence length separate prevents a common mistake: projection changes how much data each row contains, not how many audio positions exist.

A full 480,000-sample chunk gives 2,999 feature frames and 750 output rows. A 30-second chunk at 2,560 values per row carries 1,920,000 scalar values in the final row buffer. At F32, those values take 7,680,000 bytes. The row is a sequence position for prefill; its storage size and its compute cost are related but not identical measures.

The code rounds the working row dimension up to a multiple of 32 for some buffers. That padded capacity helps batched kernels operate on convenient tile sizes, but only the actual T rows are valid output positions. The embedding allocation uses T rather than the padded count. When inspecting a memory dump, do not count padded rows as prompt entries.

What limits a long recording's working set?

Chunking bounds the number of rows in the current Conformer call. It also bounds the feature tensor, q/k/v arrays, and most layer scratch. The returned embeddings from earlier chunks are retained because they will be appended to the media sequence, but the temporary arrays for a completed chunk are freed before the next chunk returns. In the current helper, the growing output buffer is copied into a newly allocated larger buffer as chunks are appended; this is simple and makes ownership clear, though it copies previous rows on each extension.

PREPRINT

For a 70-second recording, chunk outputs total about 1,750 rows. At width 2,560 and F32 output, that is about 17.9 MB of embedding values. The returned buffer is shared GPU-visible memory so the following matrix and prompt staging paths can consume it. This arithmetic excludes allocator overhead and any copy while the total buffer grows. It also excludes the audio file, sample buffer, language-model KV cache, and encoder weights.

The feature tensor for one full chunk is about 3,000 frames by 128 bands, or 384,000 F32 values. That is about 1.46 MiB. The first convolution output's half buffer is shape-dependent; the README's earlier optimization note names a 49 MB F32 intermediate avoided by rowwise output. These figures describe different stages and different tensor formats, so they should not be added into a claimed peak without proving their lifetimes overlap.

The allocation sites make those lifetimes inspectable. Arrays owned by the `AActs` value are released at the end of `audio_encode_chunk`, after the final projection and synchronization. The returned embeddings are not in that scratch list; ownership passes to `audio_encode`, which may append them to the aggregate output. This distinction matters when following a memory report: a freed activation buffer and a retained prompt embedding have different reasons to remain resident.

What do the loader checks promise?

Before an audio path can run, the GGUF metadata must identify a ‘gemma4a‘ projector and declare a supported shape. The loader reads block count, hidden width, head count, feed-forward width, and mel count, with 128 as the mel default. It checks that the head width meets the engine's alignment and maximum requirements and that the layer count fits the fixed layer array.

The tensor checks are more specific than a single overall shape check. The two convolution tensors must have three-by-three filters, compatible input channels, and a bounded output channel count. The loader rejects convolution biases, checks the relative attention and depthwise convolution shapes, and obtains every required linear tensor by name. It also reads the per-projection clamp ranges where the encoder format supplies them. An encoder file with plausible high-level dimensions can still fail if a tensor type, orientation, or optional bias differs from what this implementation supports.

This is a supported-format boundary, not a proof that the weights are trained correctly or that their output matches another implementation. Loader success means the engine accepted the declared metadata and tensor layout. A meaningful compatibility check still needs the same sample bytes, same preprocessing, same weight file, same precision, and a row comparison against a reference.

The loader also constructs its feature tables once: the periodic Hann window, FFT cosine/sine tables, triangular mel weights, and projected relative-position keys. These are not recomputed for every frame. Keeping setup work separate from per-clip work matters when measuring a short audio attachment; a cold process can include loading and table construction while a warm chat reuses the already-loaded encoder.

How should a rowwise encoder comparison be interpreted?

Suppose a reference encoder and this engine each produce the same number of rows at width 2,560. Begin by comparing corresponding rows, because the language model consumes rows in order. Cosine similarity asks whether two rows point in nearly the same direction; relative error asks how large the elementwise differences are compared with the reference magnitude. Neither statistic alone tells whether a reply will change.

Then summarize across positions. Report the mean row cosine, the worst row cosine, and the relative error over the full array. A mean can hide one difficult boundary or phoneme row; a worst row can exaggerate one harmless low-energy position. A compact table should keep the row count, width, and comparison population visible beside each statistic.

Finally run the same fixed transcription prompt on the same audio samples. Compare normalized text and word or character errors, while retaining the exact raw transcript. If the result changes at a chunk edge, investigate chunking and row alignment before attributing the difference to quantization. If rows match but text differs, first rule out sampling, prompt, language-model weights, and inference settings.

PREPRINT

The documented Q8_0 comparison makes this caution concrete: the file's audio rows had a small aggregate relative error, but one row reached cosine 0.90. That is not evidence that Q8_0 generally breaks transcription; the listed test answers remained correct. It is evidence that both aggregate and per-row views reveal different facts, and that a small mean error does not bound the worst position tightly.

What does "25 rows a second" mean for total work?

The encoder produces 25 output positions per second, but each position passes through 12 blocks whose projections are reused over the sequence. A 7.9-second clip yields about 197 rows in the ENGINE-MATH §8.4 measured example. That section estimates around 117 GFLOP for the audio path; the measured warm encode in that documented run was 96 ms. For 17.5 seconds the historical table lists 437 rows and 260 GFLOP, with 187 ms encode time. These are recorded observations from the project documents, not numbers measured in this chapter.

The activation sequence length changes the work in two ways. Most linear projections scale with the number of rows because each row is multiplied by the same learned matrix. Local attention also scales with the rows, but its key window stays capped at 12 rather than growing to the full clip length. This differs from full self-attention, whose pair count grows with the square of sequence length. The audio path's local window therefore keeps attention work predictable as a chunk grows to 30 seconds.

The 30-second split is still meaningful even though local attention is bounded. The feature front end, projections, and FFNs process every row; a longer chunk needs more activation storage and matrix work. Chunking caps that working set. It also fixes where attention and convolution context reset, so increasing the chunk size would change both resource use and model input values near boundaries.

The model receives 25 rows each second of audio, then performs its own prompt prefill over them. The audio encoder's duration and the language-model prefill duration are different measurements. A report that says "audio took 187 ms" should state whether it measures only encoding, prompt prefill, or the end-to-end request. The outlined command reports the encoder timing in its ‘audio:‘ diagnostic; wall time around the full command includes more stages.

Why is the input normalized to mono and 16 kHz?

The audio encoder's feature tables assume a 16 kHz sampling rate and a single channel. A WAV input can start at another rate and have multiple channels. The fallback reader averages channels and resamples with a windowed sinc filter; the macOS media reader asks AVFoundation for mono linear PCM at the requested rate. Both paths present mono samples to the same encoder front end.

Resampling changes the sample count as well as the waveform. A 48 kHz, three-channel recording is not handled by simply skipping two samples: first the channels are mixed, then the band-limited interpolation maps the signal to 16 kHz. The WAV path applies a Hann-windowed sinc filter with 16 zero crossings on each side and a cutoff tied to the lower Nyquist frequency. That cutoff suppresses frequency content that the output rate cannot represent.

This makes file decoding a separate potential source of differences. The same encoded MP3 may decode to slightly different samples under two libraries; a different channel mix or resampler also changes the mel rows. To compare encoders, first compare the actual float sample arrays or use a single pre-decoded WAV at 16 kHz mono for both runs. Otherwise an observed row difference may begin before the encoder weights are involved.

The media build accepts formats supported by the operating system, including common compressed formats and audio tracks in video. Without the AVFoundation media build, the C fallback reads WAV only. This capability difference belongs to the reader, not to the audio model. The model still sees the same abstract input type: a pointer to mono float samples and a sample count.

PREPRINT

What happens to a short clip and its edges?

The feature extractor pads the beginning by 160 zeros before copying in the samples. It allocates enough padded space to read a complete 512-value FFT input for every counted frame. The frame count follows the model-compatible centered framing convention rather than a simple "one window for every 320 samples" rule. This is why the first and last features depend on the exact sample count and padding policy.

The chunk function checks the frame count before allocating the feature buffer. If fewer than one feature frame can be formed, it returns no rows. The outer audio loop skips such an empty chunk. A tiny recording therefore may produce no media positions, even if a file reader successfully decoded some samples. A voice interface can choose a minimum-duration policy before invoking this path, but it should not silently assume every nonempty file produces a token row.

The convolution edges have their own padding behavior. A 3-by-3 convolution treats out-of-range coordinates as zero; the causal depthwise convolution ignores earlier taps whose time index is negative. These are separate edge rules applied at different representations. The first operates over mel feature time and frequency, while the second operates over the Conformer sequence after both subsampling stages.

At the end of a 30-second chunk, there is no right-side convolutional lookahead. At the start of the next chunk, however, its first feature window receives the standard leading padding again, and its causal history begins empty. The discontinuity is thus explicit. Concatenating the output rows makes one prompt sequence, but it does not retroactively join the encoder context between chunks.

How can source comments guide a safe port?

The opening comment above the audio implementation is a compact specification: sample rate, chunk size, feature recipe, convolution layout, block order, attention context, and output row rate. It is useful as a checklist, but a port should still confirm the details against the executable code. In this chapter, for example, the C attention task shows the actual score and softmax loops, while the Metal comment describes how lanes divide the head dimensions.

The activation structure provides a second checklist. Its declarations specify type and shape for each scratch array, including which arrays are F32 and which are F16. Following the allocator then reveals padded row counts and ownership. Comments such as "the last tap on this frame" on the depthwise weights make the causal kernel's orientation explicit; without that orientation, a convolution can be flipped in time and still look superficially reasonable.

The loader checks provide a third checklist. Shape validation captures assumptions that prose may omit: no convolution bias, expected filter dimensions, bounded channel counts, compatible projections, and supported head widths. A port that only copies the compute loop but skips these validations can accept a malformed file and then read tensors using incompatible strides. Validation is part of a reliable model reader.

One source comment states that the audio encoder follows llama.cpp's ‘mtmd‘ path. That is useful attribution, not an independent proof that every output matches. The ENGINE-MATH notes explicitly mark the full reference-output cross-check as unverified at the time of that document. The porting test should therefore compare preprocessed features and final rows against a named reference version, with precise files and sample bytes.

What can a layer trace tell us?

For debugging, a useful trace records shapes at the boundary of each stage: feature frames and 128 bands; the first convolution's time and band counts; the second convolution's time and 32 output channels; the projected encoder width; and final output width. It should also record the exact sample count and per-chunk row counts. That trace catches off-by-one lengths before a transcript comparison obscures the source.

PREPRINT

Value traces can begin with simple invariants. Feature values should be finite because the mel energy is floored before the log. ReLU outputs should be nonnegative. The GLU output should remain at encoder width. Attention probabilities over the available keys should sum approximately to one, and a row at time zero should have one visible key. The final row count should match the integer sequence-length calculation, and each row should contain the projection width expected by the E4B language model.

After invariants pass, compare selected intermediate arrays. The first test should use synthetic values that make indexing obvious: a feature plane with one nonzero location can expose convolution offset order, and a sequence where each frame holds its frame index can expose causal tap orientation. This is more diagnostic than running an entire speech clip and comparing only the final transcript.

For a loaded model, use one short fixed clip and collect a small number of stage checksums or selected values, then scale up to rowwise comparisons. Avoid dumping every large activation by default; it can create huge logs and slow the run. Record the model file's hash or exact name, encoder quantization, build flags, CPU/GPU path, and sample preprocessing. A future reader should be able to tell whether a difference came from weights, arithmetic, or the audio reader.

Why can the same rows lead to a different transcript?

The encoder output is only the input representation. The prompt says what task to perform, and the language model decides which text to generate. A transcription prompt asks for words; a summary prompt asks for a compressed account; a question may ask for a speaker, topic, or intent. Therefore, transcript evaluation must hold the prompt constant as well as the sample and encoder file.

Even for a fixed transcription prompt, a transcript match is a coarse signal. Two encoders can produce rows with different values while the language model's top choices remain the same. Conversely, a small row perturbation at a difficult word can change one token and create a visible word error. A complete comparison should include row statistics and task output, rather than use one as a substitute for the other.

The project snapshot's accuracy figures show this separation. The README describes specific synthetic voice examples and reports that the three Thai samples were transcribed exactly in that small comparison. It explicitly warns that synthetic voices are easy speech and that the small set is not a real-world error-rate estimate. Those results establish a documented test, not a general guarantee about accents, rooms, speakers, or background noise.

Silence makes the interface especially visible. Since there is no VAD, silence still passes through framing, convolutions, the Conformer, and projection. The language model receives a regular sequence of rows and may answer with a filler. An application might measure signal energy, ask for a minimum audio duration, or use a separately evaluated silence policy, but those would be application-level choices; they are not behaviors of this encoder implementation.

The snapshot's ENGINE-MATH document explicitly says that exact audio encoder output against the reference implementation had not been tested in that analysis. The project README later records measured comparisons for specific files and clips, but those records are historical evidence tied to their stated setup. A new output comparison is a reader experiment and should identify the encoder file and clip.

What happens after the encoder returns?

The result buffer is an array of F32 rows with the encoder's projected width. The media part records its row count and owns the buffer until prompt rendering and prefill have consumed it. The prompt renderer emits an audio-start marker, the media rows, and the matching audio-end marker. These rows are not converted back to text and are not looked up in the tokenizer vocabulary; each row is one model input position.

This is the same mixed-entry design Chapter 16 used to explain image rows. A rendered prompt can alternate token IDs and media embedding rows. The language model's staging code copies token embeddings and media rows into a common input buffer in the order chosen by the template. The KV

PREPRINT

cache then stores the language model's computed state for those positions. The audio encoder's own scratch arrays have already served their purpose and been released.

The sequence length therefore affects both sides of the boundary. It determines the number of rows retained for the media part and the number of language-model positions to prefill. Reducing encoder rows would change the representation; omitting them from the staging buffer would change the prompt. A memory optimization must preserve both ordering and lifetime until the consumer has safely copied or processed the data.

For a 10-second clip, the roughly 250 audio rows are only one portion of the prompt. The user instruction and role markers contribute token entries. A reply may add generated token positions and later reuse the valid prompt prefix, as Chapter 16 described. If the conversation is edited or the audio attachment is replaced, the prompt prefix comparison must treat the new media rows as new input even if the surrounding text is unchanged.

The audio path is thus a boundary between two models of data. The encoder works with sample time, frequency bins, convolution grids, and Conformer frames. The language model works with an ordered sequence of input positions and a KV cache. The projected row width makes the handoff possible; the renderer and media lifetime rules make it correct. Chapter 19 follows that common prompt boundary for images, audio, video frames, and their shared memory budget.

Why use an intermediate projection width?

The Conformer works at width 1,024, which the metadata-only probe confirmed for this E4B encoder. The output projection first maps each Conformer row to 1,536 values, normalizes that intermediate vector, and then maps it to the language model's width of 2,560. The encoder and language model can therefore be designed with different internal widths while still meeting at a well-defined prompt interface.

The memory effect is visible in one row. A 1,024-value F32 row has 4,096 bytes; a 1,536-value row has 6,144 bytes; a 2,560-value F32 row has 10,240 bytes. For 250 rows, those arrays contain 1,024,000, 1,536,000, and 2,560,000 bytes respectively. These are decimal byte counts and describe one array at each stage. Actual peak memory depends on which inputs, outputs, and scratch arrays overlap in lifetime.

The final projection is not simply a reshape or a copy. It learns how encoder features correspond to the language model's hidden representation. Its output norm and projection weights are part of the encoder file. If an application tried to bypass this projection because both systems accept vectors, the result would have the wrong width and the wrong learned coordinate system.

Projection also clarifies the distinction between "encoder rows per second" and "bytes per second." The time rate is about 25 positions. The scalar rate after the output projection is about 64,000 values a second at width 2,560. If those values are held as F32, that is 256,000 bytes a second before prompt staging and other overhead. The prompt side may store or convert embeddings differently, so this calculation describes the returned encoder array only.

The real code

The walkthrough follows the same order as the data: sample features, convolution rows, local attention, full chunk flow, then the GPU counterpart. The C implementation comes first so the fallback and data layout remain visible.

The FFT uses an iterative butterfly schedule. Bit reversal puts each input where the first stage expects it; each later stage combines pairs separated by half the current span. The loop performs all 512-point work without recursion or a temporary transform array. Its inclusive source range is short enough to see the whole transform.

```
2904 static void fft(const Audio *A, float *re, float *im) {
2905     for (int i = 1, j = 0; i < A_FFT; i++) {  // bit-reversed order
2906         int bit = A_FFT >> 1;
2907         for (; j & bit; bit >>= 1) j ^= bit;
```

PREPRINT

```
2908            j ^= bit;
2909            if (i < j) {
2910                float t = re[i]; re[i] = re[j], re[j] = t;
2911                t = im[i], im[i] = im[j], im[j] = t;
2912            }
2913        }
2914        for (int len = 2; len <= A_FFT; len <<= 1)
2915            for (int i = 0; i < A_FFT; i += len)
2916                for (int k = 0; k < len / 2; k++) {
2917                    const float wr = A->cos_t[k * (A_FFT / len)], wi = -A->
    sin_t[k * (A_FFT / len)];
2918                    const int a = i + k, b = a + len / 2;
2919                    const float xr = re[b] * wr - im[b] * wi, xi = re[b] *
    wi + im[b] * wr;
2920                    re[b] = re[a] - xr, im[b] = im[a] - xi, re[a] += xr, im[
    a] += xi;
2921                }
2922    }
```

Listing 18.1: gemma4.c:2904–2922

The feature row multiplies the 320 samples by the precomputed window, zero-fills the rest of the 512-point input, takes magnitudes, applies each triangular filter, and floors before taking the natural log. This is CPU work, parallelized over frames by the caller. The input pointer advances by one 160-sample hop.

```
2924    static void mel_frame(void *ctx, int t, int tid) {  // frame t: log-mel
        of pad[t · hop, + win) · window
2925        (void)tid;
2926        const AActs *a = ctx;
2927        const Audio *A = a->A;
2928        const float *s = a->pad + (size_t)t * A_HOP;
2929        float re[A_FFT] = {0}, im[A_FFT] = {0}, mag[A_FFT / 2 + 1];
2930        for (int i = 0; i < A_WIN; i++) re[i] = s[i] * A->window[i];
2931        fft(A, re, im);
2932        for (int k = 0; k <= A_FFT / 2; k++) mag[k] = sqrtf(re[k] * re[k] +
        im[k] * im[k]);
2933        for (int m = 0; m < A->n_mel; m++) {
2934            double sum = 0;
2935            const float *w = A->mel + (size_t)m * (A_FFT / 2 + 1);
2936            for (int k = 0; k <= A_FFT / 2; k++) sum += mag[k] * w[k];
2937            a->mel[(size_t)t * A->n_mel + m] = (float)log(sum > 0.001 ? sum
        : 0.001);
2938        }
2939    }
```

Listing 18.2: gemma4.c:2924–2939

Next, inspect the CPU path for one first-convolution output row. The im2 columns implement a 3-by-3 receptive field with zero padding and stride two. The row then accumulates each channel's nine weights, normalizes the channel vector, applies ReLU, and writes F16. The row boundary is the memory optimization: an entire F32 activation plane is unnecessary.

```
2980    static void c1_row(void *ctx, int r, int tid) {
2981        (void)tid;
2982        const AActs *a = ctx;
2983        const int n = (int)a->A->conv[0].ne[1];
2984        const float *w = (const float *)a->A->conv[0].data;
2985        float col[9], y[1024];
```

```
2986      conv_cols(a->mel, NULL, a->n_frames, a->A->n_mel, 1, r, a->B1, col,
      NULL);
2987      for (int c = 0; c < n; c++) {
2988          float acc = 0;
2989          for (int k = 0; k < 9; k++) acc += col[k] * w[c * 9 + k];
2990          y[c] = acc;
2991      }
2992      ln_relu(y, a->A->conv_norm[0], n, a->A->eps, a->c1h + (size_t)r * n)
      ;
2993  }
```

Listing 18.3: gemma4.c:2980–2994

The C attention reference makes the local window concrete. ‘nk‘ grows from one at the start to 12 once enough prior frames exist. For each distance, the score combines a scaled q/k dot product and a q/relative-position dot product, then applies the 50-point soft cap. Softmax weights mix the values at positions ‘t‘, ‘t-1‘, and so on. The separate score and value loops make the normalized mixture easy to inspect.

```
3011  static void a_attn_task(void *ctx, int task, int tid) {
3012      (void)tid;
3013      const AActs *a = ctx;
3014      const Audio *A = a->A;
3015      const ALayer *L = a->L;
3016      const int E = A->n_embd, hd = A->hd, h = task % A->n_head, t = task
      / A->n_head, nk = t + 1 < A_CTX ? t + 1 : A_CTX;
3017      float q[256], qk[256], o[256] = {0}, s[A_CTX], mx = -INFINITY, sum =
       0;
3018      for (int i = 0; i < hd; i++) q[i] = a->q[(size_t)t * E + h * hd + i]
       * L->q_scale[i], qk[i] = q[i] * L->k_scale[i];
3019      for (int d = 0; d < nk; d++) {
3020          const float sc = dot(qk, a->k + (size_t)(t - d) * E + h * hd, hd
      ) + dot(q, L->rel + (size_t)d * E + h * hd, hd);
3021          s[d] = 50.0f * tanhf(sc / 50.0f);
3022          if (s[d] > mx) mx = s[d];
3023      }
3024      for (int d = 0; d < nk; d++) sum += s[d] = expf(s[d] - mx);
3025      for (int d = 0; d < nk; d++) {
3026          const float *v = a->v + (size_t)(t - d) * E + h * hd;
3027          for (int i = 0; i < hd; i++) o[i] += s[d] / sum * v[i];
3028      }
3029      for (int i = 0; i < hd; i++) a->o[(size_t)t * E + h * hd + i] =
      float_to_half(o[i]);
3030  }
```

Listing 18.4: gemma4.c:3011–3031

The chunk function allocates the per-chunk activations, computes features, derives both subsampled lengths, then enters the 12-block sequence. The excerpt below shows the row count and early staging. The second convolution, Conformer, and projection continue later in the same function; their dispatch order is discussed in the text so this listing can remain under the line limit.

```
3085  static float *audio_encode_chunk(const Audio *A, const float *samples,
      int n, int *n_rows) {
3086      AActs a = {.A = A, .F = A->n_ff};
3087      const int E = A->n_embd, P = (int)A->out_proj.w->ne[1], c1 = (int)A
      ->conv[0].ne[1], c2 = (int)A->conv[1].ne[1];
3088      // frames as Hugging Face's feature extractor makes them: the
      samples after 160 zeros, frames of 321
```

```
3089      a.n_frames = (n + A_WIN / 2 - (A_WIN + 1)) / A_HOP + 1;
3090      if (a.n_frames < 1) return NULL;
3091      float *pad = xcalloc((size_t)(a.n_frames - 1) * A_HOP + A_FFT,
          sizeof(float));
3092      memcpy(pad + A_WIN / 2, samples, (size_t)n * sizeof(float));
3093      a.pad = pad;
3094      a.mel = xmalloc((size_t)a.n_frames * A->n_mel * sizeof(float));
3095      parallel_for(a.n_frames, mel_frame, &a);
3096      free(pad);
3097
3098      // subsampling: each convolution as a matmul of its input columns
3099      a.T1 = (a.n_frames - 1) / 2 + 1, a.B1 = (A->n_mel - 1) / 2 + 1, a.T
          = (a.T1 - 1) / 2 + 1;
3100      const int R1 = a.T1 * a.B1, R2 = a.T * A->n_band, T = a.T;
3101      const size_t Tp = ((size_t)T + 31) / 32 * 32, R2p = ((size_t)R2 +
          31) / 32 * 32, f = sizeof(float), hf = sizeof(f16);
3102      const size_t W = (size_t)(a.F > 2 * E ? a.F : 2 * E), H = (size_t)(a
          .F > E ? (a.F > P ? a.F : P) : (E > P ? E : P));
3103      a.c1h = xmalloc((size_t)R1 * c1 * hf);
3104      a.im2 = shared_alloc(R2p * 9 * c1 * hf), a.c2 = shared_alloc((size_t
          )R2 * c2 * f);
3105      a.x = shared_alloc(Tp * E * f), a.xn = shared_alloc((Tp * E > R2p *
          c2 ? Tp * E : R2p * c2) * hf);
3106      a.q = shared_alloc(Tp * E * f), a.k = shared_alloc(Tp * E * f), a.v
          = shared_alloc(Tp * E * f);
3107      a.o = shared_alloc(Tp * E * hf), a.a = shared_alloc(Tp * (E > P ? E
          : P) * f), a.g = shared_alloc(Tp * W * f);
3108      a.u = shared_alloc(Tp * E * f), a.h = shared_alloc(Tp * H * hf), a.
          emb = shared_alloc((size_t)T * A->proj_dim * f);
3109
3110      parallel_for(R1, c1_row, &a);
3111      parallel_for(R2, im2_row, &a);
3112      matmul(a.c2, a.im2, &A->conv[1], R2, 0);
3113      gpu_sync();
```

Listing 18.5: gemma4.c:3085–3113

The host can dispatch attention and the fused convolution operation to Metal. The audio attention kernel mirrors the short causal window and relative score term; the convolution kernel computes the gated depthwise path. These excerpts show the GPU's task mapping and its tap bounds. The launch wrapper and Metal operation both return to the common C block flow, which retains CPU fallbacks.

```
850 kernel void audio_attn(device const float *q [[buffer(0)]], device const
        float *k [[buffer(1)]],
851                        device const float *v [[buffer(2)]], device const
        float *qs [[buffer(3)]],
852                        device const float *ks [[buffer(4)]], device
        const float *rel [[buffer(5)]],
853                        device half *o [[buffer(6)]], constant
        AudioAttnParams &p [[buffer(7)]],
854                        uint tg [[threadgroup_position_in_grid]], uint sg
        [[simdgroup_index_in_threadgroup]],
855                        uint lane [[thread_index_in_simdgroup]], uint nsg
        [[simdgroups_per_threadgroup]]) {
856      const int task = tg * nsg + sg, E = p.n_head * p.hd, nd = p.hd / 32;
857      if (task >= p.T * p.n_head) return;
858      const int t = task / p.n_head, h = task % p.n_head, nk = min(t + 1,
         p.ctx);
859      float qv[8], qk[8], acc[8], s[16];
```

```
    for (int j = 0; j < nd; j++) {
        const int i = lane + 32 * j;
        qv[j] = q[(ulong)t * E + h * p.hd + i] * qs[i], qk[j] = qv[j] *
    ks[i], acc[j] = 0;
    }
    float mx = -1e30f;  // (not INFINITY: the C preprocessor would
    expand it in this source)
    for (int d = 0; d < nk; d++) {
        float part = 0;
        for (int j = 0; j < nd; j++) {
            const int i = h * p.hd + lane + 32 * j;
            part += qk[j] * k[(ulong)(t - d) * E + i] + qv[j] * rel[(
    ulong)d * E + i];
        }
        s[d] = 50.0f * precise::tanh(simd_sum(part) / 50.0f);
        mx = max(mx, s[d]);
    }
    float sum = 0;
    for (int d = 0; d < nk; d++) sum += s[d] = exp(s[d] - mx);
    for (int d = 0; d < nk; d++)
        for (int j = 0; j < nd; j++) acc[j] += s[d] / sum * v[(ulong)(t
    - d) * E + h * p.hd + lane + 32 * j];
    for (int j = 0; j < nd; j++) o[(ulong)t * E + h * p.hd + lane + 32 *
     j] = half(acc[j]);
}
```

Listing 18.6: gemma4_metal.m:850–879

```
kernel void audio_conv(device const float *g [[buffer(0)]], device const
     float *dw [[buffer(1)]],
                       device const float *w [[buffer(2)]], device half
    *h [[buffer(3)]],
                       constant AudioConvParams &p [[buffer(4)]], uint t
     [[threadgroup_position_in_grid]],
                       uint tid [[thread_index_in_threadgroup]]) {
    threadgroup float y[4096], tmp[8];
    float ss = 0;
    for (int c = tid; c < p.n; c += 256) {
        float acc = 0;
        for (int j = 0; j < p.taps; j++) {
            const int s = int(t) - (p.taps - 1) + j;
            if (s < 0) continue;
            device const float *gs = g + (ulong)s * 2 * p.n;
            acc += gs[c] / (1.0f + exp(-gs[p.n + c])) * dw[c * p.taps +
    j];
        }
        y[c] = acc;
        ss += acc * acc;
    }
    const float r = rsqrt(block_sum(ss, tmp, tid, 256) / p.n + p.eps);
    for (int c = tid; c < p.n; c += 256) h[(ulong)t * p.n + c] = half(
    silu(y[c] * r * w[c]));
}
```

Listing 18.7: gemma4_metal.m:885–904

Stage	C reference / host	Metal path
Features	`mel_frame`, parallel over frames	none; CPU FFT and mel filterbank
Convolution 1	`c1_row`, LayerNorm, ReLU, F16	none; row streaming avoids full F32 output
Convolution 2	columns plus batched matrix helper and row norm	shared matrix helper can use Metal
Conformer attention	`a_attn_task`, one frame/head task	`audio_attn`, SIMD group per frame/head
Conformer convolution	GLU and depthwise fallback rows	`audio_conv` kernel
FFNs and projections	common linear helpers and CPU fallback	batched matmul and audio helper kernels

The table is a map of the paths, not a claim that every matrix operation always runs on the GPU. The ‘STEP‘ and ‘GPU‘ wrappers decide based on the active build and supported operation. The CPU path remains useful for correctness checks and non-Metal builds, while the Metal kernels avoid moving large intermediates back to the host at every substep.

PREPRINT

Experiment: inspect row counts without weights

Experiment W0

The source computes the final time length by applying ceiling division by two twice. This weight-free calculation checks expected row counts for common durations; it does not run the encoder or predict exact framing for a particular file.

```
nice -n 10 python3 -c 'for seconds in (10, 17.5, 30, 70):
    samples = round(seconds * 16000)
    frames = (samples + 160 - 321) // 160 + 1
    rows = (frames + 3) // 4
    print(
        f"{seconds:>4}s samples={samples:>7} "
        f"frames={frames:>4} rows={rows:>4}"
    )'
```

The captured output is in `code/expected/ch18_w0.txt`. Ten seconds gives 999 feature frames and 250 rows; exactly 280,000 samples gives 1,749 frames and 438 rows; a full 30-second chunk gives 2,999 frames and 750 rows. Seventy seconds gives 1,750 rows. This command calculates one duration at a time; the actual encoder divides 70 seconds into three independent chunks of 30, 30, and 10 seconds. The arithmetic reads no model file and uses no GPU.

Experiment W1

Run the standard-library reader on the local encoder file. It walks the GGUF header and typed metadata, then stops before tensor descriptors and payload. The captured output is in `code/expected/ch18_w1.txt`, recorded 2026-10-22 on Mac15,12.

```
MM="$HOME/Documents/claude/gemma4-e4b-chat/"\
"mmproj-gemma-4-E4B-it-q8_0.gguf"
nice -n 10 python3 code/examples/ch18_audio_header.py "$MM"
```

The selected fields were:

```
clip.audio.attention.head_count = 8
clip.audio.block_count = 12
clip.audio.embedding_length = 1024
clip.audio.feed_forward_length = 4096
clip.audio.num_mel_bins = 128
clip.audio.projector_type = gemma4a
clip.has_audio_encoder = True
GGUF v3; tensor descriptors: 1411; payload read: no
```

This reports ‘gemma4a‘, 12 audio blocks, width 1,024, 8 audio attention heads, feed-forward width 4,096, and 128 mel bins. The 8 audio heads give 128 values per head. They are distinct from the language model’s 8 query heads and 2 KV heads. The 1,411 tensor count belongs to the combined vision/audio file, not the audio tower alone. This confirms header metadata only, not tensor validity or encoder output. The idle-GPU gate is not applicable.

Experiment W3

Try it on your machine (W3): use a known speech clip and save the audio rows for comparison:

```
nice -n 10 code/build/gemma4 -m "$MODEL" --mmproj "$MM" \\
  --audio speech.wav -p "Transcribe this audio verbatim."
```

Inspect the reported duration, row count, encode time, transcript, and optionally compare embedding rows against the reference encoder. **Try it on your machine (W3):** inspect duration, row count, encode time, transcript, and optional embedding-row comparison. The README’s 187 ms for a 17.5-second clip is a dated project measurement (README, audio results), not fresh output here.

Pitfall

Pitfall

Symptom. Silence or room noise can produce a short invented filler instead of an empty transcript. **Cause.** The audio path has no voice-activity detector. The language model receives encoder rows even when the recording contains only silence or noise, then may interpret them as weak speech. **Fix.** Treat empty-audio handling as an application decision and evaluate it against real recordings; do not claim that the encoder itself detects silence. In the documented set of 100 recordings Claude Omni kept, mostly noise, E4B returned nothing for 94. This is a historical observation of the full pipeline, not a universal rate. Source: snapshot `ENGINE-MATH.md`, section 11.7, “Silence and room noise”; the no-VAD behavior is also described in snapshot `README.md`, “Audio”.

Draft book, October 2026. Not peer reviewed.

The 30-second split is another boundary worth remembering. The README documents independent chunk encoding, so the local attention history and causal convolution context restart at each split. A long recording can still be presented as one audio attachment, but its internal representation is made from separately encoded pieces.

The available accuracy records also have limits. The README's transcription comparison used synthetic voices and three Thai clips; it is not a real-world error-rate study. ENGINE-MATH records an audio Q8_0 comparison where one row had cosine 0.90 despite a small aggregate relative error. Similar aggregate statistics can hide an individual row with a noticeable deviation, so compare rowwise values as well as a global summary.

Exercises

1. Starting from 10 seconds at 16 kHz, calculate the sample count, approximate feature-frame count, and output rows after the two stride-2 convolutions. Explain why this duration calculation is not an exact per-file guarantee.
2. For a 70-second recording, split samples into chunks of at most 480,000. Estimate the rows per chunk and state what context does not cross the two boundaries.
3. Explain why 320 samples are windowed but the FFT has 512 points. What changes if a programmer mistakenly takes power instead of magnitude before the mel filters?
4. Trace one output row of the first convolution. Identify its nine spatial inputs, the channel normalization, the activation, and the storage format. Why does rowwise output reduce the temporary memory requirement?
5. In local attention at frame 20, list the visible frame indices. Explain the separate roles of the relative-position term and the causal convolution.
6. Design a W3 comparison for two encoder files: name the fixed inputs, rowwise statistics, transcript check, and the measurement conditions you would record. Why can a correct transcript coexist with changed embedding rows?

Rebuild checkpoint m5b

Implement the audio front end as the optional extension after checkpoint m5a's text attention and KV cache. Start with a deterministic 16 kHz sample buffer, one centered 320-sample window, the periodic Hann weights, 512-point FFT, magnitude, and 128 mel sums. Check the shape and finite values before introducing any model weights. Then implement the two stride-2 convolution row builders and verify their output dimensions against the C reference on a small synthetic array.

Only after those checks should the rebuild add the Conformer: preserve half-step FFNs, local causal attention over at most 12 frames, learned relative positions, the soft cap, and the five-tap causal depthwise convolution. The end-to-end reader experiment requires a licensed local encoder file. A successful checkpoint emits rows with width 2,560 and uses the same media markers and prompt-row boundary that Chapter 19 follows.

Small implementation. A small rebuild can use synthetic filters and weights to validate indexing, but it must label that as a structural check. It should preserve the sequence of operations, the channel/band order, the beginning padding, and the exact odd-length rounding rules. It should compare intermediate shapes after each convolution before comparing final output rows.

When real weights become available to an approved test, the educational implementation should first match one mel frame against the engine for the same input samples. Then it can match one first-convolution output row, one second-convolution row, one attention task, and one final projected row. These staged checks reduce the search area when a result differs. Running a full transcript first provides less information because many operations and model layers can contribute to a changed word.

Only after the CPU reference matches should a Metal rewrite be considered. Test the attention kernel with a short sequence where every frame and head can be inspected, then test the causal convolution with

PREPRINT

values that identify each tap. The GPU and CPU should consume identical arrays. A test that generates its inputs independently on each path can accidentally hide a staging or padding mismatch.

The checkpoint is optional because the earlier book chapters build a working text engine without audio. Its value is to make the ears' input path understandable and reproducible. The required interface at the end is modest: a function accepts mono 16 kHz samples and a count, returns zero or more width-2,560 rows, and makes ownership and failure conditions explicit. The sophisticated work lies behind that interface in the feature contract and the learned encoder blocks.

The practical lesson is that speech reaches the language model through a sequence of carefully shaped rows. Framing and subsampling set the time scale; the Conformer adds local context; projection converts the representation to the language model's width. Reading the encoder is therefore a lesson in signal processing, causal sequence models, and the interface between a specialist encoder and a general language model.

PREPRINT

Chapter 19

Media in the conversation

PREPRINT

The question

How do pictures, sound, and video share one prompt and one memory budget?

Reading map: 14 pages

The chapter follows the outline's fourteen-page estimate. The allocation reserves room for the prompt diagram, media-part model, staging-buffer example, video selection, encoder lifetime, quantised encoder file, source listings, W0/W1 checks, documented pitfalls, exercises, and a final optional rebuild checkpoint.

Section	Pages
Question and shared prompt map	1
Parts, markers, and a mixed staging call	3
Video frames, timestamps, and soundtrack	2
Encoder mapping and media-row memory	3
Q8_0 encoder copy and memory budget	1
Source walkthrough and AVFoundation path	1
W0/W1 checks and W3 reader memory experiment	1
Documented pitfalls, exercises, and checkpoint	2
Total	**14**

The idea

Chapters 17 and 18 followed an image and a sound through their specialist encoders. This chapter starts where those paths meet: at a user message. The chat layer keeps each attachment as a typed part, then presents its rows and marker tokens to the same language model that reads text. Video is a sequence of image parts labelled with times, optionally followed by rows from its audio track. The encoders do not produce words; the language model interprets their rows in the message context.

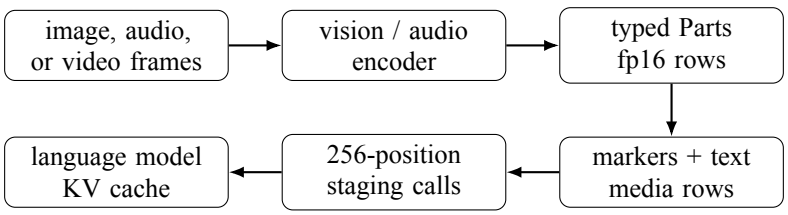


Figure 19.1: Encoders contribute rows to an ordinary conversation prompt; the chat loop stages them in bounded calls.

Parts share a prompt

What does the message retain?

A `Part` is a small record: its kind, a pointer to the encoded rows, their count, and an optional time string. A `Parts` collection owns a growing array of these records for one message. The row pointer is not a token ID. It points to vectors of width 2,560, the E4B language model's embedding width. Text remains ordinary token IDs until the chat loop needs a mixed call.

This arrangement keeps three concerns separate. The media reader turns a file into pixels or samples. The vision or audio encoder turns those into rows. The message keeps the row buffer and its kind until the conversation no longer needs it. It is then possible to write one message containing text plus an image and a recording without making the tokenizer pretend that a vector is a word.

The audio and image paths use different opening and closing markers: `<|audio>` … `<audio|>` and `<|image>` … `<image|>`. A video frame uses the image markers too; its timestamp is ordinary text placed just before that frame. These markers tell the model which kind of row sequence follows. They are not a conversion of the content into a caption, and the markers alone do not explain what is pictured or spoken.

How does text meet an encoder row?

The renderer walks messages in conversation order. At the start of a message it writes each media part, then the user's text. For an image part it appends the opening marker, one prompt entry per row, and the closing marker. Text entries hold token IDs. Media entries hold a row-array pointer and an index into that array. A prompt entry therefore has one of two sources: a token to embed, or an already-encoded vector to copy.

Here is a reduced view of the rendered input for a user who asks a question about a photo and a short recording:

```
<|turn>user
<|image> [row 0] [row 1] ... [row 279] <image|>
<|audio> [row 0] [row 1] ... [row 249] <audio|>
What is happening here?
<turn|>
```

The bracketed labels describe vectors, not literal prompt text. The actual renderer stores entries, and `chat_run` decides how each entry becomes an input vector. This also explains why a media prompt can be much longer than the visible question: every encoder row takes a position in the model's context and occupies space in the KV cache after prefill.

For the E4B, image rows are read causally like the rest of the prompt. Chapter 17 noted that this differs from the bidirectional treatment in the 26B path. That distinction affects how an image-sized run is scheduled: E4B can mix its image positions with neighboring text in bounded calls. The older bidirectional path may need to process one image run as a whole, so a single batching rule must account for both model families.

Why are media calls bounded?

The chat loop caps its usual input call at 256 positions. This is a practical activation-memory limit: a 256-row call is about as fast per position as a 512-row call in the recorded engine work, but keeps roughly half the prompt activations live. Images in the bidirectional path are a special case and may be run whole. In E4B's causal path, a 280-row image crosses the 256-position boundary and continues in another call.

For one mixed call, the loop copies each media row into the call's staging buffer and obtains each text position's embedding into the same buffer. A small ID array accompanies it so the model can apply token-specific per-layer embedding work to text positions while treating media positions as supplied vectors. The forward pass then consumes one sequence with a consistent width. The staging buffer is temporary; the message's row arrays remain available for a later cache rebuild or another turn.

PREPRINT

This is why a 256-position staging limit does not cap an attachment at 256 rows. It caps one forward call. The prompt may contain many more positions, spread over multiple calls. Those calls append their results to the KV cache, while the chat loop retains the media rows in case it must recompute part of the conversation. Context length and row-buffer memory are related but distinct budgets: one counts model positions; the other counts bytes retained in ordinary memory.

At width 2,560, one E4B row in half precision occupies 2,560 × 2 = 5,120 bytes. The retained media row is fp16, but the shared staging array is float32: `Chat.stage` has type `float *`. A 256-row staging block therefore holds 256 × 2,560 × 4 = 2,621,440 bytes, or 2.50 MiB, for its vectors alone. That is twice the size of an equally wide fp16 media buffer. It is still modest beside many rows kept for an entire video. It does not include the KV cache, token IDs, allocator overhead, encoder activations, or the model's weights.

How does prefix reuse recognize the same media?

A cached prompt entry needs to refer to the same input as before. For text, the engine compares token IDs. For media, it compares the row-array address and the row index. This is cheap: a frame with 70 rows contributes 70 entries, but comparing the prefix does not require another 70 × 2,560-value scan. It also means that two separately encoded copies of the same picture are not automatically treated as the same prefix. Their row arrays have different addresses, so the engine safely recomputes from the first changed media entry.

Consider a message that starts with an image, followed by the text question. The rendered entry sequence begins with the user marker, image marker, the image's row entries, closing image marker, then the question's tokens. If a later turn keeps that exact media part and adds a new question, the earlier entries can match and the forward pass can start after the shared prefix. If the image changes, its marker may still match, but the first row entry points to a different array; reuse stops there and the following rows and text are recomputed.

The row address is only meaningful while the backing ‘Part.rows‘ allocation is alive. When a user removes or replaces pending media, the chat code first renders the remaining conversation and shortens the cached prefix to what still matches. Only then may the removed rows be freed. Otherwise an allocator could give the same address to a different image, and pointer equality might make an obsolete cache entry look current. In the `/undo` branch, the source comment states this ordering concern. `chat_forget_pending` renders the remaining conversation and shortens the cached prefix; the caller then frees the pending rows. This is a lifetime rule: cached entries borrow row storage; they do not own another copy.

The same rule helps explain why fp16 media rows stay in ordinary memory even though the model consumes staged float rows. The retained part must outlive any cache reuse decision and remain available for replay; the staging buffer only needs to survive until its current forward call has finished. Keeping those lifetimes distinct avoids both a permanent GPU copy of every attachment and a use-after-free when a conversation is edited.

Prompt entry	Identity used for reuse	Storage lifetime
Text token	token ID	tokenizer/message history
Image or frame row	row-array address and row index	message's retained fp16 rows
Audio row	row-array address and row index	message's retained fp16 rows
Staged model input	not a prefix identity	one forward call

Chapter 16 follows the cache-prefix rules for text turns; this media case adds a pointer-backed input to the same comparison. It does not introduce a new cache format. The practical boundary is simple: render and compare while the parts are alive, shorten the reusable prefix when the history changes, and free only the rows that no remaining entry can reference.

PREPRINT

Does a reusable prefix give positions back?

No. Prefix reuse saves work by avoiding a second forward pass over entries whose inputs still match, but those entries remain part of the rendered conversation and still occupy context positions. A media row uses one position just as a text token does. Its fp16 buffer may be much smaller than the model's internal activations, yet the model still writes keys and values for that position into its cache. The row-byte budget and the context-position budget answer different questions.

The chat path renders the full conversation, then checks whether its entries fit the configured context before starting the reply. With an encoder file, the snapshot configures 16,384 context positions. Its historical ten-second video example used 958 prompt positions with sound and 705 without it. The difference is 253 positions, close to the audio encoder's approximate 250 rows for ten seconds; marker and timestamp entries mean the two quantities are not exactly identical.

For a simple fit check, suppose earlier turns already account for 15,500 positions and the new video message contributes 958. Together that is 16,458 before any generated reply, which exceeds the 16,384-position context. A matching cache prefix cannot make this fit: it only lets the engine reuse computation for that prefix. If the earlier history has 15,000 positions instead, the 958-position prompt leaves at most 426 positions before the context boundary, including room for the reply and any remaining markers or text. The chat path caps generation to the remaining cache room after prompt processing.

This is why a small image or audio attachment is not automatically cheap in every sense. Its retained fp16 rows may occupy only a few megabytes, but each row is also a context position and a source of key/value state. Long conversations with repeated attachments need both budgets considered. Removing old turns can free context, while the media row arrays remain useful only if their messages remain available for prompt reconstruction.

Video is sampled, then labelled

How many frames can one attachment contribute?

Video is not fed as every decoded frame. The engine asks AVFoundation for up to 32 frames, at most one per second, distributed from the beginning across the clip. It first reads the duration, chooses `n = int(duration)` with a minimum of one and a maximum of 32, then sets sample times at equal intervals `duration × i / n`. A 10-second clip therefore requests ten frames; a two-minute clip requests 32, not 120. Each frame is scaled during decoding to no more than 1,024 pixels on either side, then prepared for at most 70 output rows.

The frame row cap is lower than the normal still-image budget. A still image can use up to 280 rows by default, while a video frame is intentionally limited to 70 to keep a sequence of sampled moments affordable. It is a ceiling, not an exact count. Aspect ratio, alignment, and the selected image budget can reduce the returned rows. Ten frames at the ceiling would contribute 700 image rows before timestamps, markers, and audio are counted.

The frame sampler is a summary of motion, not a video understanding model in itself. At one sample per second it can miss a brief event between sample points. The time label tells the language model when each selected image came from; it does not restore the visual evidence of unsampled frames. For a long clip, the 32-frame cap also means sampling becomes sparser as duration increases.

Why is the timestamp prompt text?

A frame is rendered as a time string followed by image markers and that frame's row sequence. Consecutive frame times are separated by spaces. The timestamp is written as ordinary text, for example `00:08`, rather than as a special video token. This gives the language model explicit temporal labels while preserving the same image row interface used for still images.

The decoder may return a frame near the requested time rather than exactly at it. AVFoundation's image generator uses a tolerance bounded by half the spacing between samples and by 0.5 seconds. When

PREPRINT

it returns a numeric actual time, the code replaces the requested time with that value before the chat part receives its label. If a tolerant request fails, the decoder gets one retry allowing any frame. The reported time is therefore tied to the decoded frame, not merely copied from the original schedule.

When does the soundtrack join the message?

If the video contains audio, the media build has an audio encoder, and video sound is enabled, the same file is passed through the audio path after its frames. The video message consequently contains a timed image sequence and one audio part. `--no-video-audio` suppresses that final part. This choice changes the prompt size and what evidence the model can use: a silent frame sequence cannot carry words or events that are audible but not visible.

A ten-second example makes the row arithmetic tangible. At the per-frame ceiling, ten frames contribute 700 rows; ten seconds of audio contributes approximately 250 rows at the 25-row-per-second rate from Chapter 18. Together they are about 950 media rows before timestamp tokens, marker tokens, and the user's question. At 5,120 bytes each, the retained half-precision row arrays are 4,864,000 bytes, or about 4.64 MiB. The snapshot README records 958 prompt tokens for one historical 10-second video test and 705 without its soundtrack; those counts include prompt structure and reflect that test's actual frames, not the ceiling arithmetic here.

The times are also useful to the model only if the source and decode path preserve them. AVFoundation applies the track's preferred transform before returning pixels, so a rotated phone clip reaches the vision encoder upright. It scales large frames before copying their pixels. The ordinary image loader remains `stb_image`; the optional Objective-C file handles audio and video formats macOS can decode. A build without the media module keeps the C WAV reader, which accepts PCM or float WAV and resamples it, but cannot decode video or general audio file types.

Keep the memory budget under control

When are encoder weights mapped for Metal?

The model's main weights are used continuously during text generation, but an encoder is only needed when a new attachment is encoded. The engine identifies the byte span occupied by vision or audio tensors in the memory-mapped encoder file. Immediately before encoding, it maps that span into Metal; after the GPU work is synchronized, it unmaps the span. The operating system may keep those file pages cached, but the GPU mapping is no longer held between attachments.

The `wire` helper is intentionally small. In a CPU-only build it becomes a no-op. In a Metal build it returns early if the GPU is disabled or the span is empty; otherwise it maps on entry and waits for outstanding GPU work before unmapping on exit. The synchronization matters because the GPU must not still be reading a tensor page when the Metal mapping is released. The image, audio, and video call sites bracket their encoder calls with this helper.

Mapping only while the encoder runs trades a little setup work for lower persistent wired memory. The snapshot README says mapping again for a later image or audio item costs a few milliseconds. That is small beside the model lifetime and avoids holding roughly 990 MB of F16 encoder tensors mapped for the GPU while the user is just chatting in text. The unmap does not unload the file or erase its metadata; it releases the GPU mapping, and normal file caching remains under the operating system's control.

Why keep retained rows in half precision?

A conversation can keep its media rows after encoding because the KV cache may need rebuilding when the conversation changes. Keeping these rows in ordinary, unwired memory avoids turning every old attachment into a permanent GPU allocation. The conversion happens once: the encoder returns float rows, `half_rows` allocates a 16-bit array, converts each value, and releases the encoder's float result. Each later call copies only the rows it needs into the shared staging buffer.

PREPRINT

For the E4B width of 2,560, a media row falls from 10,240 bytes in float32 to 5,120 bytes in fp16. The README's five-minute video example estimates about 9,400 retained rows: at 5,120 bytes that is 48,128,000 bytes (48.1 decimal MB), while float32 would use 96,256,000 bytes (96.3 MB). Rounding values to half precision is not bit-neutral, so it can affect a later generated token if two candidate logits are nearly tied. The snapshot documents that possibility instead of claiming identical greedy replies after every long conversation.

The row buffers and the language model's KV cache are separate allocations. The KV cache holds attention keys and values created as the prompt is processed; the row buffers preserve the original encoder output so it can be processed again if needed. When a chat needs more cache positions, its cache starts small and grows. The snapshot reports an initial 4,096-position allocation for a 16,384-position encoder chat and a doubling strategy when a turn needs more room. It copies the global-layer cache into a larger allocation while keeping sliding-window rings. That policy avoids reserving the entire maximum for a short exchange.

Payload estimate	Rows	Bytes
Still image at 280-row ceiling	280	1,433,600
Ten seconds of audio, approximate	250	1,280,000
Ten-second video at frame ceiling, with audio	950	4,864,000
One mixed staging call (float32)	256	2,621,440
Five-minute video and soundtrack, README estimate	9,400	about 48,000,000

The still, audio, and video figures are retained fp16 row payloads; the staging figure is float32. These are payloads only. A real memory measurement includes activations, the model file's wired pages, cache growth, allocator metadata, runtime overhead, and whatever other applications are using the machine. That is why process resident-set size alone is not a sufficient measure for a Metal engine: GPU-wired file-backed pages may not appear as ordinary process memory.

What does the encoder quantisation option change?

The `--quantize` option makes a second GGUF file from the encoder file. It changes eligible two-dimensional F16 weight matrices to Q8_0 while leaving other tensor types and the metadata alone. The source divides each row into 32-value blocks. For each block it finds the largest absolute value, sets a shared scale to that value divided by 127, rounds each scaled value to a signed 8-bit integer, and stores the scale with those 32 codes. It is a file conversion; it does not quantise the main language model or the media rows retained in a chat.

The outline's snapshot README gives 990 MB for the F16 encoder file and 559 MB for the Q8_0 copy. In that historical comparison the encoders ran at roughly the same speed, within the reported ±1%, because the batched matrix multiplies were compute-bound at those row counts. The gain was memory: the smaller file used 140–330 MB less at a media turn's peak in the README's tests. The row comparison found relative errors of 2.0% for image outputs and 2.6% for audio outputs, with one audio row at cosine 0.90; the recorded test answers remained correct. Those are historical observations from a specific test set, not a guarantee for another image, voice, or near-tied answer.

The key separation is among three representations. Q8_0 changes how encoder weights are stored in the file. fp16 changes how encoded rows are retained between turns. The staging buffer contains the rows copied for the next model call. They have different lifetimes and different accuracy or memory tradeoffs. Measuring one does not measure the others.

What happens inside one Q8_0 weight block?

The file-size reduction comes from quantising eligible two-dimensional F16 weight matrices, not from compressing every byte of the encoder file. The converter handles one row at a time and divides its values into blocks of 32. It finds the largest absolute value in a block, divides that maximum by 127 to get a

PREPRINT

shared scale, and rounds each weight divided by that scale to a signed 8-bit code. The block stores one half-precision scale and 32 codes: 2 + 32 = 34 bytes. The same 32 F16 values occupy 64 bytes. That block-level comparison is almost a halving, while untouched tensor types and metadata explain why the whole encoder file does not shrink by exactly half.

Take six values from one block: minus 1.984375, minus 0.5, zero, 0.3125, 0.625, and 1.984375. The largest magnitude is 1.984375, so the scale is 1.984375 divided by 127, or 0.015625. This scale is exactly representable in F16. Dividing the six values by it and rounding gives codes minus 127, minus 32, zero, 20, 40, and 127. Multiplying those codes by the stored scale reconstructs the six values exactly in this small example. The remaining 26 positions in the actual block would use the same scale, and arbitrary trained weights generally round to nearby rather than identical values.

The actual file changes from the README's 990 MB F16 file to its 559 MB Q8_0 result, a reduction of 431 MB, or about 43.5% of the original file size. That percentage is smaller than the saving for an eligible matrix block because not every tensor is an eligible F16 matrix. The converter preserves the header metadata and copies noneligible tensor data at its existing type. This is why the command first emits the tensor descriptors with their changed types and byte offsets, then writes the header and aligned payload in order.

A smaller file is not automatically a faster encoder. The snapshot README reports essentially the same encoding time, within ±1%, because those batched matrix multiplies are compute-bound in the measured shape range. Quantisation mainly lowers the encoder file's memory footprint in this path. A row-accuracy comparison is still needed when choosing a representation: the book records rowwise errors, not only the final transcript, because a transcript can stay the same while intermediate rows differ.

Reading the source

Start and release the encoder mapping

The first excerpt shows the lifetime boundary. `wire` does not copy the encoder weights: it maps the identified file span to Metal. Its unmap branch synchronizes first so the device has finished using the span. The no-Metal branch explicitly consumes its parameters and leaves CPU execution unaffected.

```
2240 static void wire(Span s, int on) {
2241 #ifdef GEMMA4_METAL
2242     if (!use_gpu || !s.n) return;
2243     if (on && !metal_map(s.p, s.n)) die("metal: cannot map an encoder");
2244     if (!on) gpu_sync(), metal_unmap(s.p);
2245 #else
2246     (void)s, (void)on;
2247 #endif
2248 }
```

Listing 19.1: `gemma4.c`:2240–2248

Store media rows with their kind and count

The next excerpt defines the row owner and the conversion point. `Part` holds one media kind, row pointer, number of rows, and frame time. `half_rows` allocates exactly two bytes per component, converts the float results, and frees their shared-memory allocation. The following collection grows geometrically so adding a new part does not require a fixed maximum number of attachments.

```
3289 enum { PART_IMAGE, PART_AUDIO, PART_FRAME };
3290
3291 // rows: fp16, in ordinary (unwired) memory: a call copies them into its
         input buffer (see chat_reply), so the
3292 // conversation keeps half the bytes (a 5-minute video with its sound:
        ~9,400 rows, 48 MB instead of 97)
3293 typedef struct { int kind; f16 *rows; int n; char time[16]; } Part;  //
        time: a frame's, "mm:ss"
3294 typedef struct { Part *p; int n, cap; } Parts;
3295
3296 static f16 *half_rows(float *rows, int n, int width) {  // (frees rows,
        an encoder's result)
3297     f16 *h = xmalloc((size_t)n * width * sizeof(f16));
3298     for (size_t i = 0; i < (size_t)n * width; i++) h[i] = float_to_half(
        rows[i]);
3299     shared_free(rows);
3300     return h;
3301 }
```

Listing 19.2: `gemma4.c:3289–3301`

PREPRINT

Render a mixed call through one staging array

A row pointer inside a prompt entry is the clue that the entry came from media. In a mixed call, each such row is copied from the retained fp16 array into the staging buffer; an ordinary token instead gets its embedding dequantised into that same row slot. The branch sets the per-layer token ID to zero for media and to the real ID for text. This lets one forward call consume a single width-2,560 input array without mistaking a vector for a token.

This excerpt starts inside the mixed-call branch. The loop gathers no more than `T_max` entries, normally 256; a media entry copies its selected row into the float staging buffer and sets its per-layer token ID to zero, while a text entry is embedded into the same array. The E4B path reads images causally, so ordinary image rows can be mixed into these bounded calls. A separate earlier branch handles whole image runs for the bidirectional path.

```
4287             while (i + T < n && T < c->acts.T_max && !(e[i + T].rows &&
     e[i + T].whole)) media |= e[i + T++].rows != NULL;
4288             for (int t = 0; t < T; t++) {
4289                 const Entry *x = &e[i + t];
4290                 if (!media) c->acts.tokens[t] = x->tok;
4291                 else if (x->rows) {
4292                     for (int k = 0; k < E; k++) c->stage[(size_t)t * E +
     k] = f16_table[x->rows[(size_t)x->row * E + k]];
4293                     c->ids[t] = 0;
4294                 }
4295                 else {
4296                     dequant_row(c->m->tok_embd, x->tok, c->stage + (
     size_t)t * E), c->ids[t] = x->tok;
4297                     for (int k = 0; k < E; k++) c->stage[(size_t)t * E +
     k] *= c->m->embd_scale;
4298                 }
4299             }
4300             if (media) c->acts.embd = c->stage, c->acts.pl_ids = c->ids;
4301         }
4302         forward(c->m, &c->acts, &c->st, T, T - 1, 1, c->logits);
4303         c->acts.embd = NULL, c->acts.pl_ids = NULL, c->acts.bidir = 0;
```

Listing 19.3: gemma4.c:4287–4303

Dispatch the video attachment

The dispatch function selects the image, audio, or video reader. The larger `add_video` path checks for a vision encoder, obtains the clip's duration and audio-track flag, schedules frame times, prepares each decoded frame with the 70-row budget, and adds the soundtrack if it is available and enabled. The source constants immediately above it make the 32-frame and 70-row ceilings explicit.

```
3349 enum { VIDEO_FRAMES = 32, FRAME_TOKENS = 70 };  // at most, and per
     frame (Hugging Face's defaults)
3350
3351 // A video: up to VIDEO_FRAMES frames, at most one a second, evenly
     spaced from the start (each FRAME_TOKENS rows
3352 // at most), and its audio track if the model has an audio encoder (and
     enc->video_sound)
3353 static int add_video(const Encoders *enc, const char *path, Parts *ps) {
3354 #ifdef GEMMA4_MEDIA
3355     if (!enc->V) { fprintf(stderr, "no vision encoder: start with --
     mmproj FILE\n"); return 0; }
3356     char err[512];
3357     double dur;
```

```
3358      int has_audio;
3359      if (!media_video_info(path, &dur, &has_audio, err, sizeof err)) {
          fprintf(stderr, "%s\n", err); return 0; }
3360      int n = (int)dur;  // (frames dur / n apart: at least a second)
3361      if (n > VIDEO_FRAMES) n = VIDEO_FRAMES;
3362      if (n < 1) n = 1;
3363      double times[VIDEO_FRAMES];
3364      uint8_t *rgb[VIDEO_FRAMES];
3365      int w[VIDEO_FRAMES], h[VIDEO_FRAMES];
3366      for (int i = 0; i < n; i++) times[i] = dur * i / n;
3367      const double t_read = now();  // (frames decoded at most 1024 wide
          and high: a frame is ~450 × 350 pixels)
3368      if (!media_video_frames(path, times, n, 1024, rgb, w, h, err, sizeof
          err)) { fprintf(stderr, "%s\n", err); return 0; }
```

Listing 19.4: gemma4.c:3349–3368

PREPRINT

Decode a frame near the requested time

The Objective-C media helper delegates decoding to AVFoundation. This excerpt applies orientation, optionally limits the decoded dimensions, and chooses a tolerance window. It retries once with any frame if the tolerant request fails, then writes the actual decoded timestamp back to the caller's time array. The image is converted to top-to-bottom RGB bytes for the C vision path.

```
121             const int iw = (int)CGImageGetWidth(img), ih = (int)
    CGImageGetHeight(img);
122             uint8_t *rgba = calloc((size_t)iw * ih, 4), *out = malloc((
    size_t)iw * ih * 3);
123             CGContextRef ctx = CGBitmapContextCreate(rgba, iw, ih, 8, (
    size_t)iw * 4, cs,
124 
    kCGImageAlphaNoneSkipLast | kCGBitmapByteOrder32Big);
125             CGContextDrawImage(ctx, CGRectMake(0, 0, iw, ih), img);
126             CGContextRelease(ctx);
127             CGImageRelease(img);
128             for (size_t p = 0; p < (size_t)iw * ih; p++)
129                 out[3 * p] = rgba[4 * p], out[3 * p + 1] = rgba[4 * p +
    1], out[3 * p + 2] = rgba[4 * p + 2];
130             free(rgba);
131             rgb[i] = out, w[i] = iw, h[i] = ih;
```

Listing 19.5: gemma4_media.m:121–131

Experiment

Experiment W0

How large are the retained row arrays compared with one temporary call? This weight-free calculation uses the E4B width of 2,560, two bytes per retained fp16 component, and four bytes per float32 staging component. The command does arithmetic only; it reads no model or encoder file and makes no GPU call.

```
nice -n 10 python3 code/examples/ch19_media_budget.py
```

The captured output is in `code/expected/ch19_w0.txt`:

```
image 280 rows: fp16 1,433,600 B = 1.367 MiB
audio 250 rows: fp16 1,280,000 B = 1.221 MiB
video 950 rows: fp16 4,864,000 B = 4.639 MiB
stage 256 rows: f32 2,621,440 B = 2.500 MiB
```

The 950-row line is a ceiling-based ten-second example. Actual decoded frame shapes can yield fewer than 70 rows each, and the row estimate for audio is approximate. The arithmetic is a sizing aid, not a measurement of system memory.

Experiment W1

The two standard-library readers from the preceding chapters can inspect the local encoder GGUF's metadata. They stop after the metadata section and do not read tensor descriptors or payload. This read-only check identifies both towers in the same file; it does not validate matrix contents or run either encoder.

```
MM="$HOME/Documents/claude/gemma4-e4b-chat/"\
"mmproj-gemma-4-E4B-it-q8_0.gguf"
nice -n 10 python3 code/examples/ch17_vision_header.py "$MM"
nice -n 10 python3 code/examples/ch18_audio_header.py "$MM"
```

Captured on Mac15,12, 2026-10-22; the selected output was:

```
clip.has_vision_encoder = True
clip.vision.block_count = 16
clip.vision.embedding_length = 768
clip.vision.patch_size = 16
clip.has_audio_encoder = True
clip.audio.block_count = 12
clip.audio.embedding_length = 1024
clip.audio.num_mel_bins = 128
GGUF v3; tensor descriptors: 1411; payload read: no
```

The reader reports metadata stored in the Q8_0 encoder file. The 768 and 1,024 widths belong to the two specialist encoders, not to the E4B language model. Chapter 17 and Chapter 18 trace how those internal widths become projected rows of width 2,560. The 1,411 tensor count is the combined file's descriptor count. No payload bytes were read by either script, and no GPU-idle measurement applies to a header-only inspection.

Experiment W3

Try it on your machine (W2/W3): measure the same text prompt, one still image, and a ten-second video with sound. Use identical model and encoder files, `-n 20`, and record system memory before and peak rise with `tools/memwatch.sh`. Run the cases one at a time with the idle-GPU gate; save the exact commands and output.

```
nice -n 10 code/snapshot/tools/memwatch.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" -p "Describe the attached media." -n 20
nice -n 10 code/snapshot/tools/memwatch.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" --image photo.jpg \
  -p "Describe the attached media." -n 20
nice -n 10 code/snapshot/tools/memwatch.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" --video clip-10s.mp4 \
  -p "Describe the attached media." -n 20
```

Inspect the peak rise, prompt row count, and any change in the answer. **Try it on your machine (W2/W3):** compare peak memory rise, prompt rows, and answers across the three cases. The 2,825 / 3,189 / 3,224 MB after-values are dated project measurements (README, media memory comparison), not results of this run.

Pitfall

Pitfall

Symptom. Exact seeks worked on some video files but failed on others, including a long clip; the older memory note records a five-minute frame read as failed.
Cause. An exact seek may have to decode forward from a preceding keyframe each time. Some files did not support the requested exact timestamp through that path.
Fix. Ask AVFoundation for a frame within a bounded tolerance around each requested time, use the actual returned timestamp as the prompt label, and retry once with any frame if the first request fails. The snapshot README says the five-minute case then read its frames in 0.27 seconds. That timing is historical and machine-specific; the robust behavior is the tolerance and retry path, not a promised duration. Source: snapshot `README.md`, "Images, audio and video"; implementation: `gemma4_media.m:100--119`.

Another documented tradeoff appears after all encoding is complete. Keeping media rows as fp16 in ordinary memory cuts their retained size in half, but the values are rounded. The README reports a relative difference of 0.02% for that storage conversion and warns that a greedy answer can still diverge later when two token choices are nearly tied. This is a real precision boundary in the conversation path. It does not mean every converted prompt changes, and a matching short answer does not establish that the row arrays were bit-identical. Source: snapshot `README.md`, "Memory," media rows paragraph.

The memory table in that same README compares system-wide wired, anonymous, and compressed memory with `tools/memwatch.sh`. It reports 3,936 MB before and 2,825 MB after for text-only chat with the encoders loaded; 4,276 / 3,189 MB for a 12-megapixel photo; and 4,242 / 3,224 MB for a ten-second video with sound. These are historical snapshot results, not fresh measurements. They show the expected direction of the mapping and storage changes under the recorded setup. They should not be interpreted as the memory cost of an encoder alone, because the measured peak includes model, cache, activations, the operating system, and other processes.

Exercises

1. A ten-second clip samples ten frames, each with a maximum of 70 rows, and its soundtrack yields about 25 rows per second. Estimate the combined media rows and fp16 bytes. Name at least two prompt components omitted from that memory estimate.
2. Trace a media entry through `chat_render` and `chat_run`. What identifies it as a row rather than a token, and why does its per-layer token ID become zero?
3. If a still image produces 280 rows, how many bounded 256-position calls are needed for its image rows in the E4B causal path? Explain why this does not imply that the image is limited to 256 rows.
4. For a 5-minute video, compare the 32-frame ceiling with the 300 frames that a one-per-second sampler would request without a cap. What kinds of short events could the bounded sample miss?
5. Explain the difference between the encoder file's Q8_0 weights, retained fp16 media rows, and the staging buffer. Which one does `--quantize` change?
6. Design a memory comparison that can attribute a change to one media item. State which files, prompt, generated-token limit, system-memory measure, idle-GPU reading, and repeated-run statistic you would keep fixed or report.

Rebuild checkpoint m8: media entries at the chat boundary

Chapter 16's m8 checkpoint remains the formal chat milestone. This optional Part V extension adds typed media entries to the same prompt-prefix machinery: represent a text entry by its token ID and a media entry by a row pointer plus row index; render the correct opening and closing markers; then copy

PREPRINT

either the token embedding or the selected media row into a bounded 256-position input buffer. Keep a two-row synthetic image and a two-row synthetic audio part, with distinct values in each row, so ordering and marker placement are easy to inspect.

First check that the rendered order is marker, row 0, row 1, closing marker, then the user's text. Next run two turns where the first media prefix is unchanged and confirm that the cache reuses it; change the first media row and confirm that reuse stops at that position. This extends m8's prefix test without implementing a vision or audio encoder. For a full engine comparison the local encoder file and inference run are required; the W3 comparison is a reader experiment; the prefix argument predicts reuse only while the media rows remain identical.

PREPRINT

Part VI

RIGHT AND FAST

PREPRINT

Chapter 20

Is it right?

Part VI: Right and fast

The first five parts followed the engine from file to answer, including optional picture and sound inputs. This final part asks two questions that must stay separate: did a change preserve the answer, and did it make the engine faster? Chapter 20 builds the correctness gates. Chapter 21 will measure speed on a shared machine; Chapter 22 will tell the history of the optimisations; Chapters 23 and 24 will return to memory and the whole-system picture. Here, "right" means that we have evidence at the level the change could affect: exact token IDs for greedy behavior, probability comparisons for small numeric shifts, and direct checks for reusable arithmetic.

The question

How do we know the answer is still the model's answer after every change?

Reading map: 18 pages

This first pass allocates the outline's estimate across the working sections. The finished chapter keeps space for the three gates, the CPU and Metal scoring paths, measured W0 output, the W2 reader experiments, a documented failure, and the exercises.

Section	Pages
Part VI opener, question, and gate map	1
Gate 1: exact greedy tokens and high-precision logits	3
Gate 2: perplexity, KL divergence, and top-token agreement	4
Gate 3: tests of the engine's own math routines	3
CPU scoring, Metal scoring, and reporting listings	3
W0 synthetic check and W2 reader reference experiment	1
Documented token-buffer race and why chat tests missed it	1
Exercises, hints, and m7 checkpoint	2
Total	**18**

The idea

Why are three gates better than one?

A generated sentence is an attractive but weak test. A small logit shift can leave every word in a short answer unchanged.

Conversely, two correct implementations can choose different words at one position when the top two scores are almost tied. We need checks that answer different questions, in increasing cost order.

PREPRINT

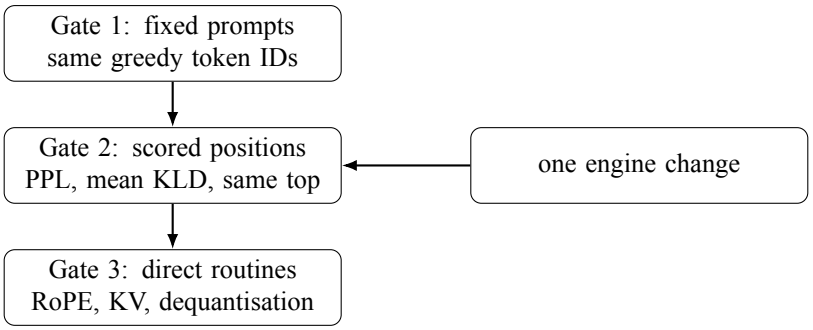


Figure 20.1: Run cheap checks first, then examine distributions and individual arithmetic where a change could hide.

Gate 1 asks whether a fixed greedy prompt produces the same token sequence. It is quick to interpret and catches conspicuous behavior changes. Gate 2 examines the scores at many known next-token positions; it can reveal numerical movement even if the winner is unchanged. Gate 3 isolates operations such as position rotation and cache quantisation, so a failed assertion points closer to its cause. These gates complement one another. Passing a local RoPE test does not prove the chat template is right; matching a handful of replies does not prove the score path is numerically stable.

The gates are deliberately ordered by cost, not by importance. Run the shortest deterministic checks during development. Before accepting a kernel rewrite, run the reference comparison that covers the affected path. A change to prompt rendering needs fixed-token and prefix tests. A change to the decode attention needs score comparisons through the decode path. A change to the tensor reader needs file and dequantisation checks. The scope of evidence should follow the scope of the change.

What does “same tokens” prove?

Greedy generation chooses the largest logit at every step. If the logits are (2.4, 0.1, -1.0), token zero wins by a wide margin. A small rounding change is unlikely to alter that choice. If the logits are (2.400000, 2.399999, -1.0), a shift in the first two values of only a few millionths can reverse the winner. The output text then diverges, and every later input differs too.

For a fixed prompt, compare the integer token IDs, not only decoded strings. Two ID streams can sometimes decode to the same bytes, and a display can hide a stop marker or whitespace token difference. Use the same model file, prompt bytes, context length, sampler settings, and output limit. Greedy sampling removes random-state variation, but it does not make near ties stable. A matching sequence is useful evidence for those prompts; it is not a proof over every possible prompt.

When a failure appears, print the first differing position and the two IDs with their token spellings. Then inspect the top logits at that position in sufficient precision. Six decimal places can expose a small change that ordinary short formatting rounds away. Keep the actual values and position with the report: “the answer changed” is a symptom; the first differing score row helps locate where to investigate.

High precision is a diagnostic, not a new acceptance threshold. A printed decimal value may differ in the last displayed place while the output token is robustly the same. Conversely, a one-unit difference in the sixth decimal place can matter if two candidates are nearly tied. The meaningful test is to compare the underlying score data and determine whether the change is expected for the operation being modified. Do not announce bit identity unless the comparison actually checked every relevant value bit for bit.

A useful tiny example makes the distinction clear. Suppose the reference scores are (1.000000, 0.999999, -2.0). The changed path returns (0.999998, 1.000001, -2.0). The top token flips even though the absolute changes are only a few millionths. In another row, scores (1.0, 0.2, -1.0) could move by the same amount without changing the winner. Token agreement records the discrete decision; a distribution metric records the broader numeric difference.

PREPRINT

How does perplexity add evidence?

Perplexity evaluates the model's score for the known next token over a text sequence. At each position the engine asks, "How much probability did this distribution assign to the token that actually follows?" It averages the negative log probability over the positions, then exponentiates that average. Lower perplexity means the sequence was less surprising to that model under that evaluation setup. It is not a general measure of quality, and comparisons only make sense when the input, chunking, model file and scored positions match.

Perplexity is a summary. Two implementations can have nearly equal averages while disagreeing on individual rows. A single altered score might make one token easier and another harder, with the effects cancelling in the mean. Therefore the engine can also compare each distribution against a stored base distribution and report mean KL divergence, percentiles, and the fraction of positions with the same highest-probability token.

The base file is central to a fair comparison. It stores quantised log probabilities for each scored position. One program can create that file and another can read it, so the engine can be compared with llama.cpp using the same text and chunking. For a self-comparison, first create a base with the reference configuration, then score the candidate with that base. Keep the file, command lines and model identity together; a KL number without its reference is not interpretable.

The book's outline records historical examples at context 512: mean KL divergence 0.000012, with a 99.9

What do KL and same-top tell us?

KL divergence compares two complete probability distributions. If the base distribution assigns probability 0.8 to one token and 0.2 to another, while the candidate assigns 0.7 and 0.3, both choose the same top token but the distributions differ. If their probabilities are reversed, the top token differs too. The KL statistic captures score movement across candidates; same-top agreement answers only whether the most likely ID matches at each position.

The implementation limits its per-position sum to base tokens whose log probability is greater than -16. Values below that cutoff correspond to very small probability. The stored base itself has finite precision: it keeps log probabilities over the top 16 nats in a compact format. That creates a measurement floor. When the reported KL is around the scale of base quantisation, tiny differences should not be over-interpreted as exact evidence of a better or worse implementation. The source's math notes estimate a quantisation step of at most about 0.00024 nats per stored log probability, so the low end is not infinitely precise.

The same-top percentage has its own limitation: it treats a large change in the winning probability the same as an almost-tie. Consider base probabilities (0.51, 0.49) and candidate (0.99, 0.01). Both have the same winner, yet the distributions are far apart. Conversely, if base is (0.50001, 0.49999) and candidate is (0.49999, 0.50001), the winner changes even though the distributions barely moved. Reading KL beside top-token agreement gives a more complete picture.

For the book's E4B reference example, c=512 and four chunks compare against llama.cpp. The source inventory records PPL 20.919 versus 20.927, mean KLD 0.000012 and same-top agreement 99.9

Which comparison belongs to which change?

Use a small matrix of evidence before touching code. A prompt or tokenizer change needs exact token arrays and fixed conversation cases. A CPU arithmetic change should compare intermediate values to the previous reference and run a broad score check. A Metal kernel rewrite should compare the affected kernel's output and then run the model-level score gate. A KV-cache format change should include both a high-precision reference and the intended quantised mode, because an expected approximation is not the same as an accidental discrepancy.

An exactness rule prevents a speed claim from silently redefining the model. The project history requires a kernel rewrite to keep perplexity equal to six digits unless the new behavior is deliberately

PREPRINT

exposed as an opt-in mode. This is a policy for deciding when to accept a change, not a theorem that six printed digits capture every meaningful difference. If the values differ, establish whether it comes from the intended quantisation, changed reduction order, a race or an error before deciding whether the change belongs in the default path.

How do we choose a trustworthy reference?

A comparison is only as strong as its reference. The reference should be independent enough to catch the suspected mistake and similar enough to answer the same question. Rebuilding the candidate twice and seeing identical output tests repeatability, but cannot establish that the candidate is correct: the same bug can be present in both runs. Comparing CPU and Metal is more informative for a GPU kernel because the paths schedule work differently, though they still share the model loader and much of the same C code. Comparing to llama.cpp adds another implementation, but only if both programs consume the same model and score the same positions.

There is a useful hierarchy of independence. A hand-calculated toy vector checks the intended operation on a tiny case. The CPU helper checks the engine's implementation. The GPU helper checks a distinct execution path. A second engine checks more shared assumptions, but introduces its own file-format, tokenizer, and template compatibility questions. End-to-end fixed prompts check user-visible behavior, but give the least local diagnostic information. Record which level each result reached instead of describing all of them as simply "cross-checked."

A base file also has an identity. Keep a manifest beside it with the model file name and hash, evaluation text hash, reference executable, command line, context size, chunk count, KV format, and number of scored positions. If the candidate is evaluated with a different text offset or skipped prefix, then row zero in one run may not be row zero in the other. The resulting number can look plausible while comparing unrelated next-token distributions. Matching the total position count is necessary, but checking a few known token IDs at the start and end of the scored region is a helpful alignment check.

When creating the base with one engine and reading it with another, check the format contract before launching a long run. Both sides need to agree on the vocabulary size, the packed row stride, and the positions represented. The command line may use similar names while selecting different base-file modes.

A small smoke test should create a base for a short text, score that same text against itself, and verify that self-comparison is near the expected quantisation floor. Then use the real evaluation text. A self-comparison does not prove cross-engine agreement, but it checks the plumbing before an expensive experiment.

Reference	Good at finding	Does not establish
Synthetic vector	Indexing, sign, simple bounds	Correct model metadata or full forward
CPU path	GPU-only schedule or kernel error	Independence from shared loader logic
Metal path	CPU/GPU disagreement at scale	Agreement with another engine
Second engine	Shared implementation assumptions	Same tokenizer/template without alignment checks
Fixed greedy prompt	Tested visible decisions	Probability agreement at near ties

This table is not a contest to find the most independent system. It is a reminder to state the scope of a result. If the tested path shares the suspected code, add a reference that does not. If two implementations differ in file interpretation, establish alignment before interpreting numeric gaps. If there is no valid independent reference for a component, use direct examples and describe the limitation plainly.

What can a score report hide?

A summary can conceal the rows that matter. Mean KL gives one average across positions, but a small number of large outliers can matter for diagnosis. The maximum reveals an outlier, but a single malformed or badly aligned row can dominate it. Median and upper percentiles show whether the difference is broad

PREPRINT

or concentrated. Same-top agreement gives a count, but not which prompts or positions changed. Keep per-position output when investigating a failure; summary statistics are an index into the evidence, not a replacement for it.

The number of positions matters too. If two configurations report the same average over very different counts, they may have sampled different parts of the text. If one scores prompt tokens and the other scores generated tokens, the outputs are not comparable even if both call the measure "perplexity." Include the scored count and describe the boundary convention: which token is the first target, whether a BOS or template marker is included, and how the first context chunk is treated. The exact choice can be valid, but it must be shared.

PPL uncertainty is also part of the report. A tiny difference in average can be smaller than the variation across the sample positions. The engine's reporter prints a standard error for its mean-based PPL estimate and for several other statistics. That uncertainty describes sampling variation under the evaluated text; it does not account for a changed text, a different model, or an invalid base file. Do not treat overlapping error bars as proof of equality, and do not treat a smaller displayed PPL as a universal quality improvement.

The evaluation text should be representative enough to exercise the changed path. A short paragraph can prove the command runs; it cannot cover long-context cache rollover, a decode-only branch, or the end of a large vocabulary scan. For a regression gate, keep a stable short fixture for fast feedback and a longer corpus for broader review. Add targeted cases when the code change concerns a specific boundary, such as the last item in a cache ring or a prompt split exactly at a chunk edge.

The same principle applies to output comparisons. Three fixed prompts should differ in useful ways: ordinary prose, control-marker-sensitive chat, and a prompt that exercises the relevant path. If the change is in media staging, a text-only prompt does not test it. If the change is in decode attention, a prefill-only score test misses the branch. A useful fixture list says what each example covers, so future maintainers can retain the coverage when replacing the text.

How should near ties be reported?

A top-token mismatch deserves context. Save the reference and candidate top scores and the margin between the first and second choices. If their order differs while the margin is tiny, the event may be consistent with harmless rounding, but it is still a changed greedy result and should be counted. If the margin is large, inspect for a more substantial error. A margin does not excuse a mismatch; it helps distinguish a sensitive decision from a broad score failure.

For a controlled kernel change, compare the same row's full logits before reducing them to an argmax. If the implementation only exposes the selected token, add a diagnostic mode or a small test harness that saves the row. The diagnostic should keep the production path's input and arithmetic intact, changing only the observation. Recomputing logits through a different fallback after the fact may answer a different question.

Repeated output can help classify the failure. A deterministic mismatch at the same position points toward a stable difference in arithmetic, indexing, or configuration. A mismatch that moves between runs suggests a race, uninitialized memory, or input variation. Repeat with the same binary and fixed inputs before changing tolerances. Then add synchronization around the suspect lifetime as a controlled experiment. If the output becomes stable, the experiment points toward ordering; it does not yet prove the final implementation is correct until the ownership rule is fixed and the broader gates pass.

Do not round intermediate data before comparison to make a result pass. The reference log-probability file is already quantised by design, and the comparison must account for that format. An extra rounding step in a test harness can erase the discrepancy the test was intended to find. Keep raw logits where possible, preserve the base's documented precision, and state exactly where conversion occurs.

PREPRINT

When is an approximation intentional?

The 8-bit KV cache is a useful example because it deliberately stores rounded keys and values. Its comparison should answer two distinct questions: does the default compressed path behave within the documented accuracy range, and does fp16 remain a suitable reference path for an exactness-sensitive change? The outline records mean KLD around 0.0002 for the E4B 8-bit cache and around 0.00001 for fp16 under one historical c=512 setup. That gap is an accepted space/speed tradeoff in the documented configuration, not a general tolerance for unrelated changes.

A candidate that changes an fp16 attention kernel should be compared against fp16 first. If the baseline already uses 8-bit storage, an observed discrepancy combines the kernel change with cache quantisation; two effects become hard to separate. After the kernel passes its fp16 comparison, repeat with the supported default format to make sure the combination remains sound. This is the same experimental logic as changing one variable at a time.

An approximation needs a written contract: which path is approximate, what resource or speed benefit it seeks, what reference it is measured against, and what distribution of errors is acceptable based on actual project evidence. Do not set that contract by copying a threshold from a different model or context length. The E2B and E4B examples in ENGINE-MATH show that the same cache format can have different measured behavior, in part because the head grouping differs. Those observations are evidence about those runs, not a derivation of a universal rule.

If the KLD is near the base file's own quantisation floor, report that limitation instead of adding more decimal places. If a result is much larger, inspect alignment, synchronization and the model path before calling it an approximation. If the mean looks expected but one percentile is unexpectedly high, examine the individual positions. An approximation is a known property with measured consequences; unexplained error is a bug report waiting to be understood.

The real code

Where does the CPU scoring reference start?

The CPU path is useful because it shows the scoring contract in straightforward loops. It optionally applies the configured soft cap to each logit, finds the maximum and minimum, sums exponentials shifted by the maximum, and computes the negative log probability of the known next token. It also records the winning ID. Read this excerpt as one row of a score table; the surrounding code invokes it once per scored position.

```
static void score_token(void *ctx, int i, int tid) {
    (void)tid;
    const ScoreJob *job = ctx;
    const int V = job->n_vocab, tok = job->next[i];
    float *lg = job->logits + (size_t)i * V;
    float *st = job->stats + 4 * i;
    if (job->softcap > 0)
        for (int j = 0; j < V; j++) lg[j] = job->softcap * tanhf(lg[j] /
    job->softcap);

    float mx = lg[0], mn = lg[0];
    int imax = 0;
    for (int j = 1; j < V; j++) {
        if (lg[j] > mx) { mx = lg[j]; imax = j; }
        if (lg[j] < mn) mn = lg[j];
    }
    double sum = 0;
    for (int j = 0; j < V; j++) sum += expf(lg[j] - mx);
    const float lse = (float)log(sum);
    st[0] = mx + lse - lg[tok];
```

PREPRINT

```
3734
3735     if (job->save) {  // quantise log-probs over the top 16 nats, as
         llama.cpp does
3736         uint16_t *row = job->save + (size_t)i * nv_of(V);
3737         mn = mn > mx - 16 ? mn : mx - 16;
3738         const float scale = (mx - mn) / 65535.f, inv = 1 / scale,
         min_log_prob = mn - mx - lse;
3739         memcpy(row, &scale, 4);
3740         memcpy(row + 2, &min_log_prob, 4);
3741         for (int j = 0; j < V; j++) row[4 + j] = scale && lg[j] > mn ? (
         uint16_t)nearest_int(inv * (lg[j] - mn)) : 0;
3742     }
3743
3744     if (job->base) {
3745         const uint16_t *row = job->base + (size_t)i * nv_of(V);
3746         float scale, min_log_prob;
3747         memcpy(&scale, row, 4);
```

Listing 20.1: gemma4.c:3715–3747

```
3748         memcpy(&min_log_prob, row + 2, 4);
3749         row += 4;
3750         double kld = 0;
3751         int imax_base = 0;
3752         float best = 0;
3753         for (int j = 0; j < V; j++) {
3754             const float lp_base = scale * row[j] + min_log_prob;
3755             if (j == 0 || lp_base > best) { best = lp_base; imax_base =
         j; }
3756             if (lp_base > -16.f) kld += expf(lp_base) * (lp_base - (lg[j
         ] - mx - lse));
3757         }
3758         st[1] = -(scale * row[tok] + min_log_prob);
3759         st[2] = (float)kld;
3760         st[3] = imax == imax_base;
3761     }
3762 }
```

Listing 20.2: gemma4.c:3748–3762

After the negative log probability, the routine can save a compact row for another program to read. It uses a floor no lower than 16 nats beneath the maximum, stores a scale and offset, and writes a 16-bit code for each vocabulary entry above the floor. When a base row is supplied, the same routine reconstructs its approximate log probabilities, accumulates KL only where the base log probability exceeds -16, and compares the base winner with the candidate winner. This is the bridge between a single forward pass and a report over many positions.

What does the Metal twin preserve?

The GPU version distributes vocabulary entries across threads. Each thread handles strided logit entries, then the threadgroup reductions find the maximum and sum the shifted exponentials. The stored-base branch reconstructs the same approximate log probabilities and accumulates KL contributions. The following excerpt shows the setup and reduction contract, including the log-sum-exp result and known-token score. It is a focused section of the scoring kernel; the remaining save and KL branches follow the same layout.

```
1561 kernel void score(device float *logits [[buffer(0)]], device const int *
         next [[buffer(1)]],
```

```
                  device const ushort *base [[buffer(2)]], device ushort *save [[buffer(3)]],
                  device float *stats [[buffer(4)]], constant ScoreParams &p [[buffer(5)]],
                  uint row [[threadgroup_position_in_grid]], uint tid [[thread_index_in_threadgroup]],
                  uint nt [[threads_per_threadgroup]]) {
    threadgroup float tv[32];
    threadgroup int ti[32];
    device float *lg = logits + (ulong)row * p.V;

    float mx = -1e38f, mn = 1e38f;
    int imax = 0x7fffffff;
    for (int j = tid; j < p.V; j += nt) {  // soft-cap in place (each thread rereads only its own elements)
        float v = lg[j];
        if (p.softcap > 0) lg[j] = v = p.softcap * precise::tanh(v / p.softcap);
        if (v > mx) { mx = v; imax = j; }
        mn = min(mn, v);
    }
    block_argmax(mx, imax, tv, ti, tid, nt);
    mn = block_min(mn, tv, tid, nt);
    float s = 0;
    for (int j = tid; j < p.V; j += nt) s += exp(lg[j] - mx);
    const float lse = log(block_sum(s, tv, tid, nt));
    const int tok = next[row];
    threadgroup_barrier(mem_flags::mem_device);
    if (tid == 0) stats[row * 4] = mx + lse - lg[tok];

    if (p.save) {  // quantise log-probs over the top 16 nats, as llama.cpp does
        device ushort *out = save + (ulong)row * p.nv;
        const float lo = max(mn, mx - 16), scale = (mx - lo) / 65535.f, inv = 1.0f / scale;
        if (tid == 0) { ((device float *)out)[0] = scale; ((device float *)out)[1] = lo - mx - lse; }
        for (int j = tid; j < p.V; j += nt) out[4 + j] = scale != 0 && lg[j] > lo ? ushort(rint(inv * (lg[j] - lo))) : 0;
    }
    if (p.kld) {
        device const ushort *b = base + (ulong)row * p.nv;
        const float scale = ((device const float *)b)[0], lo = ((device const float *)b)[1];
        b += 4;
        float kl = 0, bmax = -1e38f;
        int bi = 0x7fffffff;
        for (int j = tid; j < p.V; j += nt) {
```

Listing 20.3: gemma4_metal.m:1561–1599

```
            const float lpb = scale * b[j] + lo;
            if (lpb > bmax) { bmax = lpb; bi = j; }
            if (lpb > -16.f) kl += exp(lpb) * (lpb - (lg[j] - mx - lse));
        }
        block_argmax(bmax, bi, tv, ti, tid, nt);
        kl = block_sum(kl, tv, tid, nt);
        if (tid == 0) {
```

```
            stats[row * 4 + 1] = -(scale * b[tok] + lo);
            stats[row * 4 + 2] = kl;
            stats[row * 4 + 3] = imax == bi;
        }
    }
```

Listing 20.4: `gemma4_metal.m:1600–1611`

The CPU and Metal paths do not use identical instruction schedules. The CPU accumulates the exponential sum in a double variable, while the MSL path performs thread-local float sums and a threadgroup reduction. That difference is why an end-to-end score comparison matters even when both paths implement the same definition. A parallel reduction changes the order of additions; floating-point addition can round differently when reassociated. The important questions are whether the difference is within the expected numeric behavior and whether it changes model decisions or grows unexpectedly across positions.

How does the report summarize rows?

The reporter gathers per-position values, sorts the KL values for percentiles, calculates summary statistics, and prints PPL, mean KL, distribution percentiles and same-top agreement. Its output is intentionally plural: mean alone can conceal a tail, and maximum alone can be dominated by one unusual row. Percentiles provide context about the spread. The excerpt below shows the report's setup and calculation flow.

```
static void report_kld(const TokStat *st, size_t n) {
    double s_nll = 0, s_nll2 = 0, s_b = 0, s_b2 = 0, s_nb = 0, s_k = 0, s_k2 = 0, s_p2 = 0, s_p4 = 0;
    size_t same = 0;
    double *klds = xmalloc(n * sizeof(double));
    for (size_t i = 0; i < n; i++) {
        s_nll += st[i].nll; s_nll2 += st[i].nll * st[i].nll;
        s_b += st[i].nll_base; s_b2 += st[i].nll_base * st[i].nll_base;
        s_nb += st[i].nll * st[i].nll_base;
        s_k += st[i].kld; s_k2 += st[i].kld * st[i].kld;
        const double p2 = st[i].p_diff * st[i].p_diff;
        s_p2 += p2; s_p4 += p2 * p2;
        same += st[i].same_top;
        klds[i] = st[i].kld;
    }
    double lp, lp_u, lb, lb_u, k, k_u, mse, mse_u;
    mean_unc(s_nll, s_nll2, n, &lp, &lp_u);
    mean_unc(s_b, s_b2, n, &lb, &lb_u);
    mean_unc(s_k, s_k2, n, &k, &k_u);
    mean_unc(s_p2, s_p4, n, &mse, &mse_u);
    const double cov = n < 10 ? 0 : (s_nb / n - (s_nll / n) * (s_b / n)) / (n - 1);
    const double ratio_u = sqrt(lp_u * lp_u + lb_u * lb_u - 2 * cov);
    const double ppl = exp(lp), ppl_b = exp(lb);

    printf("====== Perplexity statistics ======\n");
    printf("Mean PPL(Q)                   : %10.6f ± %10.6f\n", ppl, ppl * lp_u);
    printf("Mean PPL(base)                : %10.6f ± %10.6f\n", ppl_b, ppl_b * lb_u);
    printf("Cor(ln(PPL(Q)), ln(PPL(base))): %6.2f%%\n", 100.0 * cov / (lp_u * lb_u));
    printf("Mean ln(PPL(Q)/PPL(base))     : %10.6f ± %10.6f\n", lp - lb, ratio_u);
```

```
3817     printf("Mean PPL(Q)/PPL(base)         : %10.6f ± %10.6f\n", exp(lp -
         lb), exp(lp - lb) * ratio_u);
3818     printf("\n====== KL divergence statistics ======\n");
3819     qsort(klds, n, sizeof(double), cmp_double);
3820     printf("Mean    KLD: %10.6f ± %10.6f\n", k, k_u);
3821     printf("Maximum KLD: %10.6f\n", klds[n - 1]);
3822     printf("99.9%%   KLD: %10.6f\n", percentile(klds, n, 0.999));
3823     printf("99.0%%   KLD: %10.6f\n", percentile(klds, n, 0.990));
3824     printf("95.0%%   KLD: %10.6f\n", percentile(klds, n, 0.950));
3825     printf("Median  KLD: %10.6f\n", percentile(klds, n, 0.5));
3826     printf("Minimum KLD: %10.6f\n", klds[0]);
3827     const double same_p = (double)same / n;
```

Listing 20.5: gemma4.c:3789–3827

```
3828     printf("\n====== Token probability statistics ======\n");
3829     printf("RMS Δp    : %6.3f ± %5.3f %%\n", 100.0 * sqrt(mse), 100.0 *
        0.5 / sqrt(mse) * mse_u);
3830     printf("Same top p: %6.3f ± %5.3f %%\n", 100.0 * same_p, 100.0 *
        sqrt(same_p * (1 - same_p) / (n - 1)));
3831     free(klds);
```

Listing 20.6: gemma4.c:3828–3831

The helper for percentiles uses linear interpolation between adjacent entries in the sorted array. For example, a reported 95th percentile can fall between the 95th and 96th ordered values instead of selecting one item by rank. This makes a percentile a compact summary, not a literal assertion that exactly that percentage of records are below a particular observed row. Always retain the count of positions and the evaluation text with the summary.

Why test the engine's own math?

The `tools/check_math.c` utility includes the engine's C implementation and renames its `main`; it calls the real static helpers rather than copying their formulas into a separate test. This is a good way to test a routine before building a second engine around it. The header explains that the utility does not run inference, although dequantisation and range suites open a GGUF to inspect selected tensors. The `rope` and `psi` checks use synthetic values and do not open a model. That distinction lets a writing task run a genuine W0 check without loading weights.

The header and test harness provide the compile contract and report format. The excerpt shows the inclusion of `gemma4.c`, the media stubs required for compilation, and the common assertion macro. It is deliberately short; the individual tests below it generate their own synthetic arrays and call the included engine functions.

```
1  // tools/check_math.c - evaluates equations of ENGINE-MATH.md against
       the engine's own functions.
2  //
3  // It compiles gemma4.c into itself (CPU-only, main renamed), so the
       functions tested are the engine's, not copies:
4  // half_to_float, kv_put / kv_get, rope, dequant_row. It loads NO model:
        the GGUF is only mmapped and a few tensors
5  // are read. Nothing here runs the E4B or uses the GPU.
6  //
7  //   cc -O2 -DGEMMA4_MEDIA -o tools/check_math tools/check_math.c -lm -
       lpthread
8  //   (-DGEMMA4_MEDIA only because a build without it does not compile,
       see the stubs below; the media functions are stubbed)
9  //   tools/check_math [all|psi|kv8|rope|dequant|range] [model.gguf]
         (model defaults to gemma-4-E4B-it-IQ4_XS.gguf)
```

```c
//
// Each test prints "ok" or "FAIL" lines and the numbers quoted in the
    document (§5.11, §5.12, §5.13, §9).
// Exit status 1 if any equation failed.

#define main gemma4_main
#include "../gemma4.c"
#undef main

float *media_audio(const char *path, int rate, size_t *n, char *err,
    size_t err_len) { (void)path, (void)rate, (void)n; snprintf(err,
    err_len, "stub"); return NULL; }
int media_video_info(const char *path, double *duration, int *has_audio,
     char *err, size_t err_len) { (void)path, (void)duration, (void)
    has_audio; snprintf(err, err_len, "stub"); return 0; }
int media_video_frames(const char *path, double *times, int n, int
    max_side, uint8_t **rgb, int *w, int *h, char *err, size_t err_len) {
     (void)path, (void)times, (void)n, (void)max_side, (void)rgb, (void)w
    , (void)h; snprintf(err, err_len, "stub"); return 0;
}

static int failures;
#define CHECK(cond, ...) do { if (cond) printf("  ok   "); else { printf
    ("  FAIL "); failures++; } printf(__VA_ARGS__); printf("\n"); } while
     (0)

static uint64_t rs = 88172645463325252ull;
static double urand(void) { rs ^= rs << 13; rs ^= rs >> 7; rs ^= rs <<
    17; return ((rs >> 11) + 0.5) / 9007199254740992.0; }
static double nrand(void) { return sqrt(-2 * log(urand())) * cos(2 *
    M_PI * urand()); }
static float hround(float x) { return f16_table[float_to_half(x)]; }

// ------------------------------------------------------------------ §4
    .5: the 1152 offset
static void test_psi(void) {
    printf("psi: the mantissa trick of §4.5\n");
    int bad = 0;
    for (int nu = -128; nu < 128; nu++) {
        const uint8_t u = (uint8_t)nu ^ 0x80;
        const float psi = half_to_float((uint16_t)(0x6400 | u));
        if (psi != 1024.0f + u || psi - 1152.0f != (float)nu) bad++;
    }
```

Listing 20.7: `tools/check_math.c:1–40`

The suite covers several different invariants. `rope` compares the engine's iterative angle construction and the Metal-style formula with a double-precision reference across positions, rotation sizes, bases and optional frequency factors. KV tests write and read synthetic vectors through the cache encoding. Dequantisation compares decoded rows against a reference block decoder and therefore needs actual quantised tensors. A passing synthetic check establishes that sampled cases satisfy a precise local property; it does not replace tests over trained weights or the full model.

For example, a RoPE test can detect an incorrect sign or pair stride without loading a model: the generated vector is known, as are the position and frequency schedule. A cache round trip can reveal a bad block offset or scale. A full-model KL run can catch a data race that only appears when the CPU and GPU overlap. Each test asks a narrower question and has a clearer failure surface than "the answer sounds reasonable."

How should a failure be narrowed down?

First make the failure repeat with the same binary and inputs. Record the first differing token or score position. If it is in prompt construction, print the token IDs before forward. If token IDs match but scores diverge, compare the CPU and GPU outputs at the earliest affected layer that can be captured. For a numerical helper, run its direct synthetic test. For an overlap or lifetime bug, repeat with synchronization added around the suspected buffer and check whether the discrepancy disappears; that is a diagnostic, not necessarily a final fix.

Do not change several operations before rerunning the gates. A sequence of simultaneous edits can make a failing test hard to attribute. Keep a small change log: source revision, exact commands, reference file hash, first failing position, and whether the failure is deterministic. For any approximate feature, record its intended error separately from unexpected errors. The distinction is especially important for 8-bit KV storage: it intentionally rounds values, while a token-buffer race is never an allowed approximation.

Experiment: W0 and W2

W0: exercise the source's RoPE test without weights

The synthetic RoPE test is a safe W0 check. From the book root, compile the snapshot utility into the generated build area, then run only its `rope` mode:

```
mkdir -p code/build
nice -n 10 cc -O2 -DGEMMA4_MEDIA -o code/build/check_math \
  code/snapshot/tools/check_math.c -lm -lpthread
nice -n 10 code/build/check_math rope
```

This mode initializes the numerical tables and tests synthetic vectors; `main` only opens a GGUF for `all`, `dequant`, or `range`. The output reports angle errors against a double-precision reference and ends with the check status. The run made for this chapter reported CPU maximum angle error from 4.33e-05 radians at position 512 to 6.87e-03 at 131071 for sliding settings; the Metal-formula values ranged from 1.52e-05 to 2.67e-03. For global settings, the corresponding maximums ranged from 2.50e-05 to 1.53e-02 for CPU and 1.36e-05 to 5.46e-03 for the Metal formula. The tool ended with "all checks passed." These are the test's measured synthetic cases on this machine, not an inference accuracy measurement. The matching output includes cosine error and an fp16-rounding comparison; the position-dependent growth is why the test includes long positions rather than only a short toy sequence. It is a direct local arithmetic check, not a model accuracy result.

W2: compare the answer distributions

The model-level experiment needs the E4B weights and an evaluation text. Keep the same GGUF, evaluation file, context length, chunk count, and base file for both sides. First produce the base log-probabilities with the reference build, then score the candidate against that base. One outline command is:

```
nice -n 10 code/build/gemma4 -m "$MODEL" -f text.txt \
  -c 512 --chunks 4 --kv f16 --kl-divergence-base base.kld
nice -n 10 code/build/gemma4 -m "$MODEL" --kv f16 \
  --kl-divergence-base base.kld --kl-divergence
nice -n 10 code/build/gemma4 -m "$MODEL" \
  --kl-divergence-base base.kld --kl-divergence
```

The first command reports the fp16-cache PPL and saves its log-probabilities. The second checks the reference path against itself; the third compares the default 8-bit cache with that same fp16 base. The scorer reads token IDs from the base file. Retain the base with the evaluation inputs.

PREPRINT

For a cross-engine comparison, use llama.cpp's `--kl-divergence-base` output as the base and this engine's `--kl-divergence` reader, or the reverse, with compatible files.

Try it on your machine (W2): compare PPL and uncertainty, mean and percentile KLD, same-top percentage, and timing scope using fixed inputs. The c=512 values PPL 20.919 versus 20.927, mean KLD 0.000012, same-top 99.9

The final command in the outline is `tools/check_math all "$MODEL"`. It is a W2 check because `all` includes dequantisation against model tensors, even though some component checks are synthetic. Do not substitute `rope` output for the full suite. As a reader experiment, compare the model-backed output with the distribution comparison using the same inputs and recorded run conditions.

A pitfall from the project history

> **Symptom.** Decode-path perplexity suddenly showed mean KLD 0.83 instead of about 0.00001, although ordinary chat tests had passed.
> **Cause.** The CPU wrote new prompt tokens into a buffer while the GPU could still be reading the previous tokens. The larger 26B model's mixture-of-experts routing happened to make the CPU wait at every layer, masking the overlap. Dense E4B had no such wait, so a later prompt chunk, segment, or scored token could overwrite the input too early.
> **Fix.** Synchronize before the next CPU write. The README's "Bugs fixed on the way" section documents the race and the added waits. The episode shows why plausible answers are a weak gate: the short chat cases did not exercise the timing overlap that the score path exposed.

This history also gives a practical debugging lesson. A spectacularly large KL value is evidence of a broken comparison or execution path, not a subtle model quality regression. Confirm that the reference rows and candidate rows correspond to the same positions, then investigate buffer ownership and synchronization before tuning floating-point thresholds. The README documents this real incident; it does not establish that every mismatch is a race.

A second safeguard is to check that the measurement reads the intended result. The outline notes an earlier pp-test revision that compared tokens per second rather than the answer. A benchmark can be internally consistent while measuring the wrong property. Tie each reported value to a named output field and inspect that field in a small known case before trusting an automated gate.

How can the gates become a repeatable routine?

A gate is valuable when another person can run it and understand what a failure means. Give each one a stable name, a narrowly stated claim, and a small fixture. The first gate can say "the generated IDs for these three prompts match the approved list." The second can say "these positions were scored against this named base, with this output summary." The third can say "the snapshot's RoPE helper agrees with the double reference on these configured cases." Avoid vague labels such as "accuracy test" when the command only verifies that a binary starts.

Store golden outputs with provenance. A token list without its prompt, model identity and template settings is not a useful golden file. A KLD base without its source command and evaluation data cannot be regenerated. A test log should begin with the snapshot or binary version, date, arguments, and whether the test required model files or a GPU. If a known reference is intentionally refreshed after an accepted change, keep the old result in the task record so the reason for the new baseline is reviewable.

Some checks should fail hard; others should warn. A checksum mismatch in the frozen snapshot means the source being studied is not the source the book expects, so stop. A malformed or missing base file means the comparison is invalid, so stop. A changed greedy token should be recorded as a failure even if the decoded sentence looks fine. A tiny KLD difference near the base precision floor may be informative but should not trigger a precision claim by itself. A reader experiment without a local result remains unmeasured; record that distinction rather than calling it a pass or a failure.

Avoid a gate that silently picks the newest file or discovers its reference from the current directory. Resolve paths explicitly and print them. A stale base file can be more dangerous than a missing one

PREPRINT

because it allows the command to finish. If the report includes a base hash, it is possible to check that the file did not change between a reference run and candidate run. The same applies to the evaluation text and model file; large files need not be copied, but their identities should be recorded.

Run order can save time. Verify the snapshot checksum and build first. Run the small source-only math checks next. Run the fixed prompt comparison before the longer corpus. Only then run the cross-engine base-file comparison or full model-backed utility. A failure at an early stage should stop the later expensive stages, while the output preserves the exact command and error. This makes a routine useful during development without confusing partial results for a complete gate pass.

The gate script should distinguish infrastructure failure from numerical failure. A missing model file, failed Metal initialization, invalid text path, and a valid but changed score are different outcomes. Print the exit status and log location, and use distinct labels in the summary. Otherwise an automated system may treat "the test could not run" as "the test passed because no mismatch was reported." A quiet-GPU condition is necessary for meaningful timing; it is not a prerequisite for the deterministic W0 RoPE assertions.

A small worked review of a proposed rewrite

Suppose a developer replaces the decode attention score reduction with a new Metal layout. First ask what can change: the output attention vector, and therefore later logits. Keep the original kernel as a reference and feed both versions the same saved query, keys, values, masks, and cache encoding. Compare the resulting vector before considering speed. Include one short sequence with hand-readable values, one longer sequence that crosses the kernel's chunk boundary, and one case with the E4B global head width. The synthetic tests reveal indexing and reduction mistakes without needing generation.

Next use the fp16 cache configuration to compare model scores on the fixed evaluation text. Save the first row where scores diverge and compare its full logits, maximum-score margin, and the output vector if available. If only a few rows differ slightly and the measured change follows from the reduction order, then check whether greedy token IDs remain stable and whether the KLD summary stays in the established range. If the scores vary between repeated identical runs, investigate synchronization and memory ownership before discussing tolerance.

After the rewrite passes the fp16 comparison, run the default 8-bit KV path. The same query and value precision is now combined with cache quantisation, so compare it with the established 8-bit baseline rather than interpreting its difference from fp16 as a new kernel regression. Finally benchmark the candidate using Chapter 21's shared-machine procedure. This ordering keeps three claims distinct: the kernel computes the intended vector, model behavior remains within the measured reference, and the change is faster under stated conditions.

A review record for this rewrite might include a row for every stage: the synthetic vector output comparison; the full-model fp16 mean and tail KLD; greedy token agreement and first changed position; the 8-bit cache comparison; and the speed result. Each row answers a different question. If a check has not yet been run, leave its result blank and identify it as a reader experiment. This prevents a strong result in one column, such as throughput, from making an unanswered correctness question look resolved.

The example is a method, not a claim that every kernel has to pass the same arbitrary decimal threshold. The existing project policy and historical baselines inform the review, but a new path may need a new focused test. Define the expected behavior before looking at the result, and document why a deviation is acceptable when it is intentional. That gives later readers a basis for deciding whether the same conclusion still applies after a compiler or hardware change.

Exercises

1. A fixed greedy prompt produces the same decoded sentence in two builds, but the token IDs differ at one position. Give two reasons the displayed sentence might remain the same, and name the first detail you would inspect.

PREPRINT

2. At one position the reference probabilities are (0.51, 0.49), while the candidate gives (0.99, 0.01). What does same-top agreement say, and what does it fail to describe?
3. Explain why the base log-probability file, evaluation text, chunk size and model identity must travel together when reporting KLD.
4. A change to RoPE passes `check_math rope` but changes a model-level score comparison. List a useful order for narrowing the difference from prompt IDs to logits.
5. The CPU and GPU scoring paths sum exponentials in different orders. Why can their results differ slightly even if both implement the same score definition? What evidence would distinguish expected rounding from a race?
6. Create a test report for one proposed kernel change. Include its affected path, the cheapest relevant gate, one broader gate, the reference configuration, and what result would block accepting the change.

Rebuild checkpoint: gates after m1–m8

The checkpoint for this chapter is a gate script, not a new forward implementation. PLAN.md's milestones m1–m8 progress from tokenization and GGUF reading through dequantisation, CPU matvec and RMS norm, attention with a KV cache, CPU forward, Metal kernels, and chat-prefix reuse. The correctness habit applies after each: preserve a known input, run the smallest direct check, then compare the broader observable behavior for the changed milestone.

A reader can build `steps/gates.sh` with three stages. The first stage runs fixed greedy prompts and compares token IDs. The second runs the saved evaluation sequence against its reference base and checks PPL, KLD and same-top output. The third runs `tools/check_math` modes that do not require a model, followed by model-backed dequantisation when the licensed local file and approved session are available. The script should stop at the first failed command, print the exact inputs and binary identity, and preserve logs for review. It should never turn an unmeasured result into a guessed threshold.

For a small implementation, start with token equality on three fixed prompts and a source-only RoPE check. Add the complete score comparison after the forward path is available. Keep separate baselines for fp16 and 8-bit KV, because one is a reference and the other is intentionally approximate. The m7 Metal milestone should not be called complete merely because it compiles: run the relevant output comparison and the end-to-end score gate. The checkpoint is complete when every stage names what it measures, what it needs, and which failures stop the rebuild.

What belongs in the review note?

A review note should let a future reader repeat the decision without relying on memory. Start with the change in one sentence, then identify the affected code path and the expected effect. Name the baseline revision, candidate revision, model file identity, evaluation text, flags, and test count. Separate captured checks from reader experiments. For each result, state whether it is a source-only synthetic check, an engine CPU/Metal comparison, or a model comparison against an independent implementation.

When an output changes, preserve the first changed row and enough surrounding context to reproduce it. Include input token IDs, the target token, the top few candidate IDs and scores, and whether the path used a cache. If the full log is too large to keep in the report, save it under the experiment's local output directory and include a digest. A concise summary can then point back to complete evidence without copying thousands of logits into prose.

A good review note also describes what the test could not see. A short evaluation does not cover a 32K global-attention path. A synthetic RoPE check does not load the file's frequency metadata. A fixed prompt set does not cover every chat template branch. A CPU/GPU comparison can share model parsing code. These are useful limitations, not reasons to discard the evidence. They help the next test author choose the missing case.

Use the same honesty for a pass. "All tests passed" is less useful than "the snapshot checksum matched, the source-only RoPE check passed across the stated positions, and the fixed prompts matched;

PREPRINT

the W2 distribution test is a reader experiment." A pass is a statement about performed checks and inputs. It should not silently expand into a statement that every prompt and every kernel is correct.

Finally, record the disposition. The change may be accepted as exact, accepted as an explicit approximation, held for more evidence, or reverted. If it is accepted with a known numerical difference, link that difference to the intended design and its measured range. If it is held, say which experiment would resolve the uncertainty. This turns correctness review into a decision trail rather than a collection of disconnected numbers.

How do we avoid weakening a gate after a failure?

A gate is easiest to trust before the first surprising result. Write down its expected scope and acceptance rule while the old implementation is still passing. If the rule is chosen only after seeing the candidate output, there is a risk of moving the line to make the new output look acceptable. This does not mean every numeric difference must fail forever; it means the reason for an updated expectation should be explicit and supported by the intended change.

When a gate fails, first classify the failure. Is the prompt different? Did a token ID change before forward? Are the logits different but the top ID stable? Is the stored base invalid? Did the run use the wrong KV format? Does a synthetic arithmetic assertion fail? Does the result vary from run to run? These are distinct observations. A useful triage record states the failed gate, its first differing position or assertion, and the next experiment that will discriminate between likely causes.

Do not respond to a failure by loosening a tolerance across the whole suite. A tolerance can be appropriate for a known approximate format or a reference file's quantisation, but it should be local to that contract. A broad tolerance may hide a new bug in a path that was previously exact. If a specific comparison is limited by a quantised base, document that floor for the KL report; keep token equality and source-level assertions strict where the expected behavior is exact.

Likewise, do not replace a failing fixture with a convenient one without preserving what it caught. If a test was too long or depended on unstable external data, reduce it while keeping the boundary condition that exposed the defect. If the fixture itself is wrong, repair it and document the correction. Keep the original failure output with the review record so the reason for changing the fixture remains visible.

A changed gate should be reviewed as code. The update should say which path changed, why the previous assertion no longer describes the intended behavior, what new assertion replaces it, and which unaffected gates still pass. This matters especially when a feature becomes opt-in. The default path should keep its old contract, while the opt-in path receives its own measured reference and clear command-line selection.

What does a useful regression fixture look like?

A small fixture is selected for the question it answers. For tokenization, include whitespace and a control marker boundary. For chunked prefill, include a prompt just below and just above the chunk size. For a sliding KV ring, include positions before and after the oldest retained entry is overwritten. For a decode attention rewrite, include short and long key sequences and both E4B head widths. Such cases are compact enough to rerun and specific enough to make an indexing mistake visible.

A numeric fixture should make the input inspectable. For an arithmetic helper, use a short vector whose components are distinct and whose expected output can be calculated independently. Repeated values can hide an accidental swap. Include at least one zero or boundary value when the operation has a special case, and include nontrivial scales when the code decodes quantised data. Randomized cases are useful for breadth, but a deterministic seed and a small hand-checkable case make failures easier to reproduce.

For a concurrency-sensitive path, the fixture must exercise the ordering that matters. A one-token call may never reuse a buffer while the GPU is still reading it. Use a sequence of chunks or overlapping operations that reproduces the production lifetime. A stress test can make a race appear more often, but a

PREPRINT

passing stress run is not proof that the race is absent. The correctness argument should also identify who owns the buffer, when the GPU's read completes, and where the CPU is allowed to write again.

For model-level evaluation, retain the exact text bytes and offsets, not merely a public dataset name. Text processing can change through newline normalization, tokenizer behavior, or different chunk boundaries. Record the source and license of the local evaluation material as the project requires. If the dataset cannot be distributed, keep a local hash and a short non-sensitive description so the run can still be identified without copying data into the book.

A regression fixture set should remain small enough that developers actually run it. One targeted case can execute on every edit; the full reference comparison can run at a deliberate checkpoint. Keep the division explicit. If the quick check becomes so large that it is routinely skipped, it no longer provides the early warning it was designed to give. If the long check is never scheduled, the project should not claim broad model-level validation.

How should results be compared across time?

A historical baseline is most useful when its conditions travel with it. Store the date, machine, operating-system and compiler context, binary identity, model format, evaluation file, command, and output summary. For deterministic token and math checks, the machine may not change the expected result; still record it so a compiler or architecture transition can be distinguished from a source edit. For performance measurements, the machine and system load are essential and Chapter 21 treats them separately.

When a compiler changes, rerun the direct arithmetic checks and the score comparison even if the source is untouched. Different optimization choices can alter reduction order or expose undefined behavior. When the model file changes, make a new base and do not compare its KLD directly with a previous file's report. When only the evaluation text changes, label the corpus version and treat its PPL as a new series. The purpose is not to make every report cumbersome; it is to keep unlike measurements from being mistaken for a regression.

A small results table can make this discipline routine:

Field	Example	Why keep it
Change	decode score reduction	Connects evidence to source scope
Reference	fp16 cache, named base	Identifies expected behavior
Coverage	4 chunks, 2,048 positions	Defines what was tested
Outcome	IDs equal; KL summary	States observation, not impression
Limit	no 32K capture	Prevents widening the claim
Disposition	held for long-context run	Makes next action clear

The examples are report fields, not measured results for this chapter's W2 experiment. Its model-backed comparison is a reader experiment. This distinction is central to a reliable record: a blank result is not a zero, a previous report is not a fresh run, and a sample number does not become a threshold just because it is easy to copy.

What has to be established before calling it right?

No single number certifies a language model engine. Matching tokens on fixed prompts says the tested greedy decisions agree. PPL says how the candidate scores a known sequence on average. KL says how its distributions differ from a saved reference, subject to the base file's precision and cutoff. Same-top agreement says how often the preferred ID is unchanged. Direct math tests check reusable operations on inputs we can inspect. Together, they form a practical ladder from quick feedback to broad evidence.

The right answer to "did this optimisation preserve behavior?" is therefore a small bundle: identify the behavior the change can affect; run a local check that isolates it; run the score comparison that reaches the full model path; and report the exact reference and remaining uncertainty. If the check fails, keep the failure visible and trace it. If the check passes, state the scope it covers. That is how a speed project keeps its answer attached to the model rather than to a plausible-looking sentence.

PREPRINT

Chapter 21

Measuring speed on a shared machine

The question

How do we trust a speed number when the GPU is shared and a fanless machine changes pace as it warms?

Reading map: 14 pages

The section budgets follow the outline's estimate. Each section gives a practical part of one measurement protocol: decide what is being measured, collect comparable runs, separate clocks, and state the limits of the result.

Section	Pages
The experiment is a time series, not a contest	2
Interleaving, percentiles, and the idle gate	3
Wall time, GPU time, and command-buffer timing	2
A bandwidth floor and bytes actually streamed	2
Source tools: A/B, quiet gate, memory, and kbench	2
W0/W1 checks and W2/W3 reader experiments	1
Pitfall, exercises, and measurement checkpoint	2
Total	**14**

The idea

Why is one run not a result?

A benchmark result is a short time series produced by a particular binary, workload, machine state, and clock. It is not a permanent speed label attached to the source code. On this shared Mac, other applications can use the GPU and memory while a benchmark runs. The fanless Air can also slow after a long prefill. Both effects are visible in the project's own history: the README's decode measurements ranged from 19.5 and 20.8 tokens/s in an alternation to 29.9 when run alone, and it records a 32K prefill followed by 12.5 tokens/s, rising to 20.2 after a two-minute pause. Those are historical observations from the README's 2026-10-03/04 measurement notes, not this chapter's fresh measurements.

The same binary can therefore produce several honest numbers. "29.9 tokens/s" answers what happened during one isolated run. "20.8 tokens/s" answers what happened in one busy alternating round. Neither describes every later run, and the faster result is not automatically more representative. The central task is to make the conditions visible and to compare candidates under the same sequence of conditions.

A useful measurement has four parts. First, say what work ran: prefill, decode at a stated context depth, a named kernel, or full end-to-end generation. Second, say what executable, model, prompt or evaluation text, and options were used. Third, show the machine state and run order. Fourth, state which

PREPRINT

summary of the repeated observations you report. If any one of these is absent, a reader may mistake unlike experiments for a speedup.

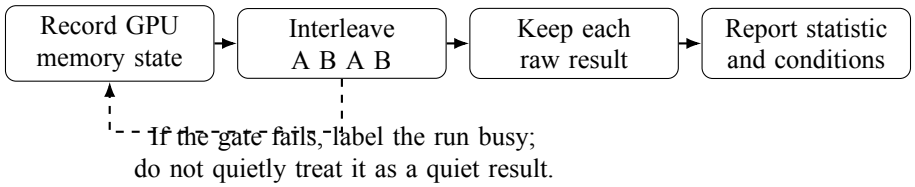


Figure 21.1: A performance result records its conditions and run order.

What question does the benchmark answer?

The word "speed" hides several different questions. Prefill speed measures how quickly the model consumes a prompt containing many known tokens. Decode speed measures how quickly it generates the next tokens one by one. A long-context decode includes attention over a longer history than a short-context decode. A kernel benchmark isolates a smaller operation and often uses synthetic inputs. End-to-end chat includes setup and application behavior beyond the model's matmuls. These results should not be combined into one leaderboard.

The engine's benchmark accepts a context depth, prompt length, generated-token count, and repetition count. It reports the prefill of the context separately, then reports prompt prefill, time to first token, and decode for synchronous and pipelined modes. The depth matters: an E4B decode at 33K asks attention to inspect much more cached history than a decode near the start of a conversation. The 8 query heads and 2 KV heads shape that work, but the reported throughput is still the full path selected by the command, not a measurement of one attention kernel.

Before collecting timings, write a one-line question such as: "Does candidate B change E4B synchronous decode at depth 512 under the same 64-token prompt and 128-token generation?" That sentence fixes which numbers belong in the comparison. If the intended change affects only prefill, a decode-only result cannot establish its benefit. If it changes the KV representation, test the same context depths and cache mode in both builds. Keep the question stable while gathering the paired runs.

Why interleave A and B?

Suppose build A is tested in the morning and B after a long prefill, with other work starting in between. The difference combines the code change with time, temperature, and background load. Running A several times and then B several times has the same weakness: a drift over the test window can look like a candidate effect.

Instead use a repeated order such as A, B, A, B, A, B. Each B is near an A in time. Compare paired differences or ratios as well as the raw values. If both builds slow down together, shared conditions are a likely contributor. If B remains faster within most neighboring pairs, that is stronger evidence than comparing the first A with the last B. Reverse the starting order in a later round when two distinct candidates are being compared, so one candidate does not always benefit from going first.

A/A is useful even when there is no candidate. Run the same binary in both slots and inspect the spread. This estimates the noise floor of the entire procedure: process startup, shared GPU activity, heat, and the chosen workload. It does not prove that a future A/B difference is caused by the code. It tells us how large a difference the method can resolve on that occasion. If A/A varies by ten percent, a measured three-percent A/B advantage is not established by that run set.

Keep each raw observation. Do not store only the best time, and do not discard a slow value merely because it is inconvenient. A timeout, failed gate, or known competing workload can be tagged and explained, but the decision to exclude it should be explicit. Saving the exact command and output makes it possible to distinguish a real repeat from a copied summary.

PREPRINT

Which statistic should we report?

Repeated runs form a distribution. The median describes the middle observation and resists one unusually high or low result. The 10th percentile describes a slower portion of the run set; for throughput, it is a conservative view of what the system delivered. State the percentile convention, especially for a small sample. The nearest-rank convention selects the item whose one-based rank is the smallest whole number at least one tenth of the sample count. With ten sorted observations, the nearest-rank p10 is the first value; the median is the mean of observations five and six.

For example, a fixed synthetic list of ten through one hundred in steps of ten has p10 = 10 and median = 55 under that convention. This toy set checks the arithmetic only. It says nothing about the engine's performance. The W0 check later in the chapter reproduces it without loading a model.

The snapshot's `kbench` source reports best and median across timed passes; it does not compute p10. The engine's end-to-end `bench` reports its own summary and standard deviation. If a report needs p10, preserve the individual observations and compute it from those rows, rather than relabeling `best`, `mean`, or `median` as p10. With only three observations, a p10 estimate is especially coarse: the nearest-rank result is simply the minimum. More repetitions help describe the tail, but they also consume shared-machine time. Use enough to answer the question and record the count.

Throughput and latency also reverse which tail is desirable. For tokens/s, a low percentile is slower and conservative. For milliseconds/token, a high percentile is slower and conservative. Name both the statistic and the unit: "decode p10, 24.1 tokens/s" or "decode p90, 41.8 ms/token." Never say only "p10 = 24.1" without saying what was sorted.

How many observations make a percentile useful?

A percentile depends on how many observations were collected. With ten sorted runs, the nearest-rank p10 is the minimum. With twenty runs it is the second-smallest; with thirty it is the third-smallest. A p10 from ten runs is therefore a compact conservative label, not a detailed map of the slow tail. State the count beside the percentile so readers can see how much data supports it.

Runs	Nearest-rank p10	What it selects
10	rank 1	slowest observed throughput
20	rank 2	second-slowest observed throughput
30	rank 3	third-slowest observed throughput

The table is a property of the chosen percentile convention, not a timing result. If the goal is a typical result, the median gives a stable center. If the goal is a conservative throughput number, the low tail gives a useful caution. Reporting both lets a reader see whether the center and the slower cases tell the same story. A wide gap between them suggests the run conditions are variable, and should trigger a look at the per-run state before any code change is credited.

Do not treat a percentile as a confidence interval. It summarizes the observations you have; it does not estimate every future run or establish the cause of a difference. Keep A and B's values separate, and preferably preserve the adjacent pairs. If a candidate has a better median but a worse p10, the result may still matter for an interactive use case, but the trade-off should be visible rather than collapsed into one "winner."

For small candidate changes, the pattern across pairs is often easier to interpret than an extra decimal place. If every B is only slightly faster than its neighboring A, that is more consistent than a single unusually fast B. If the signs alternate, or both labels drift together, collect a new controlled set or say that the current run cannot resolve the change. The raw rows let a later reader apply another robust summary without repeating an expensive experiment.

What does an idle gate establish?

The snapshot's `quiet.sh` samples the GPU utilization value from `ioreg` every five seconds. It proceeds after three consecutive readings below 15 percent. If it fails to find that sequence after about twenty

PREPRINT

minutes, it warns that the GPU is busy and still runs the command. This makes it a useful waiting helper, but its warning is not a certificate that the benchmark is quiet. A caller must inspect the message and decide whether the run belongs in a quiet set.

The measurement plan in the project sets a stricter publication gate: record the samples and accept a quiet capture only when the mean GPU utilization is below 8 percent and the maximum is below 20 percent. The existing helper's three-below-15 rule is a coarse start condition, not that full summary. A trace that misses the stricter rule may still be valuable as a contended measurement, provided it is labelled as such. Do not rename it "quiet" because the command completed.

GPU utilization is not the whole machine state. Record memory pressure and other concurrent work as well. On a shared machine, the benchmark may be affected by memory compression, an encoder, speech activity, or browser GPU work, even if a single idle sample looks low. The `nice -n 10` priority keeps the benchmark polite to ordinary work; it does not make a run controlled, and it cannot guarantee the voice has no latency. If the system becomes busy or the voice becomes delayed, stop adding measurement work and retain the run only as a busy observation.

A gate should be logged per run, not recalled later. A compact record could include the three or more utilization samples immediately before the command, the mean and maximum over the timed interval, memory pressure before and after, and a short note such as "browser active." If a tool only gates before launch, do not imply it monitored the whole benchmark. The simplest honest label is "idle gate passed before run; utilization during run not sampled."

Why keep wall time and GPU time separate?

Wall time answers how long the caller waited. GPU time answers how long the GPU reported itself busy in submitted command buffers. The engine exposes both for decode when `GEMMA4_TIMING` is enabled: it measures elapsed wall time around decode and separately subtracts the accumulated GPU busy counter before and after. It also reports waits for expert-file reads and router work. These numbers partition parts of the path; they are not alternate spellings of one clock.

A prompt benchmark may encode work on the CPU while the GPU is running earlier commands. Some host setup can overlap GPU execution. Thus wall duration can be greater than GPU-busy duration because it includes queue gaps or CPU work, or the two can be close when the GPU dominates. On another path they need not add cleanly: overlapping work means adding durations can double-count time. Report each with its label and measurement boundary. Do not divide a wall-clock token count by GPU time and call the result end-to-end throughput.

What should happen to the warm-up and cooldown runs?

A first run may include work that later runs avoid. The engine compiles specialized Metal kernels during its warm-up; the benchmark deliberately executes a warm-up repetition before it records timed repetitions. That makes its repeated decode result different from a cold-start measure. If the question is "how fast is steady decode after startup?", the warm-up should not be counted. If the question is "how long until the first useful answer after launching the process?", startup and compilation are part of the answer and should be measured separately.

The two questions need different clocks and different ledgers. For steady-state speed, record the first measured repetition only after the benchmark's warm-up path and state that warm-up was excluded. For first-answer latency, launch a fresh process under a fixed startup condition and include model mapping, compilation, prompt processing, and the first token in the wall interval. Do not compare one build's cold first token with another build's warmed decode rate. The engine's `bench` path records time-to-first-token apart from later token throughput, which makes that distinction concrete.

Heat introduces a second warm-up state: the machine itself. A long prefill can keep the fanless Air busy for minutes. The README reports that its 32K prefill left decode at 12.5 tokens/s; after a two-minute pause it measured 20.2. The engine includes `GEMMA4_BENCH_COOL=seconds` for depth benchmarks for this reason. These are historical values from the README's long-context notes, not a result

PREPRINT

for the current session. When the benchmark includes a heavy prefill, apply the same cooldown between comparable candidates or report the hot and cooled phases as separate cases.

A cooldown is part of the test design, not a knob to turn until the desired number appears. If each run gets a different pause, the comparison has changed the thermal condition along with the binary. Record the prefill size, pause duration, and order. A/B alternation may still retain heat from the previous run; a short idle period between every candidate may make the procedure too slow and change the state differently. Choose a repeatable sequence, then use the A/A spread to see whether it is stable enough for the intended claim.

The benchmark's synchronous and pipelined decode modes also answer separate questions. Synchronous mode waits for each token before queuing the next step. Pipelined mode submits the next step before the CPU reads the previous token. The engine reports both and checks whether they generated the same IDs. The faster rate is not necessarily the right product number: interactive callbacks, stop handling, or the application loop may make the synchronous behavior more representative. Match the mode to the caller whose speed you are trying to describe.

The snapshot's `kbench` repeats a kernel operation many times, warms it once, and synchronizes after a batch. Its source computes a best and median pass and labels the result as GPU time per operation. The important methodological point is that repeated work is submitted together and the wait occurs at the batch boundary. A profile that inserts a commit and wait after each tiny dependent kernel changes the schedule being measured. The project documented isolated profiling values of 190–240 microseconds for small steps, while the normal dispatch floor was around 8.5 microseconds; the wait-heavy profile was measuring a round trip, not ordinary isolated kernel cost. These are historical values from ENGINE-MATH §11.7 (2026-10-06), not fresh timings here. Chapter 6 discusses that profiling trap in its kernel context.

PREPRINT

How much speed is physically plausible?

For a decode step dominated by reading weights, a bandwidth estimate gives a floor on time. Use the bytes that the chosen path actually reads, not the total GGUF file size. E4B's historical `IQ4_XS` configuration reads about 2.61 GB of weight data per decode token; the machine's estimated peak bandwidth is about 102 GB/s. The division gives a best-case floor near 25.6 milliseconds for that weight stream alone. This is a scale check, not a promise: the peak is an estimate, other work consumes time, and attention also reads cached keys and values.

$$\text{time} \geq \text{bytes}/\text{GB}/\text{s}$$

For the documented values, 2.61 GB ÷ 102 GB/s = 0.0256 seconds. Since one second contains 1,000 milliseconds, that is 25.6 ms of ideal weight traffic per token. The four quantities are the streamed bytes, the estimated rate, the resulting seconds, and milliseconds per second. Every term must describe compatible units and the same model configuration.

A floor is useful because it can reject impossible explanations. If an alleged one-token time were 10 ms while the required weight stream alone needs 25.6 ms at the estimated peak, then either the byte count, units, or benchmark description is wrong. If measured time is much larger than the floor, that does not tell us immediately whether attention, dispatch, CPU work, contention, or a lower achieved bandwidth caused the gap. The floor sets a bound; a kernel timing and an end-to-end timing help locate the rest.

At longer context, bytes are not constant. Attention reads the KV history, so a decode at 33K has more memory traffic than a decode at depth zero. The reported 2.61 GB of weight bytes is not the full traffic at that depth. That is why the same context, cache format, and generated-token count must be held fixed in A/B runs. A token/s result without a depth cannot be compared to another decode result with a different history.

What can the floor tell us about a bottleneck?

The bandwidth floor is most useful when paired with measurements at different scales. A full decode rate includes matvecs, attention, the LM head, dispatch gaps, and CPU work. A `kbench` result isolates selected matvecs or attention shapes. If the end-to-end rate is far below the weight-stream floor, first check whether the byte count and timing boundary match; the floor is a lower bound for ideal streaming, so a measured time below it signals inconsistent assumptions. If end-to-end time is above the floor, the difference is a place to investigate, not a single named cost.

Compare only compatible quantities. A GB/s value derived from weight bytes and a per-token wall time is an effective rate for that stream, not necessarily DRAM bandwidth measured by a hardware counter. It excludes some traffic and includes time spent on other work. `kbench` prints the weight bytes assigned to each selected operation and divides by its per-operation time. That is useful for comparing the same operation across candidates. It should not be placed on the same axis as end-to-end tokens per second without explaining the units and boundary.

A kernel can approach the streaming floor while the complete model still has meaningful overhead. In a decode layer, operations depend on earlier results: the next projection cannot consume an activation until its producer has completed. Repeating a tiny kernel after a forced wait can make each dependency look much slower than its normal batched path. Conversely, averaging all work into one token time hides which operation moved. Use an end-to-end benchmark to establish the user's observed speed, then an appropriately batched kernel tool to examine a suspected component.

The chapter 8 prefill example illustrates another boundary. The historical README reports an end-to-end E4B prefill comparison near 439–440 tokens/s against llama.cpp's 264–270, while the same engine's matmul calculation is a separate, estimated compute rate. The prefill total includes attention, scheduling, and row handling; it cannot be assigned wholly to the matrix kernel. To test a kernel-specific change, compare the relevant isolated operation and preserve an end-to-end check so the local win is not mistaken for a whole-engine gain.

Which bytes belong in a bandwidth comparison?

The numerator must describe the data read by the timed operation. A GGUF's file size includes metadata and tensors that a particular GPU path may never read. The engine's E4B load leaves its large per-layer token table on the CPU, and shared layers do not need every stored K and V projection. Dividing the whole file size by a token time would therefore count bytes that this operation did not stream. Conversely, a decode at long context also reads the KV history, so a weight-only numerator leaves out traffic that really occurred.

Quantization changes the count too. The snapshot's README records about 2.61 GB of weight bytes per E4B decode token for the IQ4_XS configuration and about 2.74 GB for Q4_0. These are historical file and execution figures, not new measurements. They describe different model files, so a throughput comparison between them includes both the byte-count change and the numerical effect of quantization. If the question is whether a kernel rewrite is faster, keep the GGUF identical. If the question is whether a quantized file improves the product trade-off, measure its output agreement and speed as separate results.

For a fused gate-and-up kernel, the selected operation reads two matrices. The `kbench` source counts the tensor's bytes twice when its second weight matrix is present. That accounting is appropriate for the kernel-level effective bandwidth of that operation, but it is not the total bytes read by a full model token. Read the operation name and shape beside the rate; a GB/s number without them loses its meaning.

Quantity	Includes	Use it for
GGUF file size	Header, unused tensors, and active payloads	Identifying the file; not a streaming numerator
Kernel byte count	The selected matrix or matrices	Comparing the same isolated operation
End-to-end traffic	Weights, KV reads, activations and other work	Explaining a complete decode path

PREPRINT

Memory residency and memory traffic are different, too. A tensor can stay wired for repeated GPU access without being read in full on every token. The byte count asks what this operation consumes from the memory hierarchy; the system-memory measure asks how much memory the run caused the machine to use. A high wired total does not equal bytes transferred per token, and a high process footprint does not tell us which tensors the GPU read. Keep each resource under its own heading in the report.

Finally, `kbench` repeats the same selected operation fifty times per pass. This reduces the effect of one launch and synchronization boundary, but it is still a microbenchmark with synthetic inputs. It can reveal whether a change helps that kernel shape. It cannot show whether end-to-end decode improves by the same proportion: other operations, dependencies, attention at the chosen depth, and application work remain in the full path. The strongest speed report pairs one focused measurement with the end-to-end result and explains where their boundaries differ.

What exactly should the record contain?

Use a small run ledger. One row represents one timed repetition, not one command that hides several values. Attach the model identity and evaluation-text hash in the report header so the row stays compact.

Field	Record
Question	Operation, context depth, prompt length, generated tokens
Build	Snapshot/binary identity, compiler, relevant options
Input	Model identity, text identity, tokenizer and cache mode
Order	Candidate label, round number, A/B position
Conditions	Date, machine, GPU samples, memory pressure, other work
Timing	Raw wall time, GPU busy time if available, throughput unit
Summary	Sample count, p10 convention, median, exclusions with reasons

Do not use an attractive rounded number as the raw record. Preserve adequate precision in the log, then round in the prose to match the measurement quality. A run that prints 24.7 tokens/s does not justify four significant digits. If a model file cannot be distributed, its filename and local hash identify the input without copying it into the book. The same applies to evaluation text; the book's code snapshot currently has no evaluation text, so a future capture must use the approved local text and record its identity.

When is a small speedup not established?

The plan records that this Mac's single-GPU timings have varied by roughly 15–30 percent across runs, while a proposed optimization may be only about three percent. When the run-to-run spread is that large, a single three-percent difference cannot distinguish the candidate from the measurement noise. More digits do not solve that problem. The right conclusion is "not resolved under these conditions," not "the candidate is exactly three percent faster."

A better experiment first reduces avoidable variation: keep the exact workload fixed, wait for the documented GPU gate, note memory pressure, interleave the candidates, and compare adjacent pairs. If the A/A spread remains wider than the effect, move the capture to a quieter session or a machine whose conditions can be controlled more tightly. Repeat the paired comparison, retain all observations, and state whether the spread narrowed. If it did not, the claim remains unproven on this setup. A negative result can still guide engineering: it may be sensible to avoid a riskier kernel rewrite when the predicted gain is below the system's resolution.

This does not mean every change needs a laboratory. A large improvement that appears consistently across interleaved pairs can be useful evidence even on a shared machine, especially when the correctness gates also pass. The strength of the language should match the evidence: "the observed median rose in this set" is a direct statement; "the change is faster" needs repeated, comparable runs; "the change is three percent faster on this machine" needs a distribution narrow enough to support that estimate.

Keep accuracy and speed conclusions separate. A timing probe that skips a kernel can reveal the time associated with that path, but its output is invalid and says nothing about a shippable model. A candidate that is faster but changes token IDs or the score distribution must return to Chapter 20's gates.

PREPRINT

A candidate that passes correctness still needs the controlled speed comparison. Neither result substitutes for the other.

The real code

How does the A/B helper alternate?

The helper below nests candidate specifications inside rounds. The inner loop visits each passed candidate in the supplied order, so callers can pass A and B and get A, B, A, B. It captures the engine output lines for each run and prints a round label. It does not calculate paired statistics, p10, or a GPU gate result; those remain part of the report procedure.

```
# interleaved A/B decode timing: tools/ab.sh ROUNDS "label:binary:env" ...   (prints decode tok/s per run)
cd "$(dirname "$0")/.."; R=$1; shift
M=gemma-4-E4B-it-IQ4_XS.gguf
for r in $(seq 1 $R); do for spec in "$@"; do
  IFS=: read -r label bin envs <<<"$spec"
  out=$(env $envs ./$bin -m $M -f ../gemma4-chat/eval_text.txt --bench 64 ${TG:-128} -r 1 2>&1 | grep -iE "decode|tokens/s" | tr '\n' ' ')
  echo "r$r $label: $out"
done; done
```

Listing 21.1: `tools/ab.sh`:2–9

There is an important practical limit in this snapshot. The script hard-codes the model filename and an evaluation-text path outside the snapshot. That path reflects the engine repository's historical layout. A future writer must adapt a local copy to the approved model and text locations before using it. This book does not run the helper because its command launches model inference. A/A is the first useful run set: pass the same binary under both labels and retain the sequence, then use the same procedure for the candidate. The output is only per-run text; it must not be mistaken for a statistical summary.

What does the quiet helper actually gate?

The next excerpt makes the threshold visible. Three consecutive samples below 15 are required; after the loop the helper runs the command either way. The timeout and warning are important parts of its behavior: no quiet interval means "busy, numbers unreliable," not "quiet enough after all."

```
# wait (max ~20 min) until the GPU is idle (<15% utilisation, 3 samples in a row), then run the command
ok=0
for i in $(seq 1 240); do
  u=$(ioreg -r -d 1 -w0 -c IOAccelerator | grep -o '"Device Utilization %"=[0-9]*' | head -1 | cut -d= -f2)
  if [ "${u:-100}" -lt 15 ]; then ok=$((ok+1)); else ok=0; fi
  [ $ok -ge 3 ] && break
  sleep 5
done
[ $ok -ge 3 ] || echo "WARNING: GPU busy (${u}%), numbers unreliable" >&2
exec "$@"
```

Listing 21.2: `tools/quiet.sh`:2–11

Treat this as a pre-run wait, then save a separate utilization trace if the publication gate requires its mean and maximum. The helper does not inspect memory pressure, does not save the sample series, and does not refuse execution when it times out. Readers should understand those boundaries before using the command as evidence.

Why does the memory tool inspect the whole system?

A process's resident set can omit file-backed pages made resident and wired for GPU use. The snapshot's helper instead takes a baseline of system-wide wired, anonymous, and compressed memory, samples the same total while the command runs, and reports the peak rise. This is noisier if another application changes memory at the same time, but it includes the category the process-only view misses.

```
# memwatch.sh CMD ARGS...: runs the command and reports the most memory the system used for it: the peak rise
# of wired + anonymous + compressed pages (what Activity Monitor calls "Memory Used", minus cached files) over
# the level before it started, sampled every 0.2 s. GPU-wired pages of a mapped model file count; the page
# cache does not. Other processes add noise, so run it on an otherwise quiet machine.
used() {
    vm_stat | awk '/page size of/ {ps = $8}
        /Pages wired down/ {w = $4} /Anonymous pages/ {a = $3} /Pages occupied by compressor/ {c = $5}
        END {gsub("\\.", "", w); gsub("\\.", "", a); gsub("\\.", "", c);
     printf "%.0f\n", (w + a + c) * ps / 1e6}'
}
base=$(used)
"$@" &
pid=$!
peak=$base
while kill -0 $pid 2>/dev/null; do
    u=$(used)
    [ "$u" -gt "$peak" ] && peak=$u
    sleep 0.2
done
wait $pid
echo "memwatch: peak memory used $(( (peak - base) )) MB above the $base MB before the start" >&2
```

Listing 21.3: `tools/memwatch.sh:2–21`

The code measures memory, not speed. It is included because memory pressure is a condition that can alter a benchmark and because the method illustrates a broader rule: choose a system measure that can see the resource the workload actually uses. The helper's own header explicitly warns that other processes add noise.

How does kbench avoid a wait after every call?

The kernel benchmark's opening comments describe its measurement unit and warm-up. It issues fifty repetitions in a pass, synchronizes after the batch, and sorts timed pass durations. With `KB_PASSES` set above one, it prints best and median. It does not produce the chapter's p10 statistic; keep that distinction in the experiment report.

```
// kbench: GPU time of the decode matvecs of a model, each run REPS times back to back in one command buffer.
//   cc -O3 -DGEMMA4_METAL -o tools/kbench tools/kbench.c gemma4_metal.m -framework Metal -framework Foundation
//   tools/kbench model.gguf
#define main gemma4_main
#include "../gemma4.c"
#undef main

enum { REPS = 50 };
```

```
9
10 static int cmp_d(const void *a, const void *b) { const double x = *(
       const double *)a, y = *(const double *)b; return x < y ? -1 : x > y;
       }
11
12 // KB_PASSES (default 1) timed passes after the warm-up; with several,
       the best and the median are printed (the best
13 // is the figure to trust on a machine that is not quiet)
```

Listing 21.4: `tools/kbench.c:1–13`

```
30         if (pass) dts[pass - 1] = (now() - t0) / REPS;
31     }
32     qsort(dts, passes, sizeof dts[0], cmp_d);
33     const double bytes = (double)tensor_nbytes(w) * (w2 ? 2 : 1), best =
       dts[0], med = dts[passes / 2];
34     printf("%-34s K%-6d N%-7d %7.1f us  %6.1f GB/s   (median %7.1f us
       %6.1f GB/s)\n", what, K, N, 1e6 * best, bytes / best / 1e9, 1e6 * med
       , bytes / med / 1e9);
```

Listing 21.5: `tools/kbench.c:30–34`

End-to-end engine timing answers a different question. The excerpt below brackets each decode pass with a wall clock and snapshots the completed GPU-busy counter. The final report for the benchmark computes throughput from the collected elapsed times. When enabled, `GEMMA4_TIMING` prints wall milliseconds per token and GPU-busy milliseconds per token side by side, along with two wait categories. Retain the labels rather than treating one as a correction factor for the other.

```
3981             const double t0 = now();
3982             t_wait_ssd = t_wait_gpu = 0;
3983             const double busy0 = gpu_time();
3984             decode(m, &acts, &st, first, N, sync, logits, gen[sync],
       NULL, NULL);
3985             if (rep >= 0) tg[sync][rep] = N / (now() - t0);
3986             if (getenv("GEMMA4_TIMING"))
3987                 fprintf(stderr, "decode %.1f ms/token: GPU busy %.1f ms,
        waiting for expert reads %.1f ms, for the router %.1f ms\n",
3988                         1e3 * (now() - t0) / N, 1e3 * (gpu_time() -
       busy0) / N, 1e3 * t_wait_ssd / N, 1e3 * t_wait_gpu / N);
```

Listing 21.6: `gemma4.c:3981–3988`

Experiment

W0: can the percentile summary be checked without a model?

Use a fixed series so the expected result is hand-checkable. This run is CPU-only and needs no model or GPU. The nearest-rank p10 selects the smallest of ten observations; the median averages the fifth and sixth. The command and output below were run on 2026-10-22 with Python 3.9.6. The result is an arithmetic check, not an engine benchmark.

Experiment W0

`python3 code/examples/ch21-w0-percentiles.py`
Output: `synthetic runs=10 nearest-rank p10=10 median=55.` Captured in `code/expected/ch21-w0.txt`. The values are intentionally synthetic and have no performance units. The test confirms the stated convention only; it does not test utilization sampling, timing, or GPU behavior.

W1: what can be learned from a header-only read?

The small standard-library script reads exactly 24 bytes: the GGUF magic, version, tensor count, and metadata-entry count. It does not parse metadata values or read tensor payloads. The following command and output were captured on 2026-10-22 from the local E4B GGUF. This is a file-identification check, not a performance result.

Experiment W1

`python3 code/examples/ch21-gguf-header.py $MODEL`
Output: `GGUF version=3 tensors=720 metadata_entries=56.` Captured in `code/expected/ch21-w1.txt`. The script is deliberately narrower than the snapshot's tensor comparison tool: that tool hashes payload bytes, which this writing task does not need. For a later benchmark, record the model file's identity and use the same file for every candidate.

W2: how should readers compare runs?

This reader experiment launches model inference and requires a licensed local model plus a fixed evaluation text. First build from the snapshot with `nice -n 10 sh code/snapshot/build.sh`; it writes `code/build/gemma4`. Set `MODEL` and `TEXT` to those local inputs. Run one command at a time at low priority, retain every raw row, and alternate the same binary under labels A and A2 to measure the procedure's spread. Then repeat with the candidate in B. Gate each run and record utilization during the timed interval, memory pressure, date, and machine. The snapshot's `tools/ab.sh` demonstrates the order but must first be adapted locally because its current paths are hard-coded.

Experiment W2

```
for label in A A2 A A2 A A2 A A2 A A2; do echo "$label"; nice -n 10
code/snapshot/tools/quiet.sh code/build/gemma4 -m "$MODEL" -f "$TEXT"
--bench 64 128 -r 1; done
```

This grouped sequence is only a first A/A spread check. For a candidate comparison, use an explicit alternating order such as A, B, A, B, A, B, A, B, A, B, with one run per process and the same inputs and options. Extract each raw throughput and timing value, calculate p10 and median by the declared convention, and compare adjacent pairs. Record both wall and GPU timing if enabled; never infer one by dividing the other. The commands are specified, but no model run was made here. **Try it on your machine (W2):** run A/A, then alternate A and B with fixed inputs. Compare raw throughput and timing, p10 and median, and adjacent pairs; a real change should exceed the A/A spread. No dated value for this exact comparison is supplied.

The outline also names `tools/kbench model.gguf` for per-kernel work. It needs a model load and Metal execution, so it is likewise deferred. Its results are GPU-oriented kernel observations, not an end-to-end token rate. Record the operation, weight bytes used for the bandwidth calculation, warm-up rule, passes, and whether the result came from a quiet or contended interval. Do not present the historical best-of-seven figures in ENGINE-MATH §11.7 as measurements from this session.

How do the earlier reader experiments fit together?

Chapters 8, 9, 12, 13, 14, and 17–20 contain W2 or W3 reader experiments. Keep each experiment attached to its chapter and use this queue to plan a low-priority session if you choose to run several. Build each variant from the frozen snapshot or the chapter's stated scratch copy, set the approved local paths, and gate the runs. Interleave comparable variants instead of completing all runs for one candidate first. Media cases need their local files as well as the model and encoder; the evaluation text and media files are not in the snapshot.

For each capture, save raw output and the conditions already specified in its chapter. The flags that deliberately skip kernels or rows are timing probes only; their outputs must not be treated as valid model answers. A future session should keep the ordinary baseline beside those probes, so the comparison shows the scope and cost of each selected path.

```
# ch08: E4B prefill and CPU contrast; use the same text input
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" -f "$TEXT" --bench 512 16 -r 3
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -t -ngl 0 -m "$MODEL" -f "$TEXT" --bench 512 16 -r 3

# ch09: isolated add_norm profiling probe (not a correctness run)
GEMMA4_PROFILE=1 nice -n 10 code/snapshot/tools/quiet.sh \
  code/build/gemma4 -m "$MODEL" -p "hello" --bench 1 1 -r 3
GEMMA4_PROFILE=1 GEMMA4_SKIP=add_norm nice -n 10 \
  code/snapshot/tools/quiet.sh code/build/gemma4 -m "$MODEL" \
  -p "hello" --bench 1 1 -r 3

# ch12: context depth cases, repeated in an interleaved order
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" --bench 512 64 --depth 4096
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" --bench 512 64 --depth 16384
# Also run kbench's attention sweep on the model (GPU work, W2):
nice -n 10 cc -O3 -DGEMMA4_METAL -o code/build/kbench \
  code/snapshot/tools/kbench.c code/snapshot/gemma4_metal.m \
  -framework Metal -framework Foundation
nice -n 10 code/build/kbench "$MODEL"

# ch13: PLE timing baseline and deliberately skipped gate
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" -p "A small example" -n 1
GEMMA4_SKIP=pl_gate nice -n 10 code/snapshot/tools/quiet.sh \
  code/build/gemma4 -m "$MODEL" -p "A small example" -n 1

# ch14: row-gather comparison; prepare the documented scratch variant first
mkdir -p code/build/no-row-skip
cp -R code/snapshot code/build/no-row-skip/source
# In that copy, comment out only gemma4.c lines 2094-2108; then:
nice -n 10 sh code/build/no-row-skip/source/build.sh
GEMMA4_TIMING=1 nice -n 10 code/snapshot/tools/quiet.sh \
  code/build/gemma4 -m "$MODEL" -f code/snapshot/README.md --bench 512 64
GEMMA4_TIMING=1 nice -n 10 code/snapshot/tools/quiet.sh \
  code/build/no-row-skip/build/gemma4 -m "$MODEL" \
```

PREPRINT

Draft book, October 2026. Not peer reviewed.


```
  -f code/snapshot/README.md --bench 512 64

# ch20: score distributions and the model-backed math suite
nice -n 10 cc -O2 -DGEMMA4_MEDIA -o code/build/check_math \
  code/snapshot/tools/check_math.c -lm -lpthread
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" -f "$TEXT" -c 512 --chunks 4 --kv f16 \
  --kl-divergence-base base.kld
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" --kv f16 --kl-divergence-base base.kld --kl-divergence
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" --kl-divergence-base base.kld --kl-divergence
nice -n 10 code/snapshot/tools/quiet.sh code/build/check_math all "$MODEL"

# ch17: W3 image encoding, at two image-token limits and save rows
# Set ROWS70 and ROWS1120 to distinct local output paths.
GEMMA4_DUMP_EMBD="$ROWS70" nice -n 10 code/snapshot/tools/quiet.sh \
  code/build/gemma4 -m "$MODEL" --mmproj "$MM" --image photo.jpg \
  -p "What is in this picture?" --image-tokens 70
GEMMA4_DUMP_EMBD="$ROWS1120" nice -n 10 code/snapshot/tools/quiet.sh \
  code/build/gemma4 -m "$MODEL" --mmproj "$MM" --image photo.jpg \
  -p "What is in this picture?" --image-tokens 1120

# ch18: W3 speech encode and transcription
nice -n 10 code/snapshot/tools/quiet.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" --audio speech.wav \
  -p "Transcribe this audio verbatim."

# ch19: W3 text, image, and video memory comparison; run separately
nice -n 10 code/snapshot/tools/quiet.sh \
  code/snapshot/tools/memwatch.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" -p "Describe the attached media." -n 20
nice -n 10 code/snapshot/tools/quiet.sh \
  code/snapshot/tools/memwatch.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" --image photo.jpg \
  -p "Describe the attached media." -n 20
nice -n 10 code/snapshot/tools/quiet.sh \
  code/snapshot/tools/memwatch.sh code/build/gemma4 \
  -m "$MODEL" --mmproj "$MM" --video clip-10s.mp4 \
  -p "Describe the attached media." -n 20
```

This is a command queue, not a single shell script: some commands compare intentionally different paths, some require the scratch build for chapter 14, and some require W3 media inputs. Run each experiment only with its chapter's exact controls, capture interleaved pairs where the comparison allows it, and record the limitation for any item you cannot complete. Set `ROWS70` and `ROWS1120` to distinct output paths; the commands assign each path to `GEMMA4_DUMP_EMBD`; the 70-token and 1,120-token cases should save and compare those rows. The chapter 19 memory runs must remain separate so the added media type is clear.

PREPRINT

Pitfall: a process footprint can hide GPU memory

A process-only memory report can appear to show a small engine footprint even while the model's file-backed pages are wired for the GPU. The documented symptom is a discrepancy between process resident-set figures and the system-wide memory rise during generation. The cause is that the process measure does not account for GPU-wired file pages. The project therefore uses wired, anonymous, and compressed system memory in `tools/memwatch.sh`; it reports a peak rise from the pre-run baseline and warns that other processes add noise. Source: snapshot README, "Memory" (2026-10-03/04 results), and the comments in `tools/memwatch.sh:2--5`. When memory is part of the claim, report that system-wide measure and its limits instead of presenting the process RSS as total engine memory.

Exercises

1. Design an A/A run set for decode at a fixed context. What do you keep identical, and what does the spread tell you about a later three-percent candidate result?
2. Convert the dated 24.7 tokens/s at 33K into milliseconds per generated token. If the weight stream is 2.61 GB per token, how many GB/s of weight bytes does that throughput represent? What extra traffic is omitted?
3. Explain why A, A, A, B, B, B is weaker than A, B, A, B, A, B when the fanless machine is warming.
4. The quiet helper runs after a timeout warning. What must the report say, and which utilization samples are still needed for the stricter mean-below-8
5. A profile reports 200 microseconds for a small step while the normal dispatch floor is about 8.5 microseconds. Name the measurement boundary that changed and one way to measure repeated work without inserting a wait after each call.

Rebuild checkpoint: measure each milestone without changing its claim

Chapter 21 adds no implementation milestone. It provides the measurement wrapper used when checking m1–m8: fix the workload, capture the conditions, compare repeated runs in an interleaved order, and retain raw observations. A future checkpoint can attach the relevant workload to each milestone: tokenizer throughput for m1, file walk for m2, arithmetic kernels for m3–m4, attention and cache cases for m5, full CPU forward for m6, Metal comparisons for m7, and prefix reuse for m8. The same timing label must keep its meaning across the comparison.

A speed claim is ready when another reader can identify the binary, model, workload, order, system state, timing boundary, raw sample count, and reported statistic. If any of these are unknown, the result can still be a useful observation, but it is not yet a controlled comparison. On this shared Mac, a measurement that says "busy" is more valuable than a quiet-sounding number with no record of what was running.

PREPRINT

Chapter 22

Making it fast: the story

The question

What bought the speed, what did not, and how would you decide next time?

Reading map: 24 pages

The estimate gives room to follow twelve decisions from bottleneck to evidence. The episodes distinguish historical measurements from this chapter's safe W0 and W1 captures; the longer account also separates kernel results from end-to-end speed and records the costs in accuracy, memory, or confidence.

Section	Pages
Why the chronology matters; decode and prefill have different roofs	2
Episodes 1–3: skip dead rows, keep the table on CPU, wire only used tensors	3
Episodes 4–6: F16 layer matrices, carried norms, useful matvec rows	3
Episodes 7–8: prefill attention, decode attention, and the KV format	5
Episodes 9–11: prompt chunks, encoder residency, and E2B's eight heads	3
Episode 12: the roofline, remaining latency, and rejected candidates	3
Source listings and the W0/W1 evidence trail	2
Pitfall, decision flow, exercises, hints, and checkpoint	3
Total	**24**

The idea

Why does the order of the wins matter?

The E4B has 8 query heads and 2 KV heads: four queries share each key/value head. It also has 42 layers, with the final 18 reusing earlier KV rows. Those facts matter because an optimization can remove work only when later computation no longer needs its result. The story begins by removing such work, then follows the bytes and the steps that remain.

Prefill and decode ask different hardware questions. During prefill, one weight tile serves many prompt rows, so arithmetic dominates. During decode, the engine streams most of the model's weights for a single new row, so bytes and memory bandwidth dominate. A faster prefill kernel does not prove a faster decode, and a smaller cache does not prove a faster matmul.

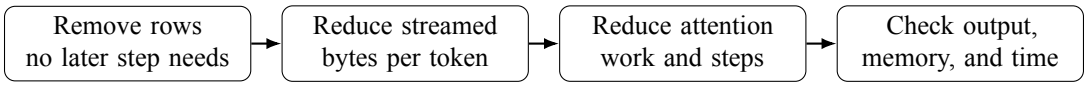


Figure 22.1: The optimization history follows the work that remains after each earlier reduction.

PREPRINT

What should every episode report?

Each episode needs a named workload and a baseline. Then it needs a measurement that identified the limiting work, a change that addresses that work, and a result under comparable conditions. Finally it needs the cost: exactness, memory, load time, or complexity. The historical figures below are from project notes dated 2026-10-03 through 2026-10-06; they are not fresh measurements on this writing run.

Episodes 1–3: where did the unnecessary work go?

The prefill path used to carry every prompt row through every layer. But once the final layer that writes a KV cache has passed, most prompt rows no longer affect a requested logit. Keeping only the needed row raised the recorded prefill rate from 260 to 440 tokens/s with bit-identical output. This is the strongest kind of optimization: less work, same answer.

The 1.94 GB per-layer token table seemed like an obvious GPU mapping target. Each token needs only one 7.4 KB row, though, and the CPU can fetch that row while the GPU works on the previous token. Leaving it on the CPU saved memory without lowering the recorded pipelined throughput. The table is not on the critical path merely because it is large.

The model file contains tensors the GPU never reads: that table, unused K/V weights in shared layers, and F32 originals after conversion to F16. The mapper therefore wires selected ranges, not the entire file. For the earlier Q4_0 file the notes record 2.63 of 4.84 GB wired; the E4B IQ4_XS file later reported 2.51 GB. These figures describe different files and must not be mixed as if they were one measurement.

Episodes 4–6: when is lower precision safe?

The 84 per-layer input-gate and projection matrices are F32 in the file and read on every token. Converting them once to F16 cuts their traffic in half. Quantising them further to Q8_0 saved another half of those bytes and gave 2

The engine also stopped asking each matvec to compute an input norm independently. The previous step now emits the next normalized input in F16, so several consumers reuse it. On the Q4_K LM head this raised the measured rate from 74 to 93 GB/s; the attention projections also saved 15 microseconds. The normalization moved into work that was already required.

For the small Q4 matvecs, four rows per simdgroup worked better than two or eight. Wider tiles looked promising because they could reuse activations, but the 64-row variant fell to 44 tokens/s from 260: accumulators spilled out of registers. A design that reduces traffic on paper can lose when the working state no longer fits the fast storage.

Episodes 7–8: why were attention kernels different?

Prefill attention on global 512-dimensional heads first reached 0.27 TFLOP/s. It reloaded queries and computed the same score work twice. The new kernel keeps the four query heads that share one KV head in a threadgroup; the eight query heads and two KV heads define the grouping. Constant loop bounds and then a better online-softmax rescale moved the historical kernel through 0.57, 0.85, and 1.44 TFLOP/s. Score comparisons stayed within the noted tolerances; the 8-bit cache remained an intentional approximation.

Decode attention had another limit. Splitting a 32K history into 1024 tiny chunks made the merge read 17 MB per layer. Longer chunks reduced the count to at most 64 for the 8-bit cache, or 16 for fp16. Separately, building fp16 bit patterns from int8 values replaced costly conversions, lifting the 32K kernel from 47 to 68 GB/s in the documented run. The result is faster attention, not a faster model by the same ratio: the weights still take most of a decode token.

PREPRINT

Draft book, October 2026. Not peer reviewed.

Episodes 9–11: what changed outside the main matvec?

Prompt calls became 256 rows instead of 512. The notes say speed per token stayed similar while activation memory halved. This is a memory-capacity improvement with no claimed throughput gain.

The encoders resize images across CPU cores and map their weights only while encoding. This avoids holding a second large set of weights for the whole text-generation turn. The episode is about residency and staging, not a universal speed number.

E2B exposed a grouping limit: its 8 query heads share one KV head, compared with the E4B's two. The 8-bit prefill kernel originally rejected that shape and fell back to CPU attention. The fix divided heads across threadgroups; prefill at 512 tokens rose from 450 to 1,652 tokens/s. Its 8-bit cache still differed more from llama.cpp than E4B's, so the speed fix and the cache accuracy question remain separate.

Episode 12: where does the remaining time go?

The decode matvecs already stream near 91–93 GB/s against an estimated 102 GB/s peak. The measured path is close to the bandwidth roof, while roughly twelve dependent GPU steps per layer leave a latency floor. Removing one step might save around 1.3–1.5

Episode 1: why can prompt rows disappear?

A prompt with 512 tokens creates 512 hidden-state rows. The model still needs the final row's next-token logits, but the rows that came before it have already supplied their keys and values to the cache. For E4B, the first 24 layers write their own cache; the final 18 shared-KV layers only read a cache written earlier. Once the forward pass has completed the cache-writing layers, those earlier rows cannot contribute a new cache entry in the shared layers. Only the rows requested by the caller need to continue.

The change is deliberately at a boundary where the dependency changes. It does not skip the early layers, where every prompt position must write its KV state. Nor does it skip a row that a caller actually requests. The source moves the chosen rows down in the activation buffers, gathers their per-layer embedding values as well, updates the row count, then continues through the later layers. The guard ensures rows are not overwritten before being copied. A chat prefill normally asks for just the last row's logits; a scoring path may request a different set.

The historical report is 260 to 440 prefill tokens/s, with perplexity unchanged to six digits and output bit-identical in the documented test. That is roughly 1.69 times the throughput for the particular benchmark. The number is a dated result, not a promise for every prompt. The causal explanation is stronger than the rate alone: fewer rows enter the remaining 18 layers, so those layers perform less matrix and per-layer-embedding work. A regression check should compare both requested logits and the KV state the later decode will consume, not merely the visible generated sentence.

Episode 2: why does the largest table stay off the GPU?

The token table is about 1.94 GB and contains 262,144 rows, each about 7.4 KB. A tempting implementation would map it into a Metal buffer because the GPU performs the model's other large matrix operations. But a decode step asks for one row selected by the previous token. Streaming the whole table would replace a small random row read with a huge upload or residency cost.

The pipeline gives the CPU a useful place in the schedule. After the GPU has selected the next token, the CPU looks up that token's row and prepares its embedding and projection. Those GPU operations are submitted while the CPU reads the row, and the GPU remains busy. The sequence has a dependency—the next token ID must exist before its row can be selected—but the row fetch can overlap later GPU work. The README's recorded comparison says the table stayed on the CPU without a speed loss in pipelined decode, while 1.94 GB did not become GPU buffers.

This is a schedule argument, not a general rule that CPUs should read embedding tables. If the CPU lookup became exposed on the critical path, if a batch needed many rows, or if the caller disabled pipelining, the balance could change. To revisit it, time the row lookup and the overlapped GPU interval

PREPRINT

separately, then compare end-to-end decode. Moving the table because it is large would answer the wrong question; the useful question is whether its access is visible in the token latency.

Episode 3: why map the holes between tensors?

A memory-mapped GGUF file is one address range, but a GPU mapping need not include every byte in it. The model loader already knows which tensors supply the token table, which layers reuse someone else's K/V cache, and which F32 source tensors have been copied to F16. The mapper sorts tensor descriptors by data address, marks those exclusions, and wires the dense ranges between them. A run may contain padding between adjacent tensors; the host maps a contiguous byte interval around that padding rather than making one object per matrix.

The ledger in the README distinguishes model variants. The initial Q4_0 file was 4.84 GB and 2.63 GB was wired. The later IQ4_XS E4B file was 4.72 GB and 2.51 GB was wired. In both cases the table remained CPU-mapped, shared-layer K/V weights were not read, and the original F32 matrices were excluded after their F16 copies were made. The unused tensors remain in the GGUF, so this strategy changes GPU residency and memory pressure without changing the model's stored file.

A mapping total is not an RSS number. Metal-wired file pages, ordinary process allocations, KV state, and compressed pages are separate parts of the machine's memory picture. The mapper's printed count answers "how many bytes did this routine map for the GPU?" It does not answer "how much memory will a long chat use?" Chapter 23 follows that wider ledger. For a mapping change, check both the mapped byte total and a system-level peak under the same workload.

Episode 4: why stop at F16 for the small matrices?

The per-layer embedding path has 42 pairs of small input-gate and projection matrices: 84 matrices, about 220 MB in F32. Every generated token reads them. The loader reads each source with uncached I/O and makes an F16 copy for the GPU, so each token streams half as many matrix bytes. This uses a modest amount of extra resident memory for the copies, but avoids keeping the originals' pages in the working set.

The next precision step was Q8_0. It halves those F16 bytes again and did produce about a 2

This is not evidence that Q8_0 is unusable for every matrix. It is evidence about these matrices, this model, this scoring set, and this speed gain. An opt-in mode would need its own documented name, tests, and user-visible quality trade-off. A safer variant could target only matrices shown to tolerate the change, but each subset would need another paired correctness and speed run. The useful lesson is to choose precision from the accuracy budget, rather than infer accuracy from the bit width or from a few readable answers.

Episode 5: can the next operation compute the norm early?

A decode layer repeatedly needs normalized input: attention projections use one normalized vector, the FFN uses the post-attention vector, and the next layer uses another. If each independent matvec computes the same RMS norm in its own threadgroups, it repeats reductions and adds work to the critical path. The engine now writes the next normalized input as part of the previous layer's final residual step, in an F16 buffer that the following kernels already consume.

The recorded Q4_K LM-head comparison rose from 74 to 93 GB/s when it received a pre-normalized F16 input. The attention projections were 15 microseconds faster. The gain is not simply a faster RMS-norm kernel: the standalone operation disappears from the schedule and its result is reused. The source shows the pipeline by passing `out` and `w2` through the fused residual-plus-normalization operation; the next layer reads `a->xn`.

That fusion introduces a data-lifetime contract. The producing kernel must finish writing the exact input buffer before its consumer reads it, and the buffer must retain the representation and scale expected

PREPRINT

by the consumer. A change to the order of operations can alter rounding even if the mathematical expression is equivalent over real numbers. Chapter 20's score tests are therefore still needed. The test should cover both the standalone helper and the full path, and the benchmark should measure the sequence with its normal command-buffer boundaries.

Episode 6: why did four rows beat a wider tile?

A tiled kernel can reduce activation traffic by reusing each activation value across more weight rows. The first proposal doubled the rows in a threadgroup from 32 to 64. In the estimate, that halves activation reads, which sounds attractive for a prompt matrix multiply. In the measured implementation, however, the extra partial sums occupied too many registers. The accumulators spilled, and performance collapsed from 260 to 44 tokens/s. Fewer input loads were overwhelmed by slower accesses to spilled state.

The smaller decode matvec experiment also varied rows per simdgroup for the Q4_0 implementation. Four rows were the useful point; two rows did not improve it, and eight were slower (the documented test was 324 versus 306 microseconds). Larger groups have fewer output groups and more values live at once. Smaller groups can leave the GPU underfilled or repeat setup work. The optimal shape depends on the register footprint, occupancy, number of rows, and weight decoder.

The practical search is a controlled sweep, not a single hand-picked tile. Start with kernel counters or a source-level estimate of live accumulators. Try a small number of shapes that fit the GPU's execution model, time them in alternation, and check the complete result path. A tile that wins on a large synthetic matrix may still lose on the engine's smaller output matrices. The failed 64-row tile is valuable history: it prevents a plausible optimization from being repeated without checking register pressure.

Episode 7: what changed the prefill-attention roof?

The early global attention kernel handled 512-dimensional heads by splitting output dimensions. In a 32K context, a 512-token chunk's attention took about four times as long as the rest of its prefill. The kernel reloaded queries for each block of eight keys and computed Q dot K twice, once for each half of the output dimensions. A benchmark isolated the global attention kernel at 0.27 TFLOP/s, showing that this part had room even though the large prefill matmuls were already compute-heavy.

The replacement grouped all four query heads that share one KV head. For E4B, there are two KV heads, so the eight query heads form two such groups. Each simdgroup owns 128 dimensions of one head; the threadgroup cooperates to form the score matrix and keeps query and output values in registers. Keys are multiplied once. This was a match between the model's 8-to-2 head geometry and the kernel's data layout, not merely a larger threadgroup.

The first layout reached 0.57 TFLOP/s. Making loops over key blocks and dimensions constant let the compiler unroll them and avoid accumulators spilling to memory, raising it to 0.85. The online softmax then proved to have a distinct cost: rescaling output accumulators after every key block made P·V expensive. A lazy maximum, which only moved for a block that exceeded the running maximum by more than eight, avoided most rescaling. When rescaling was needed, a matrix product applied the per-row factors. The reported kernel rate reached 1.44 TFLOP/s.

The accuracy notes matter here. The fp16 path's score comparisons remained stable at the printed precision in the cited cases. The 8-bit KV path already introduces quantisation; its KLD values are larger than fp16's, and the change did not make that quantisation disappear. The operation is justified by measured distribution comparisons, not by treating "same visible text" as exactness. Also note the scope: the vision encoder uses 72-dimensional heads and keeps a different kernel.

Episode 8: why did decode attention need different tricks?

At 32K, an early decode kernel divided the history into 1024 chunks of 32 keys. Every query head produced an unnormalized partial result plus its maximum and sum; a later kernel merged the pieces. That merge read about 17 MB per layer. The fix lets chunks grow with context, limiting them to 64 for

PREPRINT

the 8-bit cache or 16 for fp16. The source records the measured turning point: for 8-bit KV, 64 chunks took 1.04 ms, 128 took 1.06 ms, and 16 took 1.27 ms at 32K. Too few chunks also starve parallel work or make each chunk too long.

The first 8-bit decode kernel reached only 47 GB/s. Each query-head simdgroup converted the same int8 cache values to float. A later version uses the bit representation of a half value: each stored byte encodes an int8 value offset by 128; the byte can be placed in a half mantissa representing 1024 plus that byte, then the known offset is removed from the accumulated dot product. This turns many conversions into bit operations and simple arithmetic. The documented 32K kernel reached 68 GB/s, with time falling from 1.52 to 1.04 ms.

This trick is specific to the 8-bit representation and requires careful derivation. The scale is still applied, the offset correction depends on the query values for scores, and the value path has its own running-weight correction. A mistaken offset can produce coherent but wrong scores. Test a small vector against explicit integer conversion, then test full score rows and the attention output. The higher kernel bandwidth also does not mean the whole token improves by 45

The 8-bit cache is itself a deliberate memory and speed trade. At 32K it reduced reported memory from 3.77 to 3.36 GB and improved long-context throughput in the dated measurements. It also moved mean KLD from around 0.00001 for fp16 to roughly 0.0002 in cited contexts, with same-top agreement around 99.2–99.9

Episode 9: why halve the prompt call size?

The chat loop processes prompts in calls of 256 tokens rather than 512. The recorded rate per token was similar, while peak activation memory was halved. This affects a different resource from model weights: activation buffers scale with the number of rows in flight, while most decode weight traffic does not. It makes long prompts more comfortable on a 16 GB machine, and can reduce pressure that would otherwise evict useful pages or compete with other applications.

A smaller call changes boundaries. The engine must carry positions and KV state correctly from one call to the next, preserve the first-row offset used for the output, and ensure that rows which no longer need logits are not accidentally dropped before their KV writes. The correctness suite should compare chunked prefill against one-call prefill on a short prompt, then test a boundary where the prompt length is one more than a multiple of 256. The score path should verify the first token after the boundary.

The source note makes no throughput claim for the smaller call. Do not turn “half the activation memory” into “twice as fast.” If a machine is memory constrained, lower activation memory can prevent paging or leave room for other work; under a different workload it might not change speed at all. This is a useful example of an optimization whose value is capacity and stability.

Episode 10: why keep encoder weights transient?

Image and audio paths use large encoder weights that text generation does not need continuously. The CPU resizes image input across its cores, and the engine maps encoder weights while the encode operation runs, then releases that residency. The text model’s wired weights remain available for decode. This is a lifetime optimization: it limits how long two model components compete for unified memory.

The result depends on the product path. A video or live-audio loop may encode repeatedly, so mapping and staging overhead can become visible. A one-off picture in a conversation may make the temporary mapping cheap relative to the rest of the turn. Quantising the encoder file is another memory option, but the project notes report that Q8_0 reduced file size without making the encoder faster at the tested row counts; those matrix products were compute-bound. Lower precision should therefore be described as a storage choice unless timing shows otherwise.

A useful measurement separates resize, mapping, encoder execution, and language-model prefill. It should report the encoder file type and precision, number of image patches or audio frames, and whether the model was already warm. The per-task memory peak matters more than a static “all weights loaded”

PREPRINT

total if weights are resident only during the encoder call. Chapter 19 explains the media flow; this episode records why its memory lifetime belongs in a performance history.

Episode 11: what did E2B reveal about grouped heads?

E2B has eight query heads and one KV head, so all eight queries share the same key and value stream. E4B has eight query heads and two KV heads, or four queries per KV head. The first 8-bit prefill attention kernel allocated enough simdgroups for at most four query heads in a KV group. Its host guard rejected E2B's shape, returned failure, and let the CPU reference path compute attention. Correctness survived; performance did not. The E2B prefill sat near 450 tokens/s despite having a quarter of E4B's arithmetic, and the benchmark showed 27.7 seconds of CPU time for about five seconds of wall time.

The kernel divides a KV group's queries into pieces that fit the threadgroup-memory budget. The selected group width is the smaller of the head-sharing count and the number of available simdgroups divided by the 128-dimension parts per head. If that width does not cover every query head, multiple threadgroups handle the same KV group. For E2B's 256-dimensional case, eight heads fit in one 16-simdgroup group; a 512-dimensional case uses smaller groups and may read keys and values more than once.

After the fix, the recorded 512-token prefill rose from 450 to 1,652 tokens/s and CPU attention time fell from 27.7 to 0.16 seconds. This is a large gain because it removed an unintended CPU fallback, not because E2B became intrinsically more efficient than E4B. The accuracy check also needs nuance: E2B's one-KV-head 8-bit cache shares the same quantized values across more query heads. The notes measured mean KLD around 0.0012 and 98.9

Episode 12: which candidates were left on the table?

The last part of the story is a record of things that were estimated, attempted, or rejected. The baseline matters: the optimization notes say the decode matvecs already reached 91–93 GB/s on a quiet run, compared with a 102 GB/s peak estimate. Re-layout might recover at most the remaining bandwidth gap, but it would require about 2.2 GB of extra memory and a longer load. On a 16 GB machine already under pressure, a theoretical ten percent upper bound was not enough reason to add that copy.

Fusing one dependency could save a dispatch and barrier, estimated at roughly 8–10 microseconds per layer. Across 42 layers, that is about 0.4 ms, or roughly 1.3–1.5

Prefetch is the instructive exception: it was implemented and then reverted. The expected overlap window could theoretically fetch the next weights while small dependent GPU kernels ran. Interleaved pairs instead showed slower decode in four of five comparisons, by 3–7

What did the failed probes teach?

The optimization notes list more than one way to make attention look parallel. The decode kernel tried four-wide loads, a matrix formulation that placed the four shared-query heads into columns, and a lane layout that reused each value across four heads. None beat the original in the measured test. The matrix version had half of an eight-column product doing padding; the reuse layout added a 31-shuffle reduce-scatter plus barriers and broadcasts. The operation saved repeated conversion but added communication that cost more than it saved. The final bit-pattern path retained the simple head layout and removed only the conversion bottleneck.

Another probe made the output tile larger to reduce activation reads. Its measured 44 tokens/s instead of 260 is a warning that a kernel can be limited by local storage even when the memory-traffic estimate improves. The matvec row-count sweeps also changed sign across kernel families: Q4_0's useful choice was four rows, while the Q4_K LM head has its own two-row structure. Do not copy the row count from one dequantization format to another without measuring register footprint and occupancy.

The notes also tried key blocks of eight and sixteen for prefill attention. They reached 0.65 and 0.78 TFLOP/s, below the 32-key version, because a fixed per-step cost is paid more often. The fp16-bit-

PREPRINT

pattern conversion that helped decode reached only 1.34 TFLOP/s in prefill; the prefill data reuse and matrix layout are different. Putting P scratch in a buffer with less threadgroup memory required another barrier and ran slower. Lazy maximum in decode added a branch per key and increased the 32K attention time from 1.03 to 1.16 ms. Each rejected result answers a narrow question; it does not show that the technique is always poor.

A final set of candidates was rejected before implementation. The CPU already handled only a small token-table row and sampling while the GPU stayed busy, so more CPU SIMD had no exposed work to accelerate. The model's large weights were mapped without a copy, making private Metal storage unsuitable for the same purpose. A lower-bit weight or KV representation could shrink the byte stream, but changes output and requires an explicit quality choice. The notes leave a distinction between "not tried," "not useful in this configuration," and "too small to prove." Keep those labels in a future log rather than writing a single catch-all "optimization failed."

What makes a small measured win credible?

The performance notes distinguish quiet and contended sessions. Under quiet conditions, gate-plus-up reached about 91–93 GB/s against an estimated 102 GB/s peak. In the later optimization session, a browser, TTS batch, and 15 GB of 16 GB memory in use pushed decode results down to 19–22 tokens/s and matvecs to 59–75 GB/s. The prefill stayed near 437 tokens/s because that phase was compute-bound and less sensitive to the same bandwidth contention. Absolute rates from those sessions do not form a single baseline.

Interleaved trials help compare candidates under drift. In the prefetch experiment, one pair favored the candidate, but four of five pairs were slower and the candidate's observations were generally lower. That pattern is stronger evidence than one "best" run. The A/A spread still sets a practical limit: if ordinary repeats vary by five percent, a 1.5

The statistic should match the claim. For an end-to-end benchmark, preserve raw run outputs and report the median or the selected percentile with the sample count. For a microbenchmark, report the repeated-call boundary and whether synchronization happens around the batch. For a kernel, state the weight or activation bytes used to compute bandwidth and note that the achieved rate is derived from that byte estimate. For the project history, explicitly identify best-of-seven, interleaved pairs, or a single recorded run. If the source only gives a range or best, do not turn it into a median.

The machine state is part of the result. The historical files identify an M3 base Mac with eight GPU cores, 16 GiB shared memory and an estimated 102 GB/s peak; they also record the fanless heat effect. This chapter's W0 and W1 captures were made on the current Mac15,12, but they time no GPU work and make no speed claim. That distinction keeps a harmless script run from being mistaken for a fresh performance result.

Candidate	Why it looked useful	Evidence or cost	Decision
One fewer step	Remove a barrier and launch	About 1.3–1.5%; below observed noise	Not claimed
Weight re-layout	Better request efficiency	At most 10%; +2.2 GB	Not built
CPU SIMD/AMX	Faster row and sampling work	CPU mostly hidden by GPU work	Rejected
Private GPU heap	Lower mapping overhead	No-copy mapped weights already fit better	Rejected
Concurrent prefetch	Fill idle DRAM windows	3–7% slower in most pairs	Reverted
Lower-bit weights	Fewer streamed bytes	Changes logits and token decisions	Opt-in research only

The final estimate is not that the engine has reached a universal optimum. It is that the remaining same-output decode opportunities identified in the notes were too small to prove above the shared-machine noise, while the large byte stream was already close to the memory roof. A new machine, a different model, or a more reliable measurement window can change that conclusion. The book should preserve both the evidence and its date so readers can re-evaluate it.

PREPRINT

What does the optimization log look like as a compact record?

A short log makes it easier to compare unlike gains without flattening them into one "speedup" number.

No.	Observed limit	Change and recorded result	Cost or scope
1	Dead prompt rows in late layers	Drop rows after the cache-writing boundary; 260 to 440 tok/s	Bit-identical in cited check
2	Large token table might consume GPU memory	Read one CPU row and overlap work	No speed loss reported
3	Unneeded tensors resident on GPU	Wire only used ranges; 2.63/4.84 GB on Q4_0	File variant matters
4	F32 layer matrices read each token	Make F16 copies; Q8_0 rejected	KL and top IDs moved
5	Duplicate normalization work	Carry next F16 norm forward	Reuse and rounding must match
6	Matvec occupancy / register pressure	Four-row grouping retained; wide tile spilled	Shape is kernel-specific
7	Global prefill attention slow	Reuse query and score work; reach 1.44 TFLOP/s	Kernel-level number
8	Decode chunks and int conversions costly	Longer chunks and half-bit trick; 47 to 68 GB/s	8-bit path only
9	Activation buffers large	Prompt calls 512 to 256 rows	Memory, not speed
10	Encoder residency overlaps text weights	Map during encode, release after	Repeated media may differ
11	E2B fell back to CPU attention	Split the eight-head KV group	450 to 1,652 tok/s prefill
12	Remaining decode dominated by bytes / latency	Measure candidates; revert slow prefetch	Further gains below noise

This table is an index to the narrative, not a substitute for the dated conditions. The prefill number for episode 1 came from the Q4_0 work; the later attention numbers use specific 8-bit or fp16 paths. The current snapshot may not contain the old baseline binary or a reverted kernel. Treat the table as project history and reproduce a claim only when the exact source revision, model, workload, and measurement controls are available.

How should a reader separate the different kinds of win?

A speed story is easy to misread when one table combines different operations. The row-skipping result is end-to-end prompt prefill. The 1.44 TFLOP/s result belongs to a global attention kernel. The 68 GB/s result is a decode-attention kernel at 32K. The 93 GB/s result is a Q4_K LM-head matvec. These observations have different units and different denominators. A kernel-level gain identifies a local bottleneck; it does not tell us the whole model's speedup unless the kernel's share of the total is known.

For example, the dated model notes attribute about 7 ms of a 41 ms 32K decode token to global attention, while weight matvecs account for the other 33 ms. Even if the attention part ran twice as fast, the total would fall only by about 3.5 ms, roughly 9

The cache format changes both memory and numerical behavior. At 32K, the project table reports memory of 3.77 GB for fp16 KV and 3.36 GB for 8-bit KV, with the 8-bit run also faster in that measurement. The same section reports mean KLD around 0.0002 for 8-bit and around 0.00001 for fp16. A comparison that says only "8-bit is faster" omits the accuracy change; one that says only "8-bit is smaller" omits its runtime effect. Both belong in a complete record.

A table is most useful when each row names its phase, model file, cache type, context, chunk length, idle or heat state, and timing boundary. If a historic result lacks one of those details, mark it as absent rather than filling it from a neighboring table. The README has a particularly clear warning: decode immediately after 32K prefill was 12.5 tokens/s, while a two-minute cooldown gave 20.2. A number without its thermal state can be numerically accurate and still unsuitable for a source comparison.

REPRINT

When does fewer bytes beat a clever kernel?

Decode streams model weights for each generated token. On the recorded E4B model, the large matvecs already approached the memory bandwidth measured in a quiet run. If a kernel's time tracks the bytes it streams, reducing arithmetic in that kernel may do little; reducing the bytes or avoiding a read can matter more. But a lower-bit matrix changes its numerical representation. A smaller byte count is not free if it changes probabilities or the selected token.

The chapter's arithmetic example is intentionally simple. If 2.61 GB of weight bytes are read per token and the measured rate is 24.7 tokens/s, their product is about 64.5 GB/s. Dividing one second by 24.7 gives about 40.5 ms per token. The first number does not include KV traffic, activation reads, or CPU work. The second describes end-to-end time, not just the weight kernels. The difference between a theoretical byte floor and observed token time includes all the other work and timing boundaries.

Prefill amortizes a matrix read over many prompt rows. That raises arithmetic per weight read, so prefill matmuls can be compute-bound even though decode matvecs are bandwidth-bound. Attention also changes shape: prefill forms score matrices across many query rows, while decode attends one new row over a long cache. This explains why a row tile, a simdgroup layout, or a key chunk can help one phase and not another. Every claim should name which path it changes.

There are also two kinds of memory benefit. Removing the 1.94 GB token table from GPU mappings reduces residency, but does not reduce the file size. Reducing prompt calls to 256 rows cuts activation memory, but keeps weight bytes per token similar. An 8-bit cache reduces KV storage and may lower cache traffic, but changes the output distribution. The memory ledger and the speed table should not merge these into a generic "lighter model" claim.

What does the accuracy cost look like across the episodes?

Some changes preserve the same arithmetic and only avoid work that cannot affect requested outputs. The row-skip path is reported bit-identical. The CPU token-table path changes where a row is read, while maintaining the established pipeline result. Mapping fewer unused tensors changes residency, not the computed values. These are strong candidates for a default optimization when the correct dependencies are demonstrated.

Other changes alter representation or arithmetic. F32-to-F16 copies reduce traffic and introduce rounding, though the notes report the supported path's numerical comparisons. Q8_0 layer matrices produced a larger KLD and changed top tokens, so that variant was rejected. The 8-bit KV cache is kept as the default after broader evaluation but remains an approximation; fp16 remains available for comparisons. The E2B fix also had an arithmetic difference between CPU and Metal accumulation, and its before/after PPL values should be read with the distinct accumulation modes in view.

A stable greedy token stream is a useful but coarse gate. Many logits may move without changing the winner. KLD over matching scored positions sees more of the distribution, but depends on aligned inputs, text, chunking, and a saved reference with finite precision. Perplexity summarizes how surprising the known next tokens were; a close average can hide row-level differences. A careful acceptance note states which evidence ran and avoids converting a historical test's threshold into a universal law.

For each episode, ask what form of exactness is appropriate. A row gather should preserve the requested float values and the future cache dependencies. A format change should compare full scores and token winners. A kernel-layout change should first compare the affected output against an independent CPU implementation or a small checked vector, then run a model-level score gate. A memory-only change should verify the mapped tensor set and check that it did not cause another allocation or lifetime change. The scope of the gate follows the scope of the modification.

How can a W2 reproduction be made fair?

The source snapshot captures the final kernels, but not every historical baseline. A credible reproduction of one of the twelve episodes needs a saved baseline source and a candidate source built into separate

PREPRINT

outputs under `code/build/`. It also needs one fixed evaluation text in the book folder, since the old helper referenced an external `gemma4-chat/eval_text.txt`. The helper below accepts binaries, labels, and paths instead of relying on that external file.

Before the run, write down the comparison question. For prefill attention, fix prompt length, context, cache format, model, warm-up, and which kernel result is being extracted. For decode at long context, specify depth, generated count, cooldown, and cache type. Use the same model and text for both binaries. Interleave A then B repeatedly so a rising temperature or changing background load does not put all observations into one candidate. Reverse the starting order in a second set if the machine permits.

Keep the raw output rather than only the best rate. Record the date, chip, uptime, idle-gate reading, model identity, text identity, command, and whether the result is wall time or GPU time. For the generated output, compare fixed greedy IDs; for a numeric path, compare the affected values and then run the model-level KLD or PPL check. The chapter 21 A/A protocol estimates how much variation to expect before interpreting a small A/B difference.

The command is shown for later use only. The baseline and candidate must already be prepared in the book's isolated build directory, and the approved evaluation text must live in the book folder. This W2 run was not performed while writing the chapter.

```
MODEL=/path/to/local/gemma-4-E4B-it-IQ4_XS.gguf
TEXT=/path/to/local/ch22-eval.txt
export MODEL TEXT
nice -n 10 sh code/examples/ch22-ab.sh 5 base code/build/gemma4_base \
  candidate code/build/gemma4_candidate code/expected/ch22-w2.txt
```

Try it on your machine (W2): run the commands above with the saved binaries and fixed inputs, then inspect raw interleaved observations and exactness. Read any change against the A/A spread; a semantics-preserving optimization should retain exactness. No dated result for these exact binaries is cited here.

How do the episodes change the next experiment?

The history suggests an order of questions for a new idea. First ask whether later computation consumes the value. If not, measure whether dropping it affects any requested output or state. Next identify whether the target path is limited by bytes, arithmetic, register pressure, occupancy, or dependent launches. Only after that should a kernel shape or representation be changed. Finally, measure under conditions where the expected effect is larger than the noise.

This order prevents a common mistake: optimizing the part that looks expensive in source code. A large allocation may be cold; a short kernel may wait on a barrier; a wide tile may spill; an idle bus may not mean bandwidth is available if the next access misses cache. The notes combine source inspection, kernel microbenchmarks, end-to-end runs, and accuracy comparisons because no single tool diagnoses all of those causes.

It also changes the burden of proof. If a change removes a dependency that demonstrably cannot affect output, source reasoning plus exact checks can be compelling. If it changes numerical representation, the correctness evidence must quantify the change. If it claims two-percent speedup on a machine whose A/A spread is five percent, more controlled measurements are needed before calling it a win. If a candidate needs two gigabytes of extra memory to gain at most ten percent, that cost belongs beside the upper bound.

A useful final entry ends with one of three outcomes: keep as an exact default, expose as an explicit quality/memory option, or revert because the gain was not established. The prefetch episode ended in the third category despite correct outputs; the Q8_0 layer matrices ended there because accuracy cost outweighed the modest gain; the 8-bit KV cache remained an explicit, evaluated approximation. Keeping unsuccessful outcomes in the story saves future readers from repeating expensive experiments and gives them the original conditions to challenge.

PREPRINT

Which memory number belongs to which optimization?

Memory claims are especially easy to combine incorrectly because "the model" appears in several representations at once. The GGUF file remains on disk and is mapped by the CPU. Some of its tensor ranges are wrapped as no-copy Metal buffers and wired for the GPU. Other data is copied into F16 buffers; the token table remains a CPU mapping. At runtime, activations and the KV cache add memory that depends on prompt length, context length, and precision. The project history reports these quantities separately.

Quantity	Dated value	What it describes
E4B IQ4_XS file	4.72 GB	Entire model file, including tensors not wired for the GPU
GPU-wired weights	2.51 GB	Selected tensor ranges mapped for Metal
Per-layer token table	1.94 GB	CPU mapped; a single row is used per token
F16 layer-matrix copies	about 0.11 GB	Small F16 inputs created from F32 originals
32K KV cache, fp16	580 MB	Context state, mostly the global layers
32K KV cache, 8-bit	308 MB	Lower-precision cache with a measured KLD cost
Process memory at 32K	about 0.63 GB	Runtime allocations, in addition to wired weights

These figures do not all come from one memory counter or one model variant. The file size is decimal GB from the model record. The KV figures are calculated from active layers and context. The process-footprint number came from a particular 32K run; the system-wide memory rise also includes GPU-wired file pages and other applications. The mapper's wired number is a separate count of mapped weight ranges. A reader should keep the label and method attached to every value.

This ledger explains why an optimization may lower one number and leave another unchanged. Episode 2 keeps the 1.94 GB table outside GPU buffers, but the table remains in the GGUF and can still occupy file cache pages as the CPU reads rows. Episode 3 lowers wired bytes by excluding unused tensors. Episode 4 adds an F16 copy while bypassing cache for the F32 original. Episode 8 reduces KV bytes while changing the cache values. Episode 9 reduces prompt activations by lowering the number of rows in a call. Calling all four "model size reductions" would erase their distinct effects.

How should the dated speed numbers be grouped?

The history spans several experiments. The row-skip comparison reports Q4_0 prefill before and after the forward-path change. The 1.65× ratio belongs to prompt prefill, not generated-token decode. The 0.27, 0.57, 0.85, and 1.44 TFLOP/s sequence belongs to global prefill-attention kernel evolution. The decode-attention 47 to 68 GB/s change belongs to an 8-bit cache at 32K. The Q4_K LM-head 74 to 93 GB/s result tests the effect of supplying an already normalized F16 input. They cannot be placed into one speedup column without stating their scopes.

The E2B attention correction is another distinct experiment. Its 450 to 1,652 tokens/s result is 512-token prefill. It does not represent E4B decode. Its cause was a rejected Metal dispatch and CPU fallback for eight queries sharing one KV head. The fixed kernel's grouped-head assignment was then measured on the same E2B model. The E2B distribution comparison also reports a larger 8-bit-cache KLD than E4B's; that numerical difference belongs to the cache format and architecture, not to the performance fix by itself.

Heat created a separate split in the 32K data. The 12.5 tokens/s decode followed minutes of prefill and was explicitly labelled hot; the 20.2 tokens/s observation followed a two-minute pause. The memory and prefill table around those values used additional model and cache configurations. Any comparison that joins a hot value from one column to a cooled value from another is invalid even if both are labeled "32K decode." Chapter 21's protocol—same depth, same cache type, interleaved candidates, and state captured through the run—makes the phase boundaries explicit.

The source documentation also distinguishes measured data from estimates. The memory bandwidth peak of 102 GB/s is an estimate from the M3 memory interface, while 91–93 GB/s came from kernel timing and byte counts. The roughly 1.3–1.5

PREPRINT

What should a source citation let a reader recover?

The frozen README gives the narrative for the E4B-specific choices: “Speed and memory: what was done,” “Long context,” “The 8-bit KV cache,” “Faster prefill attention,” and “Things that did not work.” ENGINE-MATH §11 provides the machine model, roofline, candidate list, and dated results under §11.7. Its earlier attention discussions (§4.4–§4.5) explain the online-softmax and int8 offset behavior. The E2B group-size failure is described in the E2B section near §12. Each source records both the observed result and the limitation that explains why it is not universal.

The frozen C source is the authority for the current row gather (2094–2114), F16 conversion (904–925), and mapped ranges (4518–4553). In the Metal source, the current Q4_K matvec and fused normalization are at 681–740 and 765–800; prefill attention excerpt at 1172–1204; the 8-bit conversion at 1410–1445; the chunk cap at 2120–2140; and E2B’s group assignment at 1282–1286. The dated README and ENGINE-MATH notes explain the branches that changed or were reverted.

A range citation is not the whole proof. For example, the current `attn_decode` comment says the half-bit path reached 68 GB/s, but reproducing that result also requires the original 47 GB/s baseline, exact context, cache mode, kernel timing boundary, and machine state. The chapter therefore pairs code excerpts with the historical sources rather than suggesting that a final source line by itself proves the performance claim. If a future source revision changes the algorithm, update the listing and treat the old result as history until the new code has been measured.

The real code

The current source shows how the row reduction, selective mapping, and longer decode chunks are implemented. These are excerpts from the frozen snapshot; historical changes are described from the dated README and ENGINE-MATH notes.

Where do unneeded rows stop?

At the first layer whose future no longer needs the old rows, the forward pass gathers the requested rows and reduces the active row count. This is the code behind episode 1.

```
2090      int Tl = T, pos = st->n_past;  // the layers' rows, and the position
          of the first
2091      for (int l = 0; l < m->n_layer; l++) {
2092          const Layer *L = &m->layers[l];
2093          const float *next_norm = l + 1 < m->n_layer ? m->layers[l + 1].
          attn_norm : m->output_norm;
2094          if (l == m->n_kv && Tl > n_out && first >= n_out && !a->bidir) {
2095              // The layers from here on write no KV cache, so the rows
          whose logits aren't wanted affect nothing
2096              // any more: only rows [first, first + n_out) go on (in a
          prefill, just the last). (Rows are moved
2097              // down, which needs first >= n_out so that no row is
          overwritten before it is read.)
2098              for (int t = 0; t < n_out; t++) a->rows[t] = first + t;
2099              const int P = m->n_layer * m->n_pl;
2100              Tensor xr = {.type = T_F32, .ne = {E, T, 1, 1}, .data = (
          const uint8_t *)a->x};
2101              Tensor pr = {.type = T_F32, .ne = {P, T, 1, 1}, .data = (
          const uint8_t *)a->pl};
2102              EmbedJob xj = {&xr, a->rows, a->x, 1.0f}, pj = {&pr, a->rows
          , a->pl, 1.0f};
2103              GatherJob gj = {a->xn, a->xn, a->rows, E};
2104              STEP(metal_get_rows(a->x, a->x, E * sizeof(float), T_F32, a
          ->rows, n_out, E, 1.0f), n_out, embed_row, &xj);
```

```
2105            STEP(metal_gather(a->xn, a->xn, a->rows, n_out, E), n_out,
    gather_row, &gj);
2106            if (P) STEP(metal_get_rows(a->pl, a->pl, P * sizeof(float),
    T_F32, a->rows, n_out, P, 1.0f), n_out, embed_row, &pj);
2107            Tl = n_out, pos += first, first = 0;
2108        }
```

Listing 22.1: gemma4.c:2090–2108

Which tensors are left out of GPU mappings?

The mapper collects the token table, shared-layer K/V weights, and F32 inputs with F16 copies as unused. It maps only the contiguous ranges between them.

```
4518 static void map_model(const Gguf *g, const Model *m) {
4519     const Tensor **ts = xmalloc(g->n_tensors * sizeof(Tensor *));
4520     for (int i = 0; i < g->n_tensors; i++) ts[i] = &g->tensors[i];
4521     qsort(ts, g->n_tensors, sizeof(Tensor *), cmp_tensor_data);
4522     const Tensor *unused[4 * MAX_LAYERS + 1];
4523     int n_unused = 0;
4524     unused[n_unused++] = m->pl_tok;
4525     for (int l = 0; l < m->n_layer; l++) {
4526         const Layer *L = &m->layers[l];
4527         const Tensor *t[4] = {L->kv_src != l ? layer_tensor(g, l, "
    attn_k.weight", 0) : NULL,
4528                               L->kv_src != l ? layer_tensor(g, l, "
    attn_v.weight", 0) : NULL,
4529                               m->n_pl ? layer_tensor(g, l, "inp_gate.
    weight", 0) : NULL,
4530                               m->n_pl ? layer_tensor(g, l, "proj.weight"
    , 0) : NULL};
4531         if (t[2] == L->pl_gate) t[2] = NULL;  // (used as it is)
4532         if (t[3] == L->pl_proj) t[3] = NULL;
4533         for (int j = 0; j < 4; j++)
4534             if (t[j]) unused[n_unused++] = t[j];
4535     }
4536     char *expert = xcalloc(g->n_tensors, 1);  // (or unused)
4537     for (int i = 0; i < g->n_tensors; i++) {
4538         for (int l = 0; l < m->n_layer; l++) {
4539             const Layer *L = &m->layers[l];
4540             if (L->router && (ts[i] == L->exp_gate || ts[i] == L->exp_up
     || ts[i] == L->exp_down)) expert[i] = 1;
4541         }
4542         for (int j = 0; j < n_unused; j++) expert[i] |= ts[i] == unused[
    j];
4543     }
4544     size_t dense = 0;
4545     for (int i = 0; i < g->n_tensors;) {
4546         if (expert[i]) { i++; continue; }
4547         const uint8_t *start = ts[i]->data, *end = start;
4548         for (; i < g->n_tensors && !expert[i]; i++)
4549             if (ts[i]->data + tensor_nbytes(ts[i]) > end) end = ts[i]->
    data + tensor_nbytes(ts[i]);
4550         if (!metal_map(start, end - start)) die("metal: cannot map the
    weights");
4551         dense += end - start;
4552     }
4553     fprintf(stderr, "metal: %.2f GB of weights, wired\n", dense / 1e9);
```

Listing 22.2: gemma4.c:4518–4553

Where are the F16 copies made?

The loader keeps the F32 original out of the page cache while allocating a shared F16 buffer for the GPU. The comment records the Q8_0 accuracy result beside the conversion, a useful reminder that the rejected alternative belongs to this code path's history.

```
904 // An F32 matrix as an F16 copy, in memory shared with the GPU: half the
        bytes to read per token. The F32 original
905 // is read with pread (no cache), so that its pages never come into
        memory. (As Q8_0 they would be half that
906 // again, but these matrices are sensitive: KLD against llama.cpp went
        from 0.00001 to 0.001.)
907 static const Tensor *f16_copy(const Tensor *t, int fd, const uint8_t *
        base) {
908     if (t->type != T_F32) return t;
909     Tensor *c = xmalloc(sizeof(Tensor));
910     *c = *t;
911     c->type = T_F16;
912     const size_t n = (size_t)t->ne[0] * t->ne[1] * t->ne[2] * t->ne[3];
913     f16 *dst = shared_alloc(n * sizeof(f16));
914     float *tmp = xmalloc(n * sizeof(float));
915     uint8_t *p = (uint8_t *)tmp;
916     for (size_t left = n * 4, off = (size_t)(t->data - base); left;) {
917         const ssize_t r = pread(fd, p, left, (off_t)off);
918         if (r <= 0) die("cannot read the model file");
919         p += r, off += r, left -= r;
920     }
921     for (size_t i = 0; i < n; i++) dst[i] = float_to_half(tmp[i]);
922     free(tmp);
923     c->data = (const uint8_t *)dst;
924     return c;
925 }
```

Listing 22.3: gemma4.c:904–925

How does one step prepare the next normalized input?

This kernel fuses the residual update with the next RMS normalization. The optional output is half precision and is ready for the next step; the producer therefore removes a separate normalization dispatch while retaining an explicit buffer boundary.

```
765 // row tg, 256 threads (n <= 4096): x = (x + rmsnorm(a) · w) · out_scale
        , and (has_out) out = rmsnorm(x) · w2 in fp16,
766 // the next step's normed input. A whole threadgroup per row: in decode
        there is one row, and a simdgroup alone
767 // would take several times as long.
768 struct AddNormParams { int n; float eps; float out_scale; int has_out;
        };
769
770 kernel void add_norm(device float *x [[buffer(0)]], device const float *
        a [[buffer(1)]],
771                      device const float *w [[buffer(2)]], device half *
        out [[buffer(3)]],
```

```
                        device const float *w2 [[buffer(4)]], constant
    AddNormParams &p [[buffer(5)]],
                        uint row [[threadgroup_position_in_grid]], uint tid
     [[thread_index_in_threadgroup]],
                        uint nt [[threads_per_threadgroup]]) {
    threadgroup float tmp[32];
    const int m = (p.n + nt - 1) / nt;  // (at most 16)
    device const float *ar = a + (ulong)row * p.n;
    device float *xr = x + (ulong)row * p.n;
    float v[16], ss = 0;
    for (int j = 0; j < m; j++) {
        const int i = tid + nt * j;
        v[j] = i < p.n ? ar[i] : 0.0f;
        ss += v[j] * v[j];
    }
    const float s = rsqrt(block_sum(ss, tmp, tid, nt) / p.n + p.eps);
    ss = 0;
    for (int j = 0; j < m; j++) {
        const int i = tid + nt * j;
        if (i >= p.n) continue;
        v[j] = (xr[i] + v[j] * s * w[i]) * p.out_scale;
        xr[i] = v[j];
        ss += v[j] * v[j];
    }
    if (!p.has_out) return;
    const float s2 = rsqrt(block_sum(ss, tmp, tid, nt) / p.n + p.eps);
    for (int j = 0; j < m; j++) {
        const int i = tid + nt * j;
        if (i < p.n) out[(ulong)row * p.n + i] = half(v[j] * s2 * w2[i])
    ;
    }
}
```

Listing 22.4: gemma4_metal.m:765–800

How does prefill attention keep its query work local?

The specialized kernel groups the four E4B query heads that share a KV head, then assigns 128 dimensions of a head to each simdgroup. This excerpt shows the layout and the fixed loop bounds that kept accumulators in registers; the kernel continues beyond the excerpt.

```
// Flash attention for heads of 128·NP dims (256, 512: the language
    model's), T > 1. A threadgroup takes 8
// consecutive queries of all G query heads of KV head g; simdgroup (hh,
     part) owns dims [128·part, +128) of query
// head g·G + hh, so its queries and its output accumulators stay in
    registers. Per block of 32 keys each
// simdgroup computes its dims' share of S = Q·K, the NP shares of a
    head are summed through threadgroup memory,
// every simdgroup of the head runs the same online softmax on the sum,
    and accumulates P·V for its dims. The G
// heads' simdgroups read the same K and V rows, which the core's cache
    then serves. (The general kernel above
// reloaded the queries of heads over 128 dims for every 8 keys, and
    computed S twice for 512-dim heads: 0.27
// TFLOP/s; 4× slower at a 32K context than the rest of a prefill.)
kernel void attention_fa(device const half *q [[buffer(0)]], device
    const half *k [[buffer(1)]],
```

```
1181                          device const half *v [[buffer(2)]], device half
      *o [[buffer(3)]],
1182                          constant AttnParams &p [[buffer(4)]], uint2 tg
     [[threadgroup_position_in_grid]],
1183                          ushort sg [[simdgroup_index_in_threadgroup]],
     ushort lane [[thread_index_in_simdgroup]]) {
      const int NP = HD / 128, G = p.n_head / p.n_head_kv;
      const int g = tg.x, t0 = tg.y * 8, hh = sg / NP, part = sg % NP, h =
      g * G + hh, d0 = part * 128;
      const int nq = p.n_head * p.hd, nkv = p.n_head_kv * p.hd;
      threadgroup float sp[16 * 256];  // [hh][part][key block j][8 × 8]:
     the shares of S
      simdgroup_half8x8 qf[16];
      for (int d = 0; d < 16; d++) simdgroup_load(qf[d], q + (ulong)t0 *
     nq + h * p.hd + d0 + d * 8, nq);
      simdgroup_float8x8 acc[16];
      for (int d = 0; d < 16; d++) acc[d] = simdgroup_float8x8(0);
      const short2 fc = frag_coord(lane);  // this lane holds row fc.x,
     columns fc.y and fc.y + 1 of each block
      const int pos = p.n_past + t0 + fc.x, s_last = p.n_past + (p.bidir ?
      p.T : min(t0 + 8, p.T)) - 1;
      const int hi = p.bidir ? s_last : pos, lo = p.window > 0 ? pos - p.
     window + 1 : 0;
      const int s_first = p.window > 0 ? max(0, p.n_past + t0 - p.window +
      1) / 8 * 8 : 0;
      float mx = -1e30f, l = 0;
      threadgroup float *mine = sp + (hh * NP + part) * 256, *head = sp +
     hh * NP * 256;
      // (all loops over d and j have constant bounds, with the blocks
     past s_last skipped by a condition inside:
      // a loop that breaks early isn't unrolled, and then the
     accumulators it indexes go to memory, which made
      // P·V take two thirds of the time)
      for (int s0 = s_first; s0 <= s_last; s0 += 32) {  // (the same keys
     for the whole threadgroup)
          const int nj = min(4, (s_last - s0) / 8 + 1);  // blocks of 8
     keys with some at or before s_last
          simdgroup_float8x8 sf[4];
          ulong row[4];
```

Listing 22.5: gemma4_metal.m:1172–1204

Where did the decode conversion go?

The int8 cache path loads packed bytes, constructs half values from their bit pattern, and removes the constant offset in the dot product. This is the central part of episode 8's conversion change.

```
1410      // (8-bit: each simdgroup converts the same int8 for its own head,
     and int-to-float conversions limited the
1411      // kernel to 47 GB/s; through fp16 bit patterns instead, 68 GB/s at
     32K)
1412      for (int s = s0, r = s0 % p.ring; s < s1; s++, r = r + 1 == p.ring ?
      0 : r + 1) {  // r: the row of position s
1413          const ulong at = (ulong)r * nkv + lo;
1414          // (4-wide loads: a lane's nd values are contiguous and 4-
     aligned; one at a time, byte loads were the
1415          // 8-bit cache's bottleneck)
1416          float x = 0;
```

```
1417         if (KV8) {  // byte b (an int8 + 128) ORed into 1024.0h's
     mantissa is 1024 + b: no int-to-float conversions
1418             device const uint *ku = (device const uint *)((device const
     char *)k + at);
1419             for (int i = 0; i < nd / 4; i++) {
1420                 const uint u = ku[i] ^ 0x80808080u;
1421                 const half2 lo = as_type<half2>((u & 0x00FF00FFu) | 0
     x64006400u), hi = as_type<half2>(((u >> 8) & 0x00FF00FFu) | 0
     x64006400u);
1422                 x += qv[4 * i] * float(lo.x) + qv[4 * i + 1] * float(hi.
     x) + qv[4 * i + 2] * float(lo.y) + qv[4 * i + 3] * float(hi.y);
1423             }
1424             x = (x - 1152.0f * qsum) * float(ksc[at / 32]);
1425         } else {
1426             device const half4 *ks = (device const half4 *)((device
     const half *)k + at);
1427             for (int i = 0; i < nd / 4; i++) x += dot(float4(qv[4 * i],
     qv[4 * i + 1], qv[4 * i + 2], qv[4 * i + 3]), float4(ks[i]));
1428         }
1429         x = simd_sum(x);
1430         const float m_new = max(m, x), corr = exp(m - m_new), pr = exp(x
      - m_new);
1431         l = l * corr + pr;
1432         if (KV8) {  // as for K: o accumulates p·d·(1024 + b), and w the
      p·d it must take 1152 times back
1433             device const uint *vu = (device const uint *)((device const
     char *)v + at);
1434             const float pv = pr * float(vsc[at / 32]);
1435             w = w * corr + pv;
1436             for (int i = 0; i < nd / 4; i++) {
1437                 const uint u = vu[i] ^ 0x80808080u;
1438                 const half2 lo = as_type<half2>((u & 0x00FF00FFu) | 0
     x64006400u), hi = as_type<half2>(((u >> 8) & 0x00FF00FFu) | 0
     x64006400u);
1439                 o[4 * i] = o[4 * i] * corr + pv * float(lo.x), o[4 * i +
      1] = o[4 * i + 1] * corr + pv * float(hi.x);
1440                 o[4 * i + 2] = o[4 * i + 2] * corr + pv * float(lo.y), o
     [4 * i + 3] = o[4 * i + 3] * corr + pv * float(hi.y);
1441             }
1442         } else {
1443             device const half4 *vs = (device const half4 *)((device
     const half *)v + at);
1444             for (int i = 0; i < nd / 4; i++) {
1445                 const float4 vv = float4(vs[i]);
```

Listing 22.6: `gemma4_metal.m:1410–1445`

How does the decode path cap merge work?

The key chunk grows with context so that the scratch merged by the smaller number of combine threadgroups stays bounded. E4B's four query heads per KV head are reflected in the grid shape.

```
2120         // keys per threadgroup: 32, or more once there would be over
     MAX_CHUNKS chunks per KV head, so that the
2121         // chunks' outputs (hd + 2 floats per head each), which
     attn_combine reads on few threadgroups, stay small.
2122         // Fewer, longer chunks were faster down to 64 chunks for the 8-
     bit cache and 16 for fp16 (at 32K, a global
```

```
2123         // layer's decode attention: 8-bit 1.04 ms with 64, 1.06 with
     128, 1.27 with 16; fp16 1.5 - 1.7 ms with 16,
2124         // 1.9 - 2.0 with 32 to 128); 1024 chunks of 32 (the first
     version) made 17 MB per layer to merge.
2125         const int MAX_CHUNKS = kv8 ? 64 : 16;
2126         static float *part;         // per query head and chunk: hd
     outputs, max, sum
2127         static size_t part_size;
2128         const int first = window > 0 && n_past - window + 1 > 0 ? n_past
      - window + 1 : 0, span = n_past + 1 - first;
2129         const int keys = span <= 32 * MAX_CHUNKS ? 32 : (span + 32 *
     MAX_CHUNKS - 1) / (32 * MAX_CHUNKS) * 32;
2130         const int n_chunks = (span + keys - 1) / keys;
2131         const size_t need = (size_t)n_head * n_chunks * (hd + 2) *
     sizeof(float);
2132         if (need > part_size) {  // (at most 128 chunks: a superseded
     buffer may still be in use, so it is kept)
2133             part_size = need;
2134             part = metal_alloc(part_size);
2135         }
2136         const AttnParams p = {T, n_head, n_head_kv, hd, 0, 0, 0, n_past,
      n_chunks, window, 0, 0, 0, ring, 0, keys, kv8};
2137         if (profiling) snprintf(prof_key, sizeof prof_key, "hd %d past %
     d%s", hd, n_past / 128 * 128, kv8 ? " kv8" : "");
2138         return ENCODE_PS(variant(K_ATTN_DECODE, 0, 0, 0, 0, 0, hd, 0, 0,
      kv8), K_ATTN_DECODE, p, size2(n_head_kv, n_chunks),
2139                          size2(32 * (n_head / n_head_kv), 1), q, k, v,
     part, ksb, vsb) &&
2140                ENCODE(K_ATTN_COMBINE, p, size2(n_head, 1), size2(256, 1)
     , part, o);
```

Listing 22.7: gemma4_metal.m:2120–2140

How does the E2B grouping fit the same kernel?

The number of query heads assigned to each threadgroup is limited by the available simdgroups. The host dispatch accepts up to eight query heads per KV head where the 8-bit path supports it.

```
1282     const int NP = HD / 128, G = p.n_head / p.n_head_kv;
1283     // at most 16 simdgroups (threadgroup memory): a KV head's G query
     heads go in G / Gs threadgroups when G * NP > 16 (E2B: G = 8, hd 512)
1284     const int Gs = min(G, 16 / NP), SPL = G / Gs;
1285     const int g = tg.x / SPL, t0 = tg.y * 8, hh = sg / NP, part = sg %
     NP, h = g * G + (tg.x % SPL) * Gs + hh, d0 = part * 128;
1286     const int nq = p.n_head * p.hd, nkv = p.n_head_kv * p.hd;
```

Listing 22.8: gemma4_metal.m:1282–1286

Experiment

The book's original experiment asks for W2 comparisons. This chapter's writing pass does not run inference. The two checks below are safe book-material checks: W0 arithmetic only and W1 header-only reading.

W0: can the speedup arithmetic be checked without weights?

A small script uses the dated figures as inputs and recomputes the ratio and per-token bandwidth. It does not benchmark the GPU; its output verifies the arithmetic only.

```
nice -n 10 python3 code/examples/ch22-roofline.py
```

Experiment W0

```
row-skip prefill ratio: 1.692x
decode-attention bandwidth ratio: 1.447x
weight traffic at 24.7 tok/s: 64.5 GB/s
time per token at 24.7 tok/s: 40.5 ms
```

Capture: 2026-10-22, Mac15,12; `uptime` load 6.64 / 41.55 / 57.74. Idle-GPU gate: not applicable; this arithmetic script uses no GPU. Full output: `code/expected/ch22-w0.txt`.

W1: what does the GGUF header say?

The header probe reads 24 bytes and reports version, tensor count, and metadata count. It does not read tensor descriptors, weights, or model tensors; the command should name the local file explicitly.

```
nice -n 10 python3 code/examples/ch22-gguf-header.py "$MODEL"
```

Experiment W1

```
GGUF version=3 tensors=720 metadata_entries=56
```

Capture: 2026-10-22, Mac15,12; `uptime` load 6.64 / 41.55 / 57.74. Idle-GPU gate: not applicable; the script read exactly 24 header bytes. Full output: `code/expected/ch22-w1.txt`.

What do the safe captures establish?

The W0 script checks the arithmetic used to discuss the dated results. It recomputes 440 divided by 260 as 1.692, 68 divided by 47 as 1.447, and 2.61 GB per token multiplied by 24.7 tokens per second as 64.5 GB/s. It also converts the observed rate to 40.5 milliseconds per token. These calculations are useful for spotting a transcription error or an implausible ratio. Since their inputs are recorded historical values, they do not establish that the current engine still runs at those rates.

The W1 reader opens the local GGUF and reads just 24 bytes. The captured header says version 3, 720 tensors, and 56 metadata entries. That can check the file's container identity and top-level counts. It cannot establish tensor shapes, the vocabulary, quantization types, GPU residency, runtime compatibility, or model quality; those require additional header or tensor-directory reads, and those were outside this experiment. In particular, no tensor payload was read and no engine binary was started.

Tag	Captured result	What it supports
W0	1.692x, 1.447x, 64.5 GB/s, 40.5 ms	Arithmetic from dated inputs; no fresh speed measurement
W1	GGUF v3, 720 tensors, 56 metadata entries	Top-level header only; no tensor descriptors or weights
W2	Reader experiment	Saved binaries, fixed text, and interleaved controls

The two safe captures are short by design. They make the distinction between a reproducible calculation and a reproducible engine measurement visible. A future W2 result should include the evidence that this small table cannot: raw observations, the machine state, exact binaries and model, fixed text and command, and a correctness comparison for the changed path.

PREPRINT

W2: what remains to be measured?

To reproduce an episode, build base and candidate binaries from the corresponding historical source in an isolated scratch tree, then interleave the same benchmark command. The historical experiment needs old source copies that are not part of the snapshot. Do not treat the dated notes as a fresh result. The interleaved W2 command appears below with its controls and output path. The baseline and candidate sources are not present in the snapshot, so their builds and the benchmark remain a future capture.

Experiment W2

Try it on your machine (W2): record uptime, idle gate, model hash, commands, raw interleaved results, median/p10, wall or GPU time, and exactness comparison. Interpret any speed change against the A/A spread; exactness should remain unchanged for a semantics-preserving optimization.

Pitfall: a faster number may describe a hotter or busier machine

Pitfall

The long-context table reports decode at 12.5 tokens/s just after a multi-minute prefill, then 20.2 after a two-minute pause. The cooled figure is not evidence that a source change caused the difference; the run state changed. The cause was thermal throttling on the fanless machine. The fix for a fair comparison is to separate phases and record the cooldown and run order. Source: snapshot README, “Long context” and “Heat” (2026-10-06); see also Chapter 21’s shared-machine protocol.

Decision flow: what should be tried next?

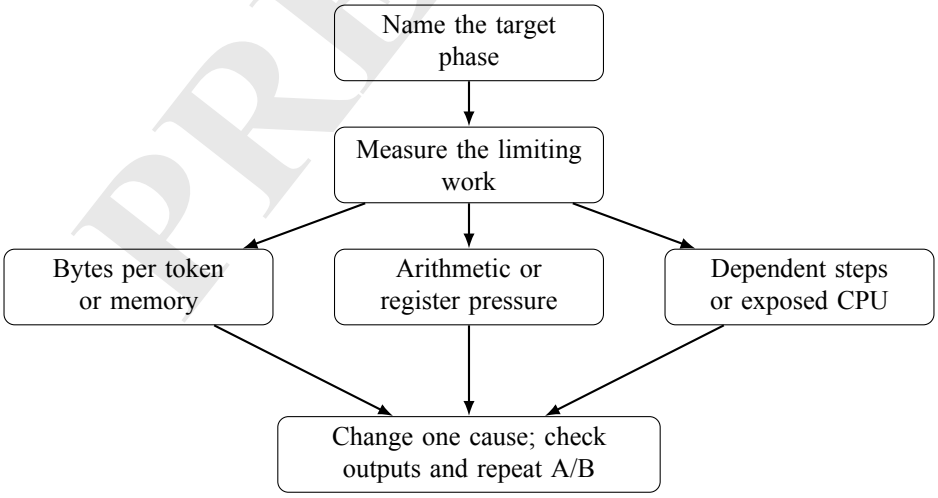


Figure 22.2: Match the proposed change to the measured limiter, then check both correctness and repeatability.

Start with the bytes and steps the target phase actually uses. If the operation is bandwidth-bound, reduce bytes without changing the answer, or state and measure an explicit quality trade. If it is compute-bound, reuse loaded values and keep working state in registers. If it is latency-bound, count dependent dispatches and compare any proposed reduction against A/A noise. Then test the affected arithmetic,

score distributions, greedy IDs, memory, and end-to-end timing. Keep a change only when its benefit is repeatable and its cost is understood.

Exercises

1. Estimate which of episodes 1–11 can still improve decode by more than five percent using the bytes-per-token table in Chapter 21. State the assumptions.
2. The per-layer matrices gained 2
3. Explain why the 64-row tile lost despite reducing activation traffic. Which hardware resource became the bottleneck?
4. Propose an A/B experiment for the 8-bit decode attention conversion. Name the exact context, controls, output checks, and statistic.
5. Write an optimization log entry for a change of your own: symptom, measurement, hypothesis, edit, result, and cost. Label any number you have not measured as unmeasured.

Rebuild checkpoint: keep the evidence with the change

Chapter 22 adds no m1–m8 implementation milestone. It applies Chapter 20's correctness gates and Chapter 21's measurement protocol to optimization work. For each change, preserve the smallest reproducible input, the relevant source range, raw before/after observations, and an exactness check matched to the affected path. A speed number without its workload, source identity, and result comparison is only an observation.

PREPRINT

Chapter 23

Memory

The question

Why does the engine use about 3.1 GB while the model file is 4.7 GB?

Reading map: 10 pages

Section	Pages
The memory ledger and what the totals count	2
Why only selected model bytes become GPU buffers	2
KV cache, activations, and context length	2
Measuring system memory and reading the capture	1
Register spills, shared-machine pressure, and the real pitfall	1
Source listings, exercises, hints, and checkpoint	2
Total	**10**

The idea

Which memory number are we trying to explain?

The model file is a disk object. The engine maps it and asks the GPU to use selected ranges. The operating system also keeps file-backed pages in its cache, while the process allocates working buffers for activations and the KV cache. Those are four different views of the same run, and a number from one view cannot be substituted for a number from another.

The project notes report about 3.08 GB of system memory rise for a 128-token generation with the Q4_0 file, against 5.08 GB for `llama-simple -ngl 99`. The E4B file is 4.84 GB in that comparison. The engine wires 2.63 GB of it for the GPU, plus its F16 copies, KV cache, and activations. The competing path maps the whole file into Metal buffers. These are historical README observations, not a new measurement from this writing pass; Chapter 21 explains why their shared-machine conditions matter.

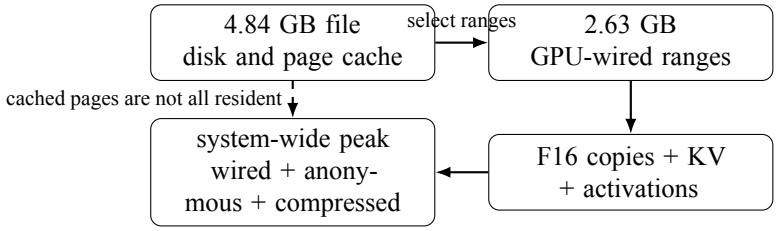


Figure 23.1: The file, GPU buffers, process allocations, and system-wide measurement overlap but are not interchangeable totals.

PREPRINT

Can the ledger add up exactly?

Start with the documented Q4_0 example. The engine's wired-weight line says 2.63 GB. Its own process footprint for a 4K chat is about 0.33 GB on top of that, covering the KV cache, activations, F16 copies, and tokenizer. Adding those two rounded values gives about 2.96 GB of process-related memory. The system-wide peak rise was 3.08 GB. The 0.12 GB difference is not a missing tensor: the measurements count different things, are rounded, and the system-wide sample includes allocator and operating-system effects.

This is a reconciliation, not a new independently measured total. Do not add the 0.11 GB of F16 copies or the 71 MB Q8 cache to the 0.33 GB again; both are already included in the process-footprint estimate. Likewise, do not add the 2.63 GB wired count a second time to the 3.08 GB system-wide rise. The latter already includes pages wired for the GPU.

For a longer-context example, the README records an IQ4_XS run at 32K with 2.51 GB wired and about 0.63 GB of process memory using the 8-bit cache, or about 3.14 GB together. It describes that as about 3.1 GB. With fp16 KV the process figure is 0.91 GB, so the corresponding rough total is 3.42 GB. Those figures belong to a different quantization and context setup than the 4K Q4_0 comparison. Chapter 11 has the cache byte totals; we use them here to identify the changing term rather than recalculate the architecture.

Why does the file not become one giant GPU buffer?

The mapping policy follows use. Dense weights read by Metal become buffers over consecutive file ranges. The E4B token table is 1.94 GB, but the CPU reads one selected row per token; the GPU does not need a buffer for it. Shared-KV layers reuse an earlier layer's cache and do not use their own K/V projection weights. The 84 F32 per-layer matrices are copied to F16 once, and their original file pages stay out of memory. Chapter 13 explains the token table and conversion; Chapter 14 gives the per-layer weight total of 2,182,512,640 bytes.

The consequence is visible in the banner printed by `map_model`: it reports the bytes wired for GPU access, not the size of the GGUF file, not total process memory, and not the system's peak. For IQ4_XS the README reports 2.51 GB wired from a 4.72 GB file. For Q4_0 the comparable values are 2.63 GB from 4.84 GB. The difference between file size and wired size is not a second copy saved in RAM: a large share is data intentionally left for CPU access or not used by this path.

What changes as the context grows?

The KV cache stores keys and values for layers that own them. E4B has 8 query heads and 2 KV heads; four query heads share each KV head. Only 24 of its 42 layers own a cache, while the 18 shared-KV layers point back to earlier storage. Sliding-window owners use a ring sized for their window and reply slack; global owners grow with context. The allocation code sums only the owning layers.

At 4K, Chapter 11 records 133 MB for fp16 KV and 71 MB for the default 8-bit cache. At 32K those totals are 580 MB and 308 MB. Thus context growth adds much less than four times the 4K cache: the sliding-window rings do not grow with the global context. The 8-bit form stores one byte per value plus an fp16 scale for each group of 32, so it is about 1.06 bytes per value rather than two. Chapter 11 shows the block layout and the exact byte accounting.

The prompt's temporary activations behave differently. Their peak is influenced by the number of rows processed together, not just the final context length. Chapter 14 describes the 256-row chat call; Chapter 17 measures the vision encoder's buffers and documents a wide tile whose register spill caused a 206 MB peak. Chapter 19 covers media staging and how reducing the encoder file from 990 MB to 559 MB affects its weight footprint. These should be added only for workloads that actually use those paths.

PREPRINT

Draft book, October 2026. Not peer reviewed.

What is fixed, and what follows the workload?

The main model's wired dense weights are mostly fixed after load. The 84 small F32 per-layer matrices become F16 copies: Chapter 13 gives 220,200,960 bytes of originals and 110,100,480 bytes of copies. The tokenizer tables and runtime structures are also modest fixed allocations. The cache depends on context and KV format. Activations depend on prefill batch size and which media encoders are active. Encoder weights are resident only when the media path requires them.

This classification helps find the first useful question when memory is tight. If changing context reduces the peak, inspect the KV allocation. If lowering the prefill chunk helps, inspect activation buffers. If loading a vision or audio prompt changes the peak, separate the encoder's temporary working set from retained language-model rows. A single "model uses X GB" number erases those useful distinctions.

Can the measurements be compared on one page?

The following table gathers earlier results into a common map. Values marked "about" are rounded project observations. The unit is decimal GB or MB, as in the README; do not silently interpret them as GiB or MiB. "System rise" is a peak delta, while the other entries describe a file, a wired range, or a process footprint.

Quantity	Reported value	What it describes	See
IQ4_XS model file	4.72 GB	bytes in the GGUF file	Ch. 3
IQ4_XS GPU wired	2.51 GB	dense tensor ranges sent to Metal	Ch. 3, 22
Q4_0 system rise	3.08 GB	peak wired + anonymous + compressed delta	Ch. 21
Q4_0 comparison	5.08 GB	llama.cpp's peak system rise	Ch. 21
Per-layer token table	1.94 GB	in the file; CPU reads one selected row	Ch. 13
Q8 KV at 4K / 32K	71 / 308 MB	sequence cache allocation	Ch. 11
FP16 KV at 4K / 32K	133 / 580 MB	sequence cache allocation	Ch. 11
F16 PLE copies	110 MB	process-side copies of 84 matrices	Ch. 13
Vision spill peak	206 MB	temporary spill/peak	Ch. 17
Q8 encoder file	559 MB	disk file; F16 original was 990 MB	Ch. 19

This table is useful because it resists a tempting but invalid operation: summing every row. The two model formats are alternatives, and the KV rows are alternatives by format and context. The token table's 1.94 GB is a file tensor, but the engine does not wire it as a GPU buffer. The 206 MB figure describes a temporary spill/peak, not a permanent allocation. The encoder file size is not automatically resident throughout text generation. Each row answers a different question.

Where would an 8 GB machine run short?

The model does not occupy its entire file size as wired memory, but the rest of the machine still needs room for the operating system, the application, cached files, and any other open programs. A small unified-memory machine can therefore feel pressure even when the process footprint looks modest. The first limit depends on the workload: a long text context grows the global KV cache; a large prefill or vision path needs more temporary rows; audio or video adds encoder work and staging; other applications reduce the headroom available to all of them.

The project provides a smaller E2B route for 8 GB systems. Its historical values are 1.23 GB wired for the model and a 244 MB process footprint, compared with E4B's 2.51 GB wired and 348 MB process footprint in the documented audio-query test. That is a substantial difference, but these are one-task observations, not an 8 GB capacity guarantee. The E2B context, media, concurrent applications, and memory pressure still matter. Chapter 19's 559 MB quantized encoder file also belongs in a media workload's inventory, but the encoder is not held in GPU buffers as a permanent addition during every text-only run.

Do not diagnose pressure from the word "compressed" alone. Compression can be an operating-system response to pressure, but a single sampled increase does not show which allocation caused it.

PREPRINT

Compare a baseline and a controlled run, keep the same open applications, and record whether compressed memory was already rising before the engine started. A user with 8 GB can first choose E2B, then reduce context or prompt batching if the measured cache or activation term is the one that grows. If the system remains pressured during a text-only short-context run, other concurrent memory users may dominate.

How should a reader interpret one capture?

Read the evidence in the order it was collected. First identify the exact GGUF and build. The header-only W1 probe confirms the container version and counts, but does not parse the tensor directory. The model's wired banner comes later, after the engine has mapped and selected tensors. A process-footprint reading belongs to the launched process; the `memwatch` delta belongs to the whole machine. A speed comparison needs the idle gate and repeated timing protocol from Chapter 21.

Next write down the workload that could allocate transient memory: context capacity, actual prompt length, number of rows processed per call, media type, and output length. Requested context and used context are not always the same: the state allocation reserves its configured cache before all positions have been filled. A short reply can therefore coexist with a cache sized for a much longer conversation. Similarly, an image can add a temporary encoder working set and then leave projected rows in the language-model prompt. Keep the encoder's input buffers separate from those retained rows when interpreting a peak.

Finally, treat a sampled peak as an observation at the sampling interval. `memwatch.sh` samples every 0.2 seconds, so it reports the highest value it saw, not a hardware guarantee that no briefer maximum existed between samples. Another process can change the system total during that interval. Run the same workload more than once and compare the raw readings; a change smaller than normal variation needs more evidence. The script also waits for its child and prints the memory summary even if the command's own output is redirected, which makes it convenient for logging but does not turn a system-wide sample into per-process attribution.

This is why the book keeps the measurements side by side rather than manufacturing one "true" total. The model banner answers "which weights are wired?" The process tool answers "what did this process report for its footprint?" The system sampler answers "how much did the machine's used-memory categories rise?" The ledger is a way to check whether those answers are plausible together, not a substitute for any one of them.

How can a reader locate a memory regression?

Change one workload dimension at a time and use the quantity that can see it. For a context regression, keep the model, prompt, and output fixed, then compare the allocated cache size reported by a diagnostic or the engine's allocation helper. The wired-weight banner should remain stable. A cache-format change should affect the cache estimate and may affect accuracy, so the comparison belongs beside Chapter 20's distribution checks.

For a prompt-batch regression, keep the final context and input text fixed but change the number of rows processed per call in a scratch build. The KV allocation can stay the same while activation buffers rise or fall. This is the sort of difference Chapter 14's call boundary and Chapter 17's activation accounting make visible. Record both peak memory and prefill behavior; reducing a batch may lower a temporary peak while adding dispatches or changing throughput.

For a media regression, begin with a text-only case and then add one image, audio clip, or video. Keep the main model and context fixed. Record whether the encoder file is loaded, the encoder's temporary working buffers, the number of projected rows, and the media staging allocation. Chapter 19 gives the staging limit and quantized encoder-file size. If you only compare total file sizes, you may mistake a smaller encoder file for a smaller runtime peak, or miss a transient staging buffer entirely.

The project history also gives a useful negative control. The wide-tile experiment did not change any model or cache allocation; its failing resource was register capacity, and the symptom was both a spill peak and a severe slowdown. If the wired banner and process allocation stay fixed but the kernel becomes

PREPRINT

slower and a profiler reports spills, adding up tensor bytes will not explain it. The evidence must match the resource that changed.

When is a smaller number a meaningful saving?

A memory optimization has at least two tests. First, did the intended category shrink under the same workload? Second, did the observable system peak or process footprint improve enough to matter outside run-to-run noise? A format can reduce a cache allocation but leave the measured system peak unchanged if another category dominates. An encoder file can be smaller on disk while the main-model wired ranges remain identical. A prompt chunk can lower activation peak without changing the reserved KV cache.

Keep correctness evidence next to these comparisons. Chapter 11 gives the 8-bit cache's size reduction; Chapter 20 records the corresponding KLD and top-token agreement. The Q8_0 encoder conversion in Chapter 19 reduces file size from 990 MB to 559 MB, but encoder-file size is not itself a quality test or a memory-watch measurement. Likewise, Chapter 13's F16 copies reduce the resident working set relative to reading all F32 originals each token, while their 110 MB footprint remains a real allocation. The right question is not simply whether a value went down, but whether the resource of interest changed, the output remained within the accepted gate, and the end-to-end workload benefited.

For a future quiet-machine capture, use an exact command and preserve all output. One possible text-only run, after building the snapshot binary, is:

```
nice -n 10 code/snapshot/tools/memwatch.sh \\
  code/build/gemma4 -m "$MODEL" \\
  -p "Name one color." -n 128
```

This is a W2 reader experiment. Before interpreting it, confirm the idle-GPU gate, record the baseline, note the wired banner and process footprint, and repeat the same prompt with the same binary. For W2, inspect wired banner and process footprint. For W3, repeat with a fixed encoder file and media input and inspect the reported rows, timing, and output; compare memory totals as separate measures.

What can the monitors attribute to one process?

The memory sampler can tell that the combined used-memory categories rose while a command was active; it cannot point to the exact page or allocation responsible for every part of the rise. If another application allocates memory during the sample, its pages enter the same system total. If the child exits quickly between sample points, the largest instant can be missed. The helper's own comment therefore asks for an otherwise quiet machine, and Chapter 21 adds the idle gate and repeated runs.

The wired-weight banner is more specific, but it has a narrower question. It reports the total byte spans the engine mapped for Metal. It does not report how many pages the operating system has evicted, how much anonymous activation storage is live, or how much compressed memory belongs to another app. Nor is that wired total the same as the GGUF's full size: the model map deliberately leaves out tensors the GPU path will not read.

The process footprint narrows ownership in another direction. The README's 4K figure of roughly 0.33 GB above wired weights describes the engine's own cache, activations, F16 copies, and tokenizer as one combined footprint. It does not isolate each allocator. The 32K readings similarly tell us that the process total is higher with fp16 KV than with Q8 KV, while Chapter 11 identifies the cache-format difference. We can connect those observations without claiming the process monitor independently measured the cache alone.

This separation is useful in debugging. If system rise changes but the process footprint and wired banner do not, investigate machine activity, sampling, and cached pages before changing engine allocation code. If the wired number changes, check tensor selection and quantization. If wired stays stable but the process footprint changes with context, inspect the KV state and any prompt buffers. If only a short-lived peak changes, inspect activation or media staging lifetime and collect a finer-grained trace if available. Each clue narrows the next measurement; none is a complete allocation profile by itself.

PREPRINT

Why does mapping a file not mean keeping all of it in RAM?

The GGUF loader creates a virtual mapping so tensors can be addressed as file-backed data. A mapping reserves an address range; physical pages are brought in when accessed. Metal then receives buffers for the selected dense ranges, and those pages must remain wired for GPU access. The rest of the file can remain mapped without every byte being resident or wired. Therefore "mapped," "resident," and "wired" describe different states.

This distinction explains how the token table can remain in the GGUF while not adding 1.94 GB of GPU-wired memory. Chapter 13 explains that a decode step needs one row and that the CPU reads that row with random-access advice. Accessing selected rows can still make their file pages useful to the CPU and the operating system's cache; it simply does not turn the entire table into a Metal buffer. Similarly, `map_model` excludes unused K/V tensors in shared layers. Their bytes remain present in the file for format compatibility, but this engine path does not wire them.

The contrast matters when comparing tools. A process listing may charge file-backed memory differently from the system's wired-page count. A file's size is stable even when most pages have never been touched. A GPU buffer total is a policy decision about what the device can access, not a census of every byte the application has ever opened. Keeping those meanings straight is the core of this chapter's memory ledger.

The real code

Which tensors are excluded from GPU wiring?

The comment states the policy, and the next source lines mark the per-layer token table and unused shared-layer K/V weights for exclusion. The same list includes F32 matrices for which the loader created F16 copies. These excerpts show why the file can remain larger than the wired set.

```
4514 // The GPU reads the dense weights through buffers over the mapped file, each a run of consecutive tensors it
4515 // reads, kept wired. Not in them: the experts (they come from the expert cache), the per-layer embeddings' token
4516 // table (the CPU reads its rows), F32 matrices the model has F16 copies of, and the K/V weights of shared-KV
4517 // layers (unused). Their pages are then never wired, and mostly never read.
4518 static void map_model(const Gguf *g, const Model *m) {
4519     const Tensor **ts = xmalloc(g->n_tensors * sizeof(Tensor *));
4520     for (int i = 0; i < g->n_tensors; i++) ts[i] = &g->tensors[i];
4521     qsort(ts, g->n_tensors, sizeof(Tensor *), cmp_tensor_data);
4522     const Tensor *unused[4 * MAX_LAYERS + 1];
4523     int n_unused = 0;
4524     unused[n_unused++] = m->pl_tok;
4525     for (int l = 0; l < m->n_layer; l++) {
4526         const Layer *L = &m->layers[l];
4527         const Tensor *t[4] = {L->kv_src != l ? layer_tensor(g, l, "attn_k.weight", 0) : NULL,
4528                               L->kv_src != l ? layer_tensor(g, l, "attn_v.weight", 0) : NULL,
4529                               m->n_pl ? layer_tensor(g, l, "inp_gate.weight", 0) : NULL,
4530                               m->n_pl ? layer_tensor(g, l, "proj.weight", 0) : NULL};
4531         if (t[2] == L->pl_gate) t[2] = NULL;  // (used as it is)
4532         if (t[3] == L->pl_proj) t[3] = NULL;
4533         for (int j = 0; j < 4; j++)
4534             if (t[j]) unused[n_unused++] = t[j];
```

```
4535     }
```

Listing 23.1: gemma4.c:4514–4535

How does the KV cache count owners and bytes?

The helper counts values and, in 8-bit mode, separate fp16 scales. It doubles each layer's key size for its key/value pair and skips layers whose `kv_src` points elsewhere. This compact path is the allocation rule behind Chapter 11's measured totals.

```
1164 // bytes of the keys (or the values) of one layer: the values, then (8-
     bit) the scales
1165 static size_t kv_layer_bytes(const Layer *L, int rows) {
1166     const size_t n = (size_t)rows * L->n_head_kv * L->hd;
1167     return kv_q8 ? n + n / 32 * sizeof(f16) : n * sizeof(f16);
1168 }
1169
1170 static size_t kv_bytes(const Model *m, int n_ctx, int slack) {  // the
     KV cache of a sequence of n_ctx positions
1171     const int ring = kv_ring(m, n_ctx, slack);
1172     size_t total = 0;
1173     for (int l = 0; l < m->n_layer; l++) {
1174         const Layer *L = &m->layers[l];
1175         if (L->kv_src == l) total += 2 * kv_layer_bytes(L, kv_rows(L,
     n_ctx, ring));
1176     }
1177     return total;
```

Listing 23.2: gemma4.c:1164–1177

What does the memory monitor sample?

The helper records its baseline, samples every 0.2 seconds while the child runs, and reports the largest observed rise. The source comment names the components included and warns that other processes add noise. This is a system-wide measurement, not a per-process resident-set reading.

```
2  # memwatch.sh CMD ARGS...: runs the command and reports the most memory
     the system used for it: the peak rise
3  # of wired + anonymous + compressed pages (what Activity Monitor calls "
     Memory Used", minus cached files) over
4  # the level before it started, sampled every 0.2 s. GPU-wired pages of a
      mapped model file count; the page
5  # cache does not. Other processes add noise, so run it on an otherwise
     quiet machine.
6  used() {
7      vm_stat | awk '/page size of/ {ps = $8}
8          /Pages wired down/ {w = $4} /Anonymous pages/ {a = $3} /Pages
     occupied by compressor/ {c = $5}
9          END {gsub("\\.", "", w); gsub("\\.", "", a); gsub("\\.", "", c);
      printf "%.0f\n", (w + a + c) * ps / 1e6}'
10 }
11 base=$(used)
12 "$@" &
13 pid=$!
14 peak=$base
15 while kill -0 $pid 2>/dev/null; do
16     u=$(used)
```

```
17      [ "$u" -gt "$peak" ] && peak=$u
18      sleep 0.2
19  done
20  wait $pid
21  echo "memwatch: peak memory used $(( (peak - base) )) MB above the $base
        MB before the start" >&2
```

Listing 23.3: `tools/memwatch.sh`:2–21

Experiment

W0: calculate an illustrative ledger without weights

The small program in `code/examples/ch23_memory_ledger.c` adds the already documented rounded values. It deliberately labels the residual rather than assigning it to an invented allocation. Build and run it with:

```
cc -O2 -Wall -Wextra \\
  -o code/build/ch23-memory-ledger \\
  code/examples/ch23_memory_ledger.c
nice -n 10 code/build/ch23-memory-ledger
```

The output is captured in `code/expected/ch23-w0.txt`. It should show 2.63 + 0.33 = 2.96 GB, against a 3.08 GB system-wide peak, and an unexplained 0.12 GB difference. These are the README's historical rounded inputs, not fresh measurements.

W1: check what the file header can establish

The header-only probe used in Chapter 22 reads exactly 24 bytes and reports the GGUF version, tensor count, and metadata-entry count. The captured line is:

```
GGUF version=3 tensors=720 metadata_entries=56
```

The capture from this W1 invocation is `code/expected/ch23-w1.txt`; the tensor payload is not read. This confirms the file's basic container facts, but cannot say which tensors are wired or how much memory inference will use.

How should a reader compare memory runs?

The historical command for a live run is `tools/memwatch.sh` around one fixed chat turn, with the model, prompt, output length, machine, and idle-gate result recorded as in Chapter 21. The E2B/E4B comparison helper also loads models and runs an audio query. Those are W2/W3 reader experiments. Try them with local inputs and record the system baseline, peak rise, process footprint, wired banner, and workload together; compare each measure separately. Record the system baseline, peak rise, process footprint, wired banner, and workload together; do not infer one from another.

PREPRINT

Pitfalls from the project notes

Pitfall

The process number can miss GPU-wired file pages. The source note says tools that report a process resident set miss file pages wired for the GPU. That can make a process-level number look smaller than the memory added to the machine. The project response was to use `memwatch.sh`, which samples system wired, anonymous, and compressed pages, while also keeping the engine's wired-weight banner and `/usr/bin/time -l` footprint as separate readings. The source is the snapshot README's "Memory" section and the explanatory comments at `tools/memwatch.sh:2--5`; Chapter 21 discusses measurement noise and gate failures.

Pitfall

A wider tile can spend memory to lose speed. The project's documented 64-row matmul tile reduced activation loads but its accumulators spilled from registers, taking 44 instead of 260 tokens/s. The README lists this as a failed experiment, not as an unexplained memory leak. Chapter 17 also records a 206 MB spill/peak in its vision-buffer experiment. The lesson is to inspect the affected working set and hardware resource, then report memory with speed; a tensor-size ledger alone will not reveal register spills. Source: snapshot README, "Things that did not work," and Chapter 17's documented experiment.

The machine in the speed notes already had 15 of 16 GB in use by other applications. That condition can cause compressed memory and noisy system-wide deltas; it does not change the model's tensor byte counts. The task's earlier chapters give the better way to keep these claims separated: Chapter 20 checks numerical behavior, Chapter 21 records machine state and measurement boundaries, and Chapter 22 connects optimizations to their resource costs.

Exercises

1. Rebuild the ledger for 16K context. Use Chapter 11's Q8 and fp16 cache totals, state which other categories are unchanged, and label any new total as an estimate rather than a measurement.
2. Where would an 8 GB machine run short first: wired model weights, context cache, media encoder buffers, or system headroom? Give one controlled measurement that could distinguish your answer from page-cache pressure.
3. Explain why 2.63 GB wired plus 0.33 GB process footprint is not required to equal the 3.08 GB system-wide rise exactly. Name what each instrument counts.
4. Compare the two cache formats at 32K from Chapter 11. How much memory is saved, and what accuracy evidence from Chapter 20 would you keep next to that saving?
5. A smaller prompt chunk lowers peak memory but leaves the 32K cache unchanged. Which chapter's measurements would you inspect to explain the change?

Rebuild checkpoint: keep the memory boundary with the number

Chapter 23 adds no m1–m8 milestone. When changing an allocation, record the requested context, KV format, prompt batch, media path, wired-weight banner, process footprint, and system-wide peak rise. Re-run the same workload behind Chapter 21's idle gate, and keep the correctness evidence required by Chapter 20. A number without its measurement boundary cannot tell the next reader what to reproduce.

PREPRINT

Chapter 24

Packaging: from engine to tool

The question

How does a program that evaluates tokens become a tool someone can launch, script, and use every day?

Reading map: 8 pages

Section	Pages
From source tree to a runnable folder	1
Build targets and the folder-aware launcher	1
Chat as a pipe protocol	2
The voice layer and the product boundary	1
Vendoring, reproducibility, and the W2 experiment	1
Pitfall, exercises, hints, and final checkpoint	2
Total	**8**

The idea

What does a person actually receive?

The engine's core job is narrow: read a model, turn a prompt into tokens, evaluate them, and return text or media-aware results. A person using it needs more than that core. They need a build that produces a predictable executable, a folder that keeps the matching model and optional encoders together, a launcher that chooses those files, and an interface that another program can drive.

A useful package is therefore a small, explicit chain rather than one enormous executable. The source tree supplies the binary; the model folder supplies data; the launcher selects defaults; and a caller can use the same chat interface as a terminal user. The voice application sits outside the engine and acts as one such caller.

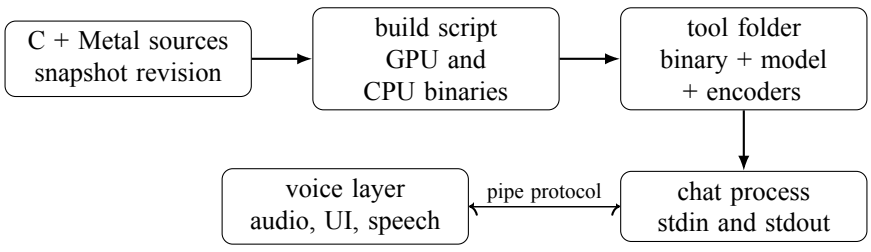


Figure 24.1: The engine is one process in a package. The launcher and the voice layer supply product behavior around it.

PREPRINT

Draft book, October 2026. Not peer reviewed.

The build target and the runtime folder solve different problems. A build target says which source files and frameworks form the executable. The folder supplies runtime inputs such as the GGUF model and, for pictures, sound, or video, an encoder file. Keeping these roles visible makes it easier to update a binary without silently changing which model it opens.

Why build in a known place?

The book's frozen build script anchors its paths to the script's own directory. It creates the build output directory, changes into the snapshot, and writes two named binaries into `code/build/`. The GPU build includes Metal and media support; the CPU build is a reference path without those frameworks. Since Metal kernels are compiled by the program at startup, the build needs the system frameworks but no separate Metal compiler step.

```
#!/bin/sh
# Snapshot-local build. Every artifact goes to the book 'projects code/
    build/.
set -eu
HERE=$(CDPATH= cd -- "$(dirname -- "$0")" && pwd)
OUT="$HERE/../build"
mkdir -p "$OUT"
cd "$HERE"
clang -O3 -march=native -DGEMMA4_METAL -DGEMMA4_MEDIA -o "$OUT/gemma4"
    gemma4.c gemma4_metal.m gemma4_media.m \
  -framework Metal -framework Foundation -framework AVFoundation -
    framework CoreMedia -framework CoreGraphics \
  -lm -lpthread
clang -O3 -march=native -o "$OUT/gemma4_cpu" gemma4.c -lm -lpthread
```

Listing 24.1: `build.sh`:1–11

This short script is also a packaging rule. Its output path is derived from `HERE`, not from whichever directory happened to be current when the user invoked it. The snapshot therefore cannot overwrite a neighboring engine or put a binary beside its source by accident. Chapter 6 explained the runtime Metal compilation; this chapter's concern is making the executable land in a deliberate place.

A distributable folder has a parallel rule: launch relative to the folder containing the launcher. In the project README, `chat-gemma` chooses the E4B model in that folder unless the caller supplied `-m`; it selects the Q8 encoder file if present, otherwise the F16 encoder file, unless the caller overrides or disables it. The wrapper then replaces itself with `gemma4 --chat`. That last step matters: signals and exit status go directly to the process doing inference, and a shell wrapper does not remain as an unnecessary parent.

The README's common controls are not packaging magic. `--think` enables the thinking channel; `-sys` supplies a system prompt; `-c` sets context capacity; and `--kv` selects the cache representation. Media flags choose an input for one-shot use, while the interactive commands attach media to a later turn. Chapter 16 covered the conversation template and cache-prefix reuse; Chapter 19 covered how media entries become model rows. Packaging makes those capabilities easy to select from one folder.

How can a terminal and a voice program share one chat path?

The interactive chat loop reads one line at a time from standard input. It writes generated reply text to standard output and prompts, status, and usage guidance to standard error. This division is a small but important protocol choice: a script can capture the model's answer without accidentally receiving a prompt such as `> ` in the same stream.

The first excerpt shows the interface boundary and the interrupt policy. With a terminal, Ctrl-C at the prompt means quit. When a program drives stdin through a pipe, an interrupt that arrives just after a reply ended should not kill the next idle read; while generation is active, the same signal stops the reply. The

chat process can therefore be controlled by a caller that knows only its process ID and its line-oriented input/output.

```
4399 // Interactive chat on stdin: one line per message; /image, /audio and /
       video FILE attach media to the next
4400 // message (needs --mmproj; /video --no-audio FILE: without its sound),
       /clear starts a new conversation, /exit
4401 // (or end of input) quits. `media` goes with the first message.
4402 static void chat_loop(Chat *c, const Encoders *enc, int N, Parts media)
       {
4403     char *line = NULL;
4404     size_t cap = 0;
4405     fprintf(stderr, "chat: one line per message; /image FILE, /audio
       FILE, /video [--no-audio] FILE, /frame FILE [mm:ss], "
4406             "/prefill, /undo, /clear, /exit%s\n",
4407             media.n ? " (the media go with your first message)" : "");
4408     chat_tty = isatty(STDIN_FILENO);
4409     signal(SIGINT, chat_sigint);
```

Listing 24.2: gemma4.c:4399–4409

The commands are lines in that same input stream, not a second socket API. A media command attaches an item to the next user message. `/prefill` advances the cached prompt with pending media before the text arrives. `/undo` removes pending media or the last exchange and shortens the reusable prefix. `/clear` starts a new conversation and resets the cache count. End of input and `/exit` both end the session.

```
4419         if (!strncmp(t, "/undo", 5)) {  // drops the attached media, or
       else the last exchange
4420             gpu_sync();
4421             if (media.n) chat_forget_pending(c), parts_free(&media),
       fprintf(stderr, "(media dropped)\n");
4422             else if (c->n_msg >= 2 && !c->msg[c->n_msg - 1].user && c->
       msg[c->n_msg - 2].user) {
4423                 // the cache keeps what the conversation without them
       still starts with (cut before their media's
4424                 // rows are freed, whose addresses may come back for new
        ones)
4425                 c->n_msg -= 2;
4426                 Entry *e;
4427                 int n, p = 0;
4428                 chat_render(c, &e, &n);
4429                 while (p < n && p < c->n_cached && same_entry(e[p], c->
       cached[p])) p++;
4430                 c->n_cached = p;
4431                 free(e);
4432                 for (int k = 0; k < 2; k++) parts_free(&c->msg[c->n_msg
       + k].media), free(c->msg[c->n_msg + k].text);
4433                 fprintf(stderr, "(undone)\n");
4434             } else fprintf(stderr, "(nothing to undo)\n");
4435             continue;
```

Listing 24.3: gemma4.c:4419–4435

```
4437         if (!strncmp(t, "/clear", 6)) {
4438             gpu_sync();
4439             for (int i = 0; i < c->n_msg; i++) parts_free(&c->msg[i].
     media), free(c->msg[i].text);
4440             parts_free(&media);
4441             c->n_msg = 0;
4442             c->n_cached = 0;  // (freed rows' addresses may come back
     for new ones: nothing cached may match)
4443             fprintf(stderr, "(new conversation)\n");
4444             continue;
4445         }
4446         if (len == 8 && !strncmp(t, "/prefill", 8)) {
4447             chat_prefill(c, media);
4448             continue;
4449         }
```

Listing 24.4: `gemma4.c:4437–4449`

The voice program uses this interface as a small subprocess protocol. In the source tree outside this snapshot, `omni_chat.py` starts the engine with pipes, sends media and text commands, reads the reply stream, and coordinates stop/undo behavior. It also manages the audio helper, speech synthesis, optional camera or screen input, and the user interface. Those are product responsibilities: they may change without changing the transformer, the tokenizer, or the engine's GGUF reader.

The README describes a useful scheduling detail. During speech synthesis, the voice layer can stop the engine with `SIGSTOP` so the synthesizer has the GPU, then resume it with `SIGCONT`. This is a coordination policy between two processes that share a scarce device. The engine itself remains unaware of microphones, speakers, or UI controls. Keeping that boundary lets a shell script, a text interface, and a voice interface drive the same engine.

Input line	Effect	Protocol use
Ordinary text	Adds a user entry and produces a reply	One line per turn; read reply text from stdout
`/image`, `/audio`, `/video`	Attaches media for the next message	The launcher must have selected an encoder file
`/frame FILE [mm:ss]`	Adds one timestamped video frame	Lets a caller send selected frames instead of a whole clip
`/prefill`	Runs the pending prefix into the KV cache	Can overlap media prefill with the user's remaining speech
`/undo`	Drops pending media or the last exchange	Supports interruption and speculative turns
`/clear`, `/exit`	Resets the conversation or ends the process	Lifecycle controls for an interactive caller
`SIGINT`	Stops active reply generation; idle behavior depends on terminal input	A caller can interrupt without inventing a new engine API

The media attachment commands are handled here:

```
4463         static const char *cmd[] = {"/image", "/audio", "/video"};
4464         int kind = -1;
4465         for (int k = 0; k < 3; k++)
4466             if (!strncmp(t, cmd[k], 6) && (len == 6 || t[6] == ' ' || t
     [6] == '\t')) kind = k;
4467         if (kind >= 0) {
4468             size_t pl;
4469             char *rest = strndup(t + 6, len - 6);
4470             const char *pt = trimmed(rest, &pl);
4471             Encoders e = *enc;
4472             if (kind == MEDIA_VIDEO && pl >= 10 && !strncmp(pt, "--no-
     audio", 10) && (pl == 10 || pt[10] == ' ')) {
```

PREPRINT

```
4473                 e.video_sound = 0;
4474                 pt = trimmed(pt + 10, &pl);  // (pt's end is rest's:
      trimmed again from there)
4475             }
4476             char *path = strndup(pt, pl);
4477             if (!pl) fprintf(stderr, "usage: %s FILE\n", cmd[kind]);
4478             else if (add_media(&e, kind, path, &media))  // (cmd's order
       is MEDIA_*'s)
4479                 fprintf(stderr, "(attached to your next message)\n");
4480             free(path);
4481             free(rest);
4482             continue;
4483         }
```

Listing 24.5: `gemma4.c:4463–4483`

This table describes the command handling in the frozen source and README, not every option accepted by every launcher. The line protocol has a narrow purpose: it carries user text, media references, and a few conversation controls. It does not make the engine responsible for audio device routing or the voice application's presentation.

What belongs in the engine, and what belongs around it?

A product folder may include the engine, model, optional encoder, launcher, audio helper, and voice model, but that does not make them one software layer. The engine owns inference, the chat template, supported media decoding, and a stable stdin/stdout contract. A launcher supplies paths and defaults. A voice layer handles capture, turn-taking, speech generation, and scheduling. A Mac app provides buttons, permissions, and a transcript window.

The project README describes `Omni Chat.app` as a window around the voice command, with Voice, Screen, and Camera modes. It also says the app is signed locally with an ad hoc signature, so macOS asks again for permissions after a rebuild. This is a local development behavior, not a promise that the app is ready for distribution. A release workflow would separately decide identity signing, notarization, installer layout, license notices, model download instructions, and update policy. Those decisions do not belong inside the model loop.

The package must also distinguish executable code from model weights. The model and encoder files are large runtime inputs and are not included in this book's source snapshot. Their terms and redistribution rights need to be resolved separately. A folder-oriented launcher can tell a user where to place a model without embedding that model in the program or the printed book.

What should a staged folder contain?

A staging folder is a reviewable version of the tool before anyone installs it. It should contain only the files needed for the chosen mode: the engine executable, its launcher, the selected model file, an optional encoder file, and the small support files the launcher actually reads. A text-only package can omit the encoders and voice system. A voice package may add the audio helper, Python environment instructions, speech model, and app, but those additions should be explicit instead of silently inherited from the developer's machine.

Before making an archive, inspect the folder by role. The executable should be the expected architecture and build; the launcher should resolve paths from its own location; the model and encoder should match their recorded sizes and hashes; and the directory should not contain caches, temporary screenshots, logs, or a second unintended model variant. Keep the license notices and third-party attributions beside the release metadata. A model's terms are separate from the engine's code license: a script that fetches weights does not grant permission to redistribute them.

PREPRINT

A small manifest makes this check practical. It can record the engine revision, build command and compiler version; hashes of the executable, launcher, model and encoder; the default model/encoder selection; the minimum macOS version; and any required frameworks or Python packages. Hashes identify bytes, while the notes identify intended use. A hash alone cannot tell a reader whether an encoder is optional or whether a model is licensed for their use.

Package item	What to record	What the check establishes
Engine binary	Hash, architecture, build revision	Which executable was staged
Launcher	Hash and default path rules	Which model and encoder it will select
Model and encoder	Filename, size, hash, terms location	Which runtime bytes are expected
Voice/app extras	Versions and runtime requirements	Whether the voice path has its dependencies
Release notes	Supported modes and known limits	What a user can expect from this package

A clean stage also supports rollback. If an update changes only the binary, the manifest makes that visible; if it replaces the model or encoder too, the change is explicit. Preserve the previous folder until the new one passes its checks. This is especially useful for a local assistant: a broken launcher or missing encoder can disrupt the user experience even if the model computation is unchanged.

What changes between a local build and a distributed app?

For development, the snapshot's command-line binary is enough. A Mac app adds a bundle structure, an identifier, icons and permission descriptions. The project README says its current app uses an ad hoc signature, which identifies the local build but is not a Developer ID distribution signature. Rebuilding changes the app bytes, and macOS treats it as a new local app for permission purposes.

For a distributed app, the usual sequence is to assemble the bundle, sign nested helpers and the app with a Developer ID certificate, verify the signature, submit the build for notarisation, staple the returned ticket, and package the result for installation. These are separate steps: a valid signature says who signed the bytes and whether they changed; notarisation is Apple's review service; a package installer lays out files and can run installation scripts. Neither a signature nor an installer resolves model licensing or supplies weights automatically.

The following commands are release-plan examples only. The placeholders are deliberately not real identity or account values, and none of these commands is run for the book. The first signs an application using a distribution identity; the next two represent Apple's notarisation and a local installer build. Submission to notarisation sends the artifact outside the Mac, so it requires a separate decision and is outside this local packaging task.

```
codesign --force --deep --options runtime \
  --sign "Developer ID Application: NAME" \
  "stage/Omni Chat.app"
codesign --verify --deep --strict --verbose=2 \
  "stage/Omni Chat.app"
xcrun notarytool submit "OmniChat.zip" \
  --keychain-profile "PROFILE" --wait
pkgbuild --root stage --identifier com.example.gemma4 \
  --version 1.0 --install-location /Applications "Gemma4.pkg"
```

An actual release would need to decide where user data and large weights live, how updates preserve them, and how the application reports a missing or mismatched model. The folder-based command-line package avoids pretending those policies are already solved. A local app can point at a user-managed model folder; an installer should not copy several gigabytes of weights without an explicit, licensed download workflow.

PREPRINT

What can a caller rely on across the pipe?

A process boundary is a useful seam only when both sides agree on what crosses it. The current chat loop accepts one line per user turn, and the engine flushes generated text as tokens arrive. Diagnostics and prompts go to standard error. End of input exits the loop. The same command can therefore be used interactively in a terminal or driven by a parent process, but those two modes do not render every token in exactly the same way.

The code comment at `gemma4.c:4235--4257` calls out one such difference: on a terminal, the thinking channel is styled and its markers are hidden; through a pipe, the reply is printed as it is. A caller that enables thinking should not assume the pipe contains only user-facing prose. It should either keep thinking off for that integration or explicitly handle the channel representation. This illustrates why a protocol test should preserve flags as well as prompt text.

A parent process should also distinguish three outcomes: the model completed its reply, the user interrupted generation, or the process failed before a reply was complete. The current interface streams text and accepts SIGINT to stop a running reply; a higher-level API could make completion status more explicit, but adding such an API is a design change, not a wrapper-only formatting tweak. Until then, keep the child exit status and stderr log with captured stdout, and avoid treating a truncated stream as a complete answer.

Boundary	Current behavior	Caller practice
Input	One line is a message or a slash command; EOF exits	Send complete lines and close stdin when finished
Output	Tokens stream on stdout; prompt and status use stderr	Capture the two streams separately
Thinking	Terminal formatting differs from pipe output	Fix the flag and parse the expected representation
Interrupt	SIGINT stops active generation; idle handling depends on TTY	Send it only to the child and retain partial output as partial
Timeout	The engine does not impose a wrapper's product timeout	Supervise the child and reap it on timeout

A wrapper should not parse a display prompt or a debug line as model text. If an integration needs a stable machine-readable format, JSON or an event protocol is a reasonable extension, but its escaping, partial-reply rule, and error status need to be specified first. Exercise 3 asks for that design. The existing line protocol is enough for the voice stack described here because it controls a local child process and already knows how to interpret its streaming output.

What does it mean to vendor the engine?

Claude Omni keeps a copy of the engine under its vendor directory so that the voice application has a known local dependency. A vendor copy is useful only if its identity is clear. Source equality can be checked by comparing file hashes; for a binary or a scripted chat path, compare the built artifacts or the bytes produced for fixed inputs. A passing comparison belongs to a named revision and exact inputs, not to an informal claim that two directories "look the same."

The outline records the existing integration gate: six fixed prompts produced byte-identical output, and the `e4b-compare` report is dated 2026-10-21. That is historical evidence from the project's prior check, not a new run for this chapter. The frozen snapshot manifest provides the repeatable source identity for this book. The rule is simple: record the revision, build from that revision, compare the intended files or outputs, and retain the result beside the command. A changed source hash means the comparison must be repeated before treating the vendor copy as equivalent.

What can we check without opening a model?

A W0 build check establishes that the snapshot still compiles into its two expected outputs and that the GPU binary links the intended system frameworks. Its `--help` output can confirm that command-line

PREPRINT

controls are present. Inspecting the bundle or executable for embedded resources can establish whether the model has accidentally been copied into the build output. None of these checks establishes that a model loads or that a reply is correct.

The full integration experiment in the outline is W2 because it starts the model and asks it to answer through a pipe. In a scratch copy with licensed local inputs, build two revisions from their own source trees and run the same three-turn input against both. Save stdout separately from stderr and diff the replies. The existing six-prompt gate is a useful wider follow-up. Record binary and model identities, prompt bytes, options, date, and machine. **Try it on your machine (W2):** compare stdout replies and stderr diagnostics for the same three-turn input across the two revisions, then extend to the six-prompt gate. Semantics-preserving revisions should retain matching replies; no dated result for these revisions is cited.

```
printf '%s\n' 'Hello.' 'What did I just say?' 'Name one color.' |
  nice -n 10 code/build/gemma4 \
    -m "$MODEL" --chat -n 32 \
    > reply.txt 2> engine.log
```

For a repeatable output comparison, keep protocol output and diagnostics separate. The sample is a W2 recipe only: this chapter did not run it, and no expected reply text is predicted. A mismatch can come from model bytes, executable bytes, options, or prompt framing; the logs and input hashes make those causes distinguishable. A byte-identical output is evidence for those inputs, not proof that every possible chat behaves identically.

What packaging failure did the project actually encounter?

> **Pitfall — the audio product boundary can hide a routing bug.** Symptom: with Bluetooth as the default audio route, music in the headphones became mono and thin, and sometimes the voice reply did not play. Cause: the voice-processing path was being applied when the default devices changed to Bluetooth. Fix: the README's later behavior uses voice processing only when the required route is available; with a Bluetooth default route it runs without that processing, and the app reopens audio when the route changes. This was recorded as fixed by 2026-10-04 in the snapshot README, in the echo-cancellation discussion. Keep device policy in the audio product layer and test route changes there; an inference test cannot catch this class of failure.

The broader lesson is that a package crosses boundaries the engine never sees. The model can return the right text while the audio output is silent, the microphone route changes, or the UI continues to display an interrupted answer. Diagnose the layer that owns the failing behavior, and use the engine's pipe contract as the seam between them.

Exercises

1. Trace the snapshot build script's path from its own location to the two output binaries. What changes if the script instead writes `./gemma4` after changing into the snapshot?
2. Write a shell wrapper that adds a 10-second timeout to a one-turn chat command. Decide what it should do with the child process and its stderr when the timeout expires.
3. Design a `--json` output mode for a script caller. Specify how ordinary reply text, errors, and interruption status would be represented without mixing diagnostics into stdout.
4. For a three-turn pipe test, choose a model, prompt lines, context, output cap, and output files. List which byte identities and environmental conditions you would preserve to compare two builds fairly.
5. Draw the boundary between engine, launcher, voice layer, and Mac app. Assign media-file selection, microphone capture, prompt rendering, GPU scheduling during synthesis, and permission prompts to the layer that owns each.

PREPRINT

Final checkpoint: ship a traceable tool

The smallest packaging checkpoint is a folder with a named build, an explicit model path, and a shell caller that can send three lines and capture replies separately from diagnostics. Add a manifest containing the source revision and hashes of the executable and runtime files. For a model-backed acceptance gate, compare the same fixed input across the candidate and vendored builds and retain the raw outputs. Do not describe a package as equivalent because the launchers have similar names; make the comparison reproducible.

This closes the path followed by the book: source and weights become a forward pass, the forward pass becomes a chat protocol, and the protocol becomes one component in a tool. The model engine stays small enough to test on its own, while the product around it can grow to include encoders, voice, and a window.

PREPRINT

Appendices

PREPRINT

Appendix A

Weight formats in full

How many bytes describe a row?

Figure 4.3 introduces a quantized block as a small shared scale followed by compact integer codes. This appendix is the byte-level lookup for the formats the snapshot decodes. It answers three separate questions: how many values one block covers, where each field sits, and how to expand its codes back to weights. The layouts are the ones consumed by `dequant_row` in `code/snapshot/gemma4.c`; the GGUF type number identifies the representation, while a tensor's shape determines how many complete blocks occur in each row.

In the maps below, byte zero is the start of a block or super-block. A 16-bit field is stored in the file's native little-endian representation. The pictures describe serialized bytes, not the arrangement a GPU lane must use. The decoder may load a field, unpack its bits, and rearrange values before a dot product. See Chapter 4 for why accumulation and stored precision are separate decisions.

Table A.1: Block formats decoded by the snapshot. Bytes per value are a storage average, not an accuracy score.

Type	Values	Bytes	Bits/value	Typical decoding rule	Code
Q8_0	32	34	8.50	dq	8
Q4_0	32	18	4.50	$d(q-8)$	2
Q4_1	32	20	5.00	$dq+m$	3
Q4_K	256	144	4.50	$dsq-d_{min}m$	12
Q5_K	256	176	5.50	Q4_K plus high bits	13
Q6_K	256	210	6.5625	signed 6-bit levels	14
IQ4_NL	32	18	4.50	$d\,L[q]$	20
IQ4_XS	256	136	4.25	$d\,s\,L[q]$	23
BF16	1	2	16	top 16 bits of F32	30
F16	1	2	16	IEEE binary16	1

The average includes scales and packed metadata. For example, Q4_0 stores 32 four-bit codes (16 bytes) plus one two-byte scale: 18 bytes total, or 4.5 bits per value. Q4_K stores 256 four-bit codes (128 bytes) and 16 bytes of scales/minima plus two two-byte super-scales, for 144 bytes. Its nominal four-bit codes therefore average 4.5 bits on disk. Q5_K adds 32 bytes of high-bit flags; Q6_K uses 128 low-bit bytes, 64 high-bit bytes, 16 signed subscales, and a two-byte scale. These totals explain why a format name alone is not its exact storage rate.

What is in a 32-value block?

The first four layouts are easiest to read as a header followed by codes. Q8_0 stores signed bytes, while Q4_0 stores two unsigned nibbles per byte. In the Q4_0 map, low nibbles represent positions 0–15

PREPRINT

and high nibbles positions 16–31; the decoder subtracts eight from each code before applying the half-precision scale. Q4_1 instead has a scale and an additive minimum, so its nibble codes run from zero upward without the Q4_0 offset. The drawings show logical field order and exact byte counts.

Table A.2: Serialized maps for the simple 32-value blocks.

Type	Byte offsets	Meaning
Q8_0	0–1 `d`; 2–33 `q[32]`	$w_i = d\,q_i$, signed 8-bit codes
Q4_0	0–1 `d`; 2–17 `qs[16]`	low/high nibbles, two groups of 16
Q4_1	0–1 `d`; 2–3 `m`; 4–19 `qs[16]`	$w_i = d\,q_i + m$
IQ4_NL	0–1 `d`; 2–17 `qs[16]`	nibbles index a fixed non-linear level table

In IQ4_NL, the 16 possible nibble values select the ordered level table in the C source. The table is non-uniform: nearby entries are not equally spaced. This lets a four-bit codebook spend more representational detail around common levels. The same level table is used by IQ4_XS, but IQ4_XS gives each 32-value subgroup its own signed scale inside a larger 256-value unit. Do not interpret either format's nibbles as ordinary signed integers.

The Q4_0 and Q8_0 worked examples and their CPU/Metal decoding paths are in Chapter 4; the serialized Q4_0 layout is also shown in Figure 4.3. Chapter 4.5 gives the Q4_K field map. These references are useful when tracing one actual byte: first identify the super-block, then its 32-value subgroup, then the code position within that subgroup. The tensor row length must be divisible by 32 for these decoders, and the super-block formats group eight such subgroups.

Why do K-quants use super-blocks?

K-quants amortize a global scale over 256 values while allowing smaller local scales to follow changes in the row. The 256-value organization is always eight consecutive 32-value decoding units. For Q4_K and Q5_K the first four bytes are the half-precision global scale d and minimum scale d_{min}. The next 12 bytes encode eight six-bit scale values and eight six-bit minimum values. The remaining fields hold the quantized codes; Q5_K inserts its high-bit field before the codes.

Table A.3: K-quant super-block offsets in the implementation.

Type	Header and local scales	Packed codes	Decoder's subgroup
Q4_K	0–3: d, d_{min}; 4–15: packed scales/minima	16–143: 128 code bytes	16 bytes per pair of subgroups
Q5_K	0–3: d, d_{min}; 4–15: packed scales/minima	16–47: qh; 48–175: qs	32 high-bit bytes plus 128 low nibbles
Q6_K	0–127: ql; 128–191: qh; 192–207: signed scales; 208–209: d	low/high bit fields	32 values per subgroup

For Q4_K and Q5_K, each subgroup selects one six-bit scale and one six-bit minimum. The stored nibbles provide the base level; Q5_K takes a fifth bit from the matching bit of `qh`. For Q6_K, each code combines four low bits from `ql` and two high bits from `qh`. The result is centered by subtracting 32 and multiplied by the subgroup's signed scale and the super-block half scale. A common source-reading mistake is to treat `ql` as 256 sequential nibbles; its byte grouping is arranged so the four 32-value subgroups can be decoded efficiently.

The scale/minimum packing in Q4_K and Q5_K is not a byte-per-scale table. Some six-bit values cross byte boundaries; the implementation uses separate cases for subgroup indices below and above four. Chapter 4's worked decode follows one subgroup through those shifts. A robust independent reader

PREPRINT

should check both halves of the subgroup range, because testing only the first four never exercises the alternate bit extraction.

What is special about IQ4_XS?

IQ4_XS also covers 256 values in 136 bytes. Bytes 0–1 hold the half scale d; bytes 2–3 hold a packed 16-bit high-scale field; bytes 4–7 hold eight low four-bit scale fragments; bytes 8–135 hold 128 nibble-code bytes. For subgroup b, two high bits come from the packed field and four low bits from one nibble in the scale bytes. These form a signed six-bit value by subtracting 32. That subgroup scale multiplies the shared half scale, then each nibble indexes the 16-entry non-linear table.

The apparent paradox is useful: IQ4_XS stores only 4.25 bits per value on average yet has a scale for every 32-value group. Its global scale is shared, while subgroup scales are compactly packed. “4-bit” describes the codes, not every field in the block. Figure 4.7 in Chapter 4 shows the byte arrangement and the chapter compares the CPU unpacking with its Metal twin.

Which two-byte numbers are not the same?

F16 and BF16 each occupy two bytes per value, but they divide those bits differently. F16 has one sign bit, five exponent bits, and ten fraction bits. BF16 has one sign bit, eight exponent bits, and seven fraction bits; it is the high half of an IEEE single-precision bit pattern. BF16 therefore keeps approximately the F32 exponent range but rounds more coarsely, while F16 gives finer significand detail across a smaller range. In the snapshot, F16 is converted by `half_to_float`; BF16 is expanded by shifting its 16 bits into the upper half of a 32-bit word.

The type code is not an instruction to accumulate in that type. A tensor can be stored in a compact format, expanded to float for the CPU reference, or consumed in a Metal kernel that decodes blocks while accumulating. These choices are covered in Chapters 4, 5, and 7. The table here is a storage reference: it does not promise bit-identical results across implementations or replace the chapter’s tests against the engine’s decoder.

A compact audit recipe

When an unfamiliar tensor appears in a directory listing, record its GGUF type, row length, number of rows, and byte size before interpreting its contents. Divide the row length by 32 to count decoding units, then group eight units for a super-block. Multiply the relevant block count by the format’s byte size and compare with the declared tensor extent. A mismatch usually means a shape orientation error, a wrong block size, or a tensor whose row width is not aligned to the format. Chapter 3 explains the tensor directory and row orientation; this appendix supplies the block constants that make the size calculation possible.

The layouts above are for the types this snapshot’s row decoder handles. They are not a complete catalogue of every GGUF or ggml type. In particular, a reader should reject an unknown type rather than guess its layout from a similar name. The engine’s supported type codes are enumerated beside its tensor reader; Chapter 3 explains how unsupported types fail safely. For the accuracy consequences of choosing among these formats, see Chapters 4, 20, and 22.

Format cards for source reading

The table below gives a compact cue for each family. It is deliberately about decoding structure, not a ranking of quality: the same nominal bit rate can behave differently because one code is uniform, another uses a non-linear lookup table, and another spends extra bits on local scale metadata. Chapter 4 discusses the accuracy side and the corresponding experiment.

PREPRINT

Q8_0.. Each 34-byte block begins with a two-byte half scale and continues with 32 signed bytes. The scale multiplies every signed code. Its 8.5-bit average includes the scale overhead. It is a convenient simple decoder and a useful reference for small encoder matrices, but the name does not mean exactly eight stored bits per weight.
Q4_0.. Each 18-byte block begins with the half scale followed by 16 packed bytes. Each byte carries two four-bit values. The low nibble contributes to the first half of the output sequence; the high nibble contributes to the second half. Subtract eight, then multiply by the shared scale. Figure 4.3 illustrates the ordering that is easy to reverse accidentally.
Q4_1.. The same 16 code bytes are preceded by two half values, a scale and a minimum. Each code is unsigned and the minimum is added after scaling. Its extra two bytes cost 0.5 bit per value compared with Q4_0 for a block of 32.
Q4_K.. A 256-value super-block has 128 code bytes and 16 bytes of packed local scales/minima, in addition to the two global half values. Eight groups of 32 values each have their own six-bit scale and six-bit minimum. Codes are four bits; the minimum contribution is subtracted from the scaled code.
Q5_K.. This extends the Q4_K code field with one high bit per value. The implementation stores those bits as 32 bytes, then 128 low-nibble bytes. The 16-byte local-scale table and four-byte global header stay the same as Q4_K, giving 176 bytes per 256 values.
Q6_K.. This uses 128 bytes of low bits and 64 bytes of high bits, then 16 signed local scales and a two-byte global scale. The decoder assembles a six-bit code, centers it around 32, and applies both scales. The field order is not the same as a flat sequence of six-bit integers.
IQ4_NL.. One half scale plus 16 packed bytes cover 32 values. Every nibble is an index into a fixed 16-level table, so the code values are not evenly spaced integer steps.
IQ4_XS.. One global half scale, a packed high-scale field, four bytes of low-scale nibbles, and 128 code bytes cover 256 values. Each of eight groups has its own signed six-bit scale assembled from the high and low fields. Codes then index the same non-linear level table used by IQ4_NL.
BF16 and F16.. These are scalar formats rather than blocks: each value occupies two bytes. The interpretation differs at the bit level even though the storage size matches.

A row-size check with a real model width

Chapter 7 uses a row length of 2,560 values for a common E4B projection. That width is 80 groups of 32, or 10 groups of 256. Multiplying by the block sizes gives Q4_0: $80 \times 18 = 1,440$ bytes; Q4_1: $80 \times 20 = 1,600$ bytes; Q4_K: $10 \times 144 = 1,440$ bytes; and IQ4_XS: $10 \times 136 = 1,360$ bytes. These are storage bytes for one row, before any alignment between tensor regions. They match the row-size comparison in Chapter 7 and show why the label “4-bit” is not enough to compute traffic.

The same calculation should be repeated for the actual tensor rather than copied from this example. Read `ne[0]` as the row length and `ne[1]` as the number of rows, as Chapter 3 explains. Calculate bytes per row from complete blocks, then multiply by row count and, for a three-dimensional tensor, by the expert or slice count. Compare the result against the tensor directory’s next offset or the reported size. A row-size calculation can be correct while the total is wrong if a dimension was interpreted as rows instead of input features.

Byte-order and boundary checklist

All packed fields here are read from a little-endian GGUF on the supported platform. The snapshot uses byte-wise accesses and copies half fields into 16-bit integers before looking them up in its conversion table. A portable parser must make its endianness assumptions explicit; a cast to a wider pointer can also introduce alignment assumptions that a byte-level description does not imply.

Before trusting a decoder, check at least one block boundary and one super-block boundary. For a 32-value block, test both the first and second nibble halves. For K-quants, test subgroup indices on both sides of the branch that changes how scale bits are assembled. For Q5_K, verify the high-bit extraction

DRAFT

as well as the low nibbles. For Q6_K, verify the selected quarter and half determine both the low-bit address and the two-bit shift. For IQ4_XS, verify the scale nibble and two high bits produce the intended signed local scale. These are structural checks, not a request to invent expected decoded values; use the engine's own checked rows where the W1 workflow provides them.

Finally, separate three validation levels. A byte-count check proves that the declared shape fits the stored tensor extent. A hand-worked block proves that a particular field is unpacked as intended. A comparison against the engine's decoder checks the implementation across actual rows. None alone proves model-level output agreement, which is why the book keeps block decoding, inference accuracy, and performance in separate chapters.

PREPRINT

Subgroup locator for 256-value blocks

For a 256-value block, divide the local value index by 32. The quotient selects one of the eight 32-value groups, and the remainder selects the value within it. This is a useful conceptual locator even when the serialized codes are interleaved. The table describes which pieces of metadata the decoder consults; it intentionally does not replace the exact bit expressions in Chapter 4.

Table A.4: Metadata consulted for one 32-value subgroup of each super-block format.

Format	Group-local selection	Information combined
Q4_K	One of eight subgroup indices	Global half scale; local six-bit scale; local six-bit minimum; four-bit code
Q5_K	Same group index as Q4_K	Same global/local scales; four low code bits plus one high bit
Q6_K	Group selects quarter and half	Global half scale; two signed local scales; four low bits plus two high bits
IQ4_XS	One of eight scale entries and code spans	Global half scale; packed signed local scale; non-linear level-table index

The value index and the packed-byte index are not interchangeable. Q4_K stores two groups' nibbles in a 32-byte span; selecting the lower or upper subgroup changes whether the decoder reads the low or high nibble. Q5_K follows the same low-nibble pattern and adds a bit from the corresponding high-bit word. Q6_K organizes four groups within each half of a super-block; the code's quarter selector affects both the low-byte address and the shift used on the high-byte field. IQ4_XS uses 16 bytes for each subgroup's two 16-value halves.

When reading the implementation, write down the group index, then substitute it into every field address and shift expression. This mechanical process is less error-prone than trying to recognize a packed format from its name. At a boundary between groups, check that the final value of one group and the first value of the next select distinct intended code positions. At a boundary between super-blocks, check that the base pointer advances by the full serialized size: 144, 176, 210, or 136 bytes for the formats in this table.

The storage calculation and the address calculation should agree. If one group contains 32 values, eight group steps cover 256 values. The total code bytes plus the header and local metadata must equal the super-block size in Appendix A. This cross-check catches an offset that accidentally treats one subgroup as a whole block or one byte of high bits as a separate scale. Use the production decoder comparison when the hand inspection has isolated the field; use model-level evaluation only after the row decoder itself agrees.

PREPRINT

Appendix B

Command line and environment reference

Which path does a command select?

The command line exposes several related jobs through one executable: continuation, scoring, benchmarking, interactive chat, and encoder-only work. The options below follow the snapshot's `usage()` at `gemma4.c:4559`; Chapter 15 explains greedy generation, and Chapter 14 follows the work performed by a forward pass. A model path is required for language-model work. Encoder-only use is the exception: a multimodal projection file and media input can be supplied without `-m`.

Table B.1: Language-model and evaluation options.

Option	Value or default	Meaning and interaction
`-m FILE`	GGUF path	Select the language model. Required for generation, scoring, benchmark, and chat.
`-p TEXT`	prompt string	Continue plain text, unless chat mode is selected. In chat it supplies one user turn.
`-n N`	128 plain; 1024 chat reply	Maximum generated tokens. In chat this is the reply limit.
`-f FILE`	text file	Tokenize a corpus for perplexity or benchmark.
`-c N`	512 scoring; chat 4096/16384	Context size. Chat defaults to 4096, or 16384 for E2B/E4B with an encoder file; a base comparison can read its context from that file.
`--chunks N`	workload choice	Limit scoring to a number of chunks. Each perplexity chunk starts with BOS.
`--bench P N`	prompt and decode lengths	Benchmark prefill/TTFT on the first P tokens of the input, then N greedy decode steps. Requires `-f`.
`-r N`	5	Repetitions of each benchmark test.
`--depth D`	0 unless supplied	Benchmark after D tokens of context; the depth prefix is run once.
`--tokens`	flag	Print token IDs for the prompt or corpus and exit; it does not generate.
`--kl-divergence-base FILE`	base file	Save base log probabilities with scoring, or read a base file for comparison. The format is compatible with llama.cpp.
`--kl-divergence`	flag	Compare the current model against the supplied base file. This comparison mode uses token positions from that file.

What does chat add?

In interactive chat, the program recognizes `/image FILE`, `/audio FILE`, `/video FILE`, `/clear`, and `/exit`. The exact available media operations depend on how the executable was built and on which encoder file is used. Chapters 16–19 explain how chat rendering, media rows, and prompt caching work; these flags only select those paths. A plain `-p` without chat mode is continuation text and does not parse marker-looking strings as a chat protocol.

PREPRINT

Table B.2: Chat and media options.

Option	Value	Meaning and interaction
`--chat`	flag	Render Gemma 4 chat messages. With `-p`, answer one turn; otherwise read one message per input line.
`-sys TEXT`	system message	Add a system prompt and select chat behavior.
`--think`	flag; off by default	Enable thinking mode and chat rendering.
`--mmproj FILE`	encoder GGUF	Load vision weights; E2B/E4B encoder files also enable audio and video paths described by the build. An encoder file implies chat in the implementation.
`--image FILE`	image path	Attach an image to the first message; repeat to attach multiple items. Supported image formats are listed by `usage()`.
`--audio FILE`	audio path	Attach audio to the first message. WAV is available in the base build; additional formats require the media build.
`--video FILE`	video path	Attach video when built with media support. The command help caps the sample at 32 frames, at most one per second, with audio included by default.
`--image-tokens N`	default 280; range 70–1120	Set the image token budget. A larger budget retains more detail at higher encoder and prefill cost.
`--no-video-audio`	flag	Keep video frames and omit their soundtrack.
`--quantize OUT`	output path	With `--mmproj FILE`, write a Q8_0-quantized copy of the encoder tensors.

How are cache, threads, and backend controlled?

Table B.3: Runtime controls and their scope.

Option	Value	Scope
`-t N`	thread count	CPU worker pool size. Affects CPU work; it does not select a Metal threadgroup size.
`-ngl 0`	zero GPU layers	Force the CPU route in a Metal-capable binary. The CPU-only build needs no such switch.
`--kv q8`	default	Use the 8-bit KV cache representation; it uses less memory than F16 and is an approximation.
`--kv f16`	alternative	Use half-precision key/value storage. Useful as a comparison reference; see Chapters 11 and 20.
`--cache GB`	55% expert share, capped at all but 9 GB	Metal expert-cache budget. If memory requires it, the default reduces the cache to leave the reserve; it is not a KV-cache setting.

The command parser also recognizes `--cache` even though the E4B-oriented command list in the chapter outline omits it. Its description in the snapshot and in `README-26B.md` is for the routed-expert cache used by the 26B MoE path. Do not confuse that with `--kv`, which selects how attention state is stored. `-ngl 0` is a backend choice, not a quantization setting. Chapter 20 uses the CPU path as a correctness comparison; Chapter 21 explains why benchmarking options must be reported with their run conditions.

Which environment variables are diagnostics?

Environment variables below are read by the frozen source. They are opt-in diagnostics or behavior switches; the default run leaves them unset. Several can intentionally slow execution or produce output that is not a normal generation result.

`GEMMA4_PROFILE` and `GEMMA4_TIMING` answer different questions. Profiling isolates kernels in separate command buffers; the extra boundaries change the schedule. Timing reports wait intervals in the normal benchmark path. Neither should be substituted for the wall-time benchmark in a performance claim. `GEMMA4_SKIP` is not a way to make a valid model run faster: skipping arithmetic makes its outputs invalid. The only documented use is to estimate the time of the remaining kernels. Chapter 21 gives the measurement protocol and cautions about shared GPU load.

PREPRINT

Table B.4: Environment variables documented by the snapshot.

Name	Typical value	Effect
GEMMA4_TIMING	1	In benchmark mode, report CPU wait time for expert reads and the GPU.
GEMMA4_PROFILE	1	Run kernel steps separately and report GPU time by kernel and shape. This changes scheduling and is for timing inspection.
GEMMA4_SKIP	comma-separated kernel names	Omit named kernels. Results are deliberately wrong; use only to time what remains.
GEMMA4_BENCH_COOL	seconds	Pause before benchmark tests, allowing a fanless machine to cool.
GEMMA4_PPL_DECODE	1	Score the latter half of each perplexity chunk one token at a time, to exercise decode kernels.
GEMMA4_DUMP_EMBD	output path	Save encoder rows in the compatible debug-embedding format. Encoder work can run without a language model.
GEMMA4_VISION_GELU_QUICK	1	Use the quick GELU variant in vision for comparison with llama.cpp. This changes encoder outputs.

Which build switches change the executable?

The snapshot has a plain C build and optional macOS backends. GEMMA4_METAL compiles the Metal interface and links Metal and Foundation; kernels are compiled at runtime from the bundled source. GEMMA4_MEDIA adds the AVFoundation media decoder and links the media frameworks. Without it, the C path supports WAV audio only. G4_MAXX enables diagnostic maximum-value tracking in selected feed-forward paths; it is a compile-time investigation aid, not a runtime user option.

The source header gives minimal compiler command lines, while Chapter 6 explains the host/device boundary and Chapter 24 covers packaging. Keep compiler definitions distinct from environment variables: definitions are fixed when building, whereas runtime variables alter a process after it starts. The snapshot's build workflow writes outputs into the book's build directory; do not use the live engine's build instructions for this reference.

A few complete command shapes

```
# Plain greedy continuation
gemma4 -m model.gguf -p "A short prompt" -n 32

# CPU reference path in a Metal build
gemma4 -m model.gguf -p "A short prompt" -n 8 -ngl 0

# Token IDs only; no forward pass
gemma4 -m model.gguf -p "Hello" --tokens

# Perplexity, then save a base for a later comparison
gemma4 -m model.gguf -f corpus.txt -c 512 --chunks 4 \\
  --kl-divergence-base base.kld

# Benchmark a prompt and greedy decode continuation
GEMMA4_TIMING=1 gemma4 -m model.gguf -f corpus.txt \\
  --bench 512 64 -r 5 --depth 512

# Chat with an encoder file and an attached image
gemma4 -m model.gguf --mmproj mmproj.gguf \\
  --image photo.jpg --chat -p "Describe the picture."
```

These are shapes, not captured experiment results. Replace file names with local inputs and build the

PREPRINT

executable from the snapshot. In particular, the benchmark example is meaningful only when its corpus, model quantization, cache type, context depth, repetition count, machine state, and timing statistic are recorded. The chapter experiments specify which of those controls are relevant to each comparison. This appendix is intended for option lookup; Chapters 20–22 explain how to decide whether a result is correct and repeatable.

A mode matrix for option combinations

Several options share a spelling across multiple jobs, but their surrounding mode determines what they do. Use the matrix as a first check when a command appears to ignore an argument.

Table B.5: Primary command modes and their required inputs.

Mode	Typical selectors	What the process does
Continuation	`-m`, `-p`, optional `-n`	Tokenizes plain prompt text and greedily generates a continuation. Without chat selection, marker-looking strings are not a rendered conversation.
Token inspection	`-m` plus `-p` or `-f`, and `--tokens`	Prints input token IDs and exits; it does not evaluate model layers.
Perplexity	`-m`, `-f`, optional `-c`, `--chunks`	Scores corpus positions and reports aggregate probability statistics.
Base creation	Perplexity inputs plus `--kl-divergence-base FILE`	Saves per-position base log probabilities for a later comparison.
Base comparison	`-m`, `--kl-divergence-base FILE`, `--kl-divergence`	Scores the token sequence represented by the base file and compares distributions.
Benchmark	`-m`, `-f`, `--bench P N`	Measures prompt processing and greedy decode under the requested benchmark controls.
Chat	`--chat` or implied chat selectors	Renders the conversation template, maintains cache state, and optionally attaches media.
Encoder-only	`--mmproj`, media input, no `-m`	Runs the selected encoder and can save its rows with `GEMMA4_DUMP_EMBD`.

The base comparison is especially easy to misread. Saving a base and comparing against one are distinct runs. The comparison file supplies the token positions; it is not a replacement for selecting the current model. To compare two model paths fairly, retain the same base, corpus identity, context and chunking choices. Chapter 20 discusses the meaning and limits of KLD and same-top agreement.

Common command questions

Does `--tokens` run the model?. No. It exits after tokenization. This makes it a low-cost inspection mode, but it cannot answer questions about logits, generation, attention, or model speed.

Does `-p` always use chat formatting?. No. Plain continuation is the default path. Chat formatting is selected by `--chat`, or by selectors such as a system prompt, thinking, or an encoder path as the parser implements them.

Does `--mmproj` mean image only?. No. The E2B/E4B encoder file supports the audio path as well; video additionally depends on the media build. The accepted containers are identified by the usage text, and the decoder availability depends on build flags.

Does `--cache` choose KV precision?. No. It controls the 26B expert cache. Use `--kv q8` or `--kv f16` for the attention KV representation.

Does `GEMMA4_PROFILE=1` give the usual end-to-end time?. No. Each step is run alone to measure kernels, introducing boundaries that change execution. Use benchmark wall time for end-to-end speed and treat profile output as diagnostic.

Can a skipped kernel be used for an answer?. No. `GEMMA4_SKIP` deliberately removes required arithmetic. The resulting output is not a valid model result.

Can I infer the media decoder from the filename?. No. The executable must include the matching build support, and the encoder GGUF must contain the required model. The source build switches and Chapter 19 explain this boundary.

PREPRINT

Preserve the command as part of the result

A useful benchmark record contains more than the executable and a headline rate. Save the complete command, model identity and quantization, input text identity, context and chunking, cache type, repetition setting, date, machine, background load gate, statistic, and whether the number is wall or GPU time. If the run includes media, name the encoder file and input asset too. This is not clerical overhead: changing any of those can select a different code path or produce a different working set.

For a correctness comparison, save the base-file provenance, tokenization mode, selected KV format, and the first mismatch if one appears. For a profile, identify whether profiling altered the command-buffer schedule. For a chat capture, include whether thinking was enabled and whether the prompt was plain text or templated. These small details make a future rerun interpretable and help distinguish a real regression from a changed workload.

Appendix D turns these options into a staged rebuilding plan. Chapter 21 provides the shared-machine measurement rules, while Chapter 24 gives the packaging context for launch scripts and user-facing command defaults. This reference lists what the snapshot accepts; it does not replace those chapters' explanations of when an option is appropriate.

PREPRINT

Argument parsing details worth checking

The command parser is a small hand-written loop over `argv`. Options that take a value consume the next argument; a missing value falls into the usage/error path rather than silently using an arbitrary string. Numeric arguments are parsed as numbers and should therefore be written as plain values, without units attached. For example, the expert cache value is a number of gigabytes because the option names that unit; the context option is a token count. A suffix such as `GB` is not part of the documented numeric syntax.

Options can imply a broader mode. The parser sets chat behavior when a chat-specific choice is present, including system or thinking options and encoder use. This means a wrapper that adds one of those flags can change how the prompt string is rendered, even if it still passes `-p`. Check the full argument vector when comparing a shell invocation with an interactive chat. The equivalent visible text can produce different token IDs if one call uses the protocol template and the other is plain continuation.

The parser handles media selectors in the order described by `usage()`; multiple media inputs may be repeated. A video may contribute both sampled frames and sound unless the no-audio flag is selected. A test that intends to compare frame count or memory should therefore state that flag explicitly. Similarly, an image token budget affects the encoder's row count and downstream prefill work; a budget value is part of the workload, not an incidental presentation preference.

The `--depth` benchmark control is also distinct from context capacity. It sets the already-consumed prefix length used for the benchmark test; `-c` bounds the context. A long context option does not itself create a deep benchmark, and a deep test must have enough input tokens in its text file. When copying a benchmark command, preserve both values and the input corpus. Chapter 21 explains why a comparison should interleave variants and record timing statistic rather than rely on one run.

Finally, distinguish options that change model behavior from diagnostics. `--kv` and `--think` select different computation or protocol paths; `--image-tokens` changes encoder output rows. Profiling changes measurement scheduling, timing reports wait intervals, and `--tokens` exits before inference. Keep that distinction in mind when interpreting results.

PREPRINT

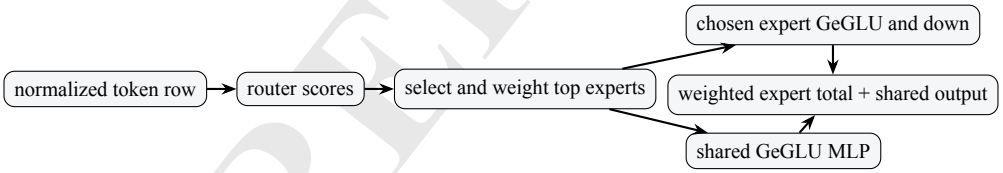
Appendix C

The 26B-A4B: what is different

What changes when the feed-forward block routes?

The main book follows the E4B dense path. The 26B-A4B keeps the same broad transformer shape but replaces a single large feed-forward choice with a shared MLP plus a bank of routed experts. This appendix is a map for readers who encounter that path in the source. It does not expand the E4B chapters into a second walk-through. The C implementation is concentrated in `gemma4.c:1376--1922`; the architecture and dated measurements are summarized in the frozen `README-26B.md`. The per-layer feed-forward and embedding discussion in Chapter 13 is the nearest companion.

The router produces one score per expert for each token. The implementation chooses the highest-scoring eight among 128, with lower expert index winning an exact tie because the scan only replaces the current best when it finds a strictly larger logit. It then normalizes the selected experts' exponentials over that selected set. Each selected expert computes a gate and up projection, applies GeGLU, then projects back down. Its result is multiplied by its routing weight and the layer's optional expert output scale before being added to the shared path.



The drawing separates routing from expert arithmetic because it has a scheduling consequence. The router's result determines which expert weights must be available. On the GPU path, the CPU reads the router logits, chooses the experts, checks the cache, and arranges token assignments by expert. Chapters 7 and 15 describe dense matvec and greedy selection; the 26B path adds this routing decision between layers.

How is the router decision made?

The excerpt below shows the tie rule and selected-set normalization. It is a short CPU reference, not a listing of all routed computation. The surrounding source defines the number of experts and selected experts from model metadata. The relevant loop is in the frozen snapshot at `gemma4.c:1446--1457`.

The implementation is at `gemma4.c:1446--1457`: it scans for the remaining maximum, leaves ties with the earlier index, then normalizes the selected weights.

The source comparison `lg[e] > lg[best]` gives deterministic lower-index tie behavior: expert indices are scanned upward, and an equal score does not displace the earlier candidate. The denominator is the sum of exponentials for selected IDs only. Subtracting the first selected logit before exponentiation avoids unnecessarily large exponent values and leaves the normalized weights unchanged. The chosen experts are then sorted into groups by the CPU so one expert's matrix rows can be processed together.

PREPRINT

Draft book, October 2026. Not peer reviewed.

Where do expert bytes live?

The routed expert weights dominate the 26B file. A mapped model file is not enough for the GPU path: prefill touches many experts in succession, and a file-backed page cache too small for the expert bank can thrash. Instead, the engine reads selected experts from the file into wired cache slots, using a per-layer share and staging areas. The cache is approximate in residency only; the stored expert bytes and arithmetic remain the model's. The cache budget changes memory use and miss traffic, not which expert IDs the router selects.

The structure excerpt below is a field guide to that policy. A slot belongs to a layer's share; the lookup arrays map layer/expert pairs to slots; scores and last-use stamps support eviction; the staged arrays record prefill reads that do not fit in a layer's share. The listing is lines 1534–1559 in the snapshot.

The fields are declared at `gemma4.c:1534--1559`.

The design gives each expert layer the same number of resident expert slots rather than allowing one layer to consume the whole pool. During prefill, the source starts reading the next layer's uncached experts while the GPU works on the current layer. During decode, it makes room within that layer's quota and favors recent routed use. This is the specialist part of the implementation: the dense path maps selected tensor ranges, while the MoE path has to choose which large expert tensors are resident now. The full cache lifecycle is documented in `README-26B.md`; the book does not claim those figures for E4B.

How do shared and expert paths meet?

Each token has a shared MLP result and a weighted sum of its selected expert results. The expert output scale is applied to each routed contribution when the engine builds routing weights. The two paths are accumulated into the layer's output. This is why "A4B" does not mean that every operation reads only four billion parameters: a token reads the shared dense weights and a subset of the expert bank, and different tokens may choose different experts.

There are two separate sorts of sharing in this codebase. The 26B's shared MLP is an ordinary feed-forward branch used alongside routed experts. The E4B's shared KV layers, described in Chapters 3, 11, and 14, reuse earlier key/value cache state. Similar names do not mean the same mechanism. The source's model loader assigns router and expert tensors to a layer only when the GGUF metadata describes them; the forward pass selects a dense or routed route from those populated fields.

What do the documented measurements mean?

The 26B README's performance table uses a Q4_0 file of about 14.6 GB and an M3 MacBook Air with 16 GB of memory. The expert cache makes the model runnable without requiring the full expert bank to remain resident. Its benchmark figures include a warm-up that fills the cache with experts used by repeated decode. They should not be compared directly with the E4B values in Chapter 1: the model, quantization, routing work, benchmark setup, and cache state differ.

The README also describes a later UD-IQ4_XS file whose dense tensors use Q8_0, experts use IQ3_S for gate/up and IQ4_NL for down, and one layer uses IQ4_XS/Q8_0. Its reported decode bytes and throughput belong to that file. They are historical observations, not a new W0/W1 experiment and not a promise for a different conversion. Chapter 21 gives the discipline for reporting such numbers; Chapter 22 explains why individual speed improvements must retain their model and measurement context.

How should an E4B reader use this map?

Use this appendix when following the router or expert-cache code. It points to the gate, up, and down tensors as well. Start with the model loader to see whether a layer has a router. Follow the route function to see selected IDs and weights.

PREPRINT

Table C.1: The 26B path at a glance. Architecture is distinct from the dated README measurements.

Question	26B-A4B map
Feed-forward paths	Shared GeGLU MLP plus routed experts
Expert bank	128 experts; top 8 selected per token
Routing normalization	Softmax weights renormalized over selected experts
Residency problem	Routed weights exceed the memory budget; Metal path uses an expert cache
Read path	Missing experts read from the model file into wired cache/staging storage
Measurement source	Snapshot `README-26B.md`; historical test machine and file are named there

Then inspect the expert cache if the question concerns residency or I/O, and the expert kernel dispatch if it concerns arithmetic. The short source excerpts here orient those searches; they are not replacements for reading the surrounding function boundaries.

For the dense E4B route, return to Chapters 13 and 14. For model-file tensor types and row orientation, use Chapter 3 and Appendix A. For command-line cache control and build variants, use Appendix B. This map keeps one important boundary clear: the 26B is supported by the engine, but it is not the model used for the main book's E4B rebuild checkpoints.

Follow one routed layer through the code

When tracing a layer, begin at the forward-pass dispatch and ask whether its layer record has a router. The dense path has one feed-forward block; the MoE path first computes router logits and the shared MLP work, then chooses expert IDs. This conditional is a useful navigational landmark in the C file because it separates the two layouts without requiring the reader to infer the architecture from tensor names alone.

After selection, the route builder counts assignments per expert and creates prefix offsets. A token that selected several experts appears in several assignment slots. Prefix offsets make the rows for each expert contiguous, so an expert's matrix operation can process its assigned rows together. The source keeps both the original token index and its sorted position, allowing the outputs to be accumulated back to the correct token. When inspecting a mismatch, check the mapping in both directions: the selected expert list and weights, then the token-to-sorted-row mapping, then the inverse accumulation.

The cache path follows a different set of indices. A logical expert is identified by layer and expert number; a slot is a current memory location. The `slot_of` table answers where an expert resides, while `owner` answers what a slot currently contains. A slot can be evicted and reused, so the expert's identity cannot be inferred from its numeric slot alone. The `last_use` and `freq` arrays track recency and a decaying score. This separation is essential: routing is a property of the model computation, while slot choice is a property of memory residency.

During a prefill, expert use is broad and can exceed the per-layer cache share. Staging receives the missing expert bytes that cannot become permanent residents at that moment. During decode, a token selects a smaller set and the cache can admit missing experts while preserving the entries used by the current layer. The source waits for that layer's reads before dispatching its expert kernels. This explains why cache misses can affect latency even when the dense weights are already available.

Read the performance evidence with its limits

The 26B README separates model-file bytes, dense weights, expert bytes, and cache residency. A weight-read estimate for one generated token is not the same as bytes read from SSD: cache hits are served from the wired pool, while misses trigger file reads. Likewise, a process footprint can omit or count mapped pages differently from a system-wide memory measurement. Chapter 23 makes the same

PREPRINT

distinction for the E4B mapping, and Chapter 21 explains how heat and background GPU use affect time measurements.

The benchmark README reports results for explicit model variants and commands. The Q4_0 file and the later UD-IQ4_XS file have different dense and expert formats, so their bytes-per-token and accuracy figures must remain with their respective file descriptions. A repeated benchmark can also reuse experts warmed by an earlier repetition; a fresh process starts with a different cache state. That is why the README names the test command and cache behavior rather than presenting one rate as a model constant.

This appendix uses the 26B path to explain a design, not to prescribe its parameters for a new machine. A smaller expert cache reduces resident memory but can increase reads and waits; a larger one can leave less room for the rest of the system. The appropriate setting depends on memory available to this process and on the sequence's routing pattern. The default and its command-line control are listed in Appendix B; the measurement procedure belongs to Chapter 21.

PREPRINT

Appendix D

Rebuild milestones

What does “rebuild” mean here?

The checkpoints are small programs that reconstruct one engine responsibility at a time and compare it with the frozen engine. They are a reading path through the real implementation, not a promise that the complete engine can be recreated by copying snippets. PLAN §4.4 specifies checkpoint programs of roughly 50–250 lines each. Their important output is a comparison against a known path, not a particular line count. The proposed 1,200–1,500-line mini-engine is separate future work and is not part of the guaranteed milestone sequence.

Milestones m1–m8 connect source reading to a working implementation. Each row names the smallest responsibility, the reference comparison, and its dependency. The source chapters explain the production code; this table is the checklist for a learner’s own compact version. The design and test strategy chapters are referenced by number because not every chapter has a stable internal section label.

Table D.1: Checkpoint sequence from tokenizer to chat prefix reuse.

Step	Program focus	Implement	Check against
m1	Tokenizer	Read vocabulary/merge metadata and reproduce byte-to-ID behavior	Engine `--tokens` on a fixed 20-sentence set
m2	GGUF directory	Parse metadata and tensor records; report tensor names, types, shapes, and byte sizes	`tools/gguf_cmp.py`; same directory rows
m3	Block decoder	Decode supported row formats from stored bytes	`tools/check_math dequant` on its selected tensors
m4	One CPU layer component	RMS norm plus a Q projection matvec	Engine CPU path on a layer-zero input vector
m5	Attention state	Add RoPE, attention, and a KV-cache update	The engine’s layer output for the same positions
m6	Whole CPU forward	Assemble the E4B forward pass for one token	Same top-1 ID as `gemma4_cpu` on ten prompts
m7	Metal path	Port a handful of kernels and preserve the same comparison boundary	CPU reference and engine output for the same test inputs
m8	Chat prefix reuse	Render entries and retain the common cached prefix	New-turn cost with a fake model whose logits are a token hash

How do the checkpoints depend on each other?

m1 and m2 can begin with small metadata fixtures, but the real comparison requires a local model file. m3 depends on the tensor type and row boundaries from m2. m4 needs m3 because its matrix-vector product must read the same quantized rows. m5 adds position and cache state to the arithmetic already understood in m4. m6 assembles the whole CPU path, and m7 moves selected operations to Metal only after the CPU path is a useful reference. m8 exercises the chat renderer and cache-prefix rule; it can use a fake model and does not need to generate natural language.

PREPRINT

This order reflects debugging cost. A wrong token ID can make every later value differ, so confirm tokenization before blaming attention. A tensor row interpreted with the wrong block stride corrupts matrix products, so validate the directory and decoder before building a layer. A cache indexing error can look like an attention formula error, so test one update and retrieval before integrating a full forward pass. Chapter 20's gates are the larger project's version of this same rule: start with the cheapest evidence that can isolate a fault.

m1: tokenize the same bytes

Build the string and pair tables from the GGUF metadata, form initial UTF-8 spans, and apply the merge order. Preserve the calling flags: ordinary tokenization and chat-template tokenization do not recognize special spellings in the same way. Compare exact ID sequences, not just reconstructed text. PLAN §4.4 sets a 20-sentence fixed comparison; include boundary inputs such as repeated spaces, newline runs, Unicode, and marker-like text. Chapter 2 explains the algorithm and its W1 probe.

m2: list the GGUF tensors

Read the header, key/value metadata, tensor directory, and aligned data start. Report type, dimensions, and calculated bytes without reading tensor payloads. Match the result to `tools/gguf_cmp.py`. The comparison catches a wrong dimension orientation as well as an offset or type error. It is a metadata-only milestone: do not turn it into a model-loading or inference test. Chapter 3 gives the file layout and explains why the first tensor dimension is the row width.

m3: decode one row format at a time

Implement Q8_0, Q4_0, Q4_1, Q4_K, Q5_K, Q6_K, IQ4_NL, IQ4_XS, BF16, and F16 row handling in the sequence the checkpoint specifies. Test the selected real tensors against `check_math dequant`. Keep type-specific block sizes explicit; a decoder that happens to work for one 32-value block can still misread the 256-value super-block layouts. Chapter 4 shows representative field maps, and Appendix A is the offset lookup.

m4 and m5: one layer, then its state

m4 computes an RMS-normalized input and one projection using the row decoder. Use the CPU implementation as a simple, inspectable reference. m5 introduces the attention cache: rotate query and key positions, store the key/value state at the correct position, and compare the resulting layer output. The test boundary must specify the position, mask type, and cache contents; otherwise two correct computations can be asked different questions. Chapters 10–12 cover the components, and Chapter 11 describes the cache's storage choices.

m6: one complete CPU prediction

Join embedding lookup, layer operations, final normalization, output head, and deterministic argmax. The test is the same top-1 token as the engine CPU build on ten prompts. A whole-model comparison is slow by design; PLAN describes it as about a token per few seconds, which is adequate for a short correctness sample. Matching one top ID does not prove every logit matches, so retain lower-level tests from m3–m5. This is the recommended single stopping point for a reader who wants one complete text-only CPU result. Chapters 13–15 connect the feed-forward block, full layer schedule, and token choice.

m7 and m8: hardware and interaction

m7 ports a small number of known-correct operations to Metal. It is not a request to reproduce every optimized kernel. Compare the GPU path against the CPU reference at the narrowest useful boundary,

PREPRINT

and then rerun the output check. Chapter 6 provides the host/Metal setup; Chapters 7–9 show kernel shapes and fusion.

m8 renders a conversation to a sequence of token or media entries, compares it with the previous rendered sequence, and keeps the longest unchanged prefix. A fake model can make logits a deterministic hash of the input token sequence, so the test can verify which positions are recomputed without loading weights. The checkpoint's promised behavior is prefix reuse, not conversational quality. Chapter 16 describes the production chat loop.

If you complete only one

Choose m6 if the goal is to understand how the pieces form a working language model. It is the first milestone whose test crosses the complete CPU forward path and returns a next-token decision. It also provides the reference for m7. If the goal is instead to learn file parsing or quantized arithmetic, stopping at m2 or m3 is a sound smaller result; each has its own direct comparison and does not need to be presented as an incomplete engine.

The mini-engine remains a proposal. PLAN estimates roughly 1,200–1,500 lines for a CPU-only, text-only E4B assembled from checkpoints and treats its design and validation as extra work. That proposal is neither an additional checkpoint nor a hidden requirement for finishing the book's rebuild path. The reliable promise is the sequence of independently testable milestones; decide separately whether the assembled artifact is worth the extra maintenance.

Checkpoint record

For each milestone, write down the snapshot revision, local model file identity if used, compiler command, input fixture, comparison command, and the first differing value if the check fails. Do not save only a success sentence: a future change needs to reproduce what "same" meant. For weight-dependent comparisons, record the exact tensor names and row positions. For tokenizer tests, keep the original UTF-8 bytes and flags. For Metal tests, say whether equality means same ID, same logits within a stated tolerance, or bit identity.

This record is small compared with the source code but turns the checkpoints into useful learning artifacts. It also follows Chapter 21's advice to keep conditions beside measurements and Chapter 20's distinction between visible output and numeric correctness. A test is strongest when its input and comparison rule make failure easy to locate.

PREPRINT

Draft book, October 2026. Not peer reviewed.

A review sheet for each checkpoint

The table below adds the review question that should be answered before moving on. It is not another test suite; it helps identify which layer of the implementation a failed comparison implicates.

Table D.2: What to inspect when a milestone does not match.

Step	If the comparison differs, first inspect	Keep with the result
m1	BOS/special flags, UTF-8 byte spans, space marker expansion, newline-run behavior, merge rank and tie order	Input bytes, flags, IDs, and spelling for each ID
m2	GGUF version/counts, metadata value widths, tensor dimensions, alignment and offset base	Tensor name, type, shape, byte count and data-start position
m3	Block stride, low/high nibble order, super-block subgroup, scale extraction and scalar conversion	Tensor/row identity and first differing decoded value
m4	Row orientation, input vector, quantized row decode, normalization scale and accumulation type	Layer/tensor names and output-vector comparison rule
m5	Position, rotary coordinates, causal/window mask, cache owner and read/write index	Initial cache state, position sequence, and layer output
m6	Layer ordering, shared-KV metadata, residual scales, PLE row, final norm and argmax tie	Prompt, IDs, first changed top token, and intermediate boundary if available
m7	Buffer lifetime, synchronization, kernel dimensions and CPU/GPU input identity	Build flags, device path, input hash and numerical comparison
m8	Rendered entry identity, common-prefix boundary and rollback rule	Previous/current entry sequence and number of retained positions

These are diagnostic locations implied by the milestones and the production data flow, not claims that a particular failure was observed in every checkpoint. The project's documented pitfalls remain in their chapters with their original sources. In particular, Chapter 15's token-buffer race shows why a GPU/CPU synchronization issue can corrupt later results while leaving the process running; Chapter 20 explains why one visible transcript is not a complete numeric gate.

Keep the comparison boundary narrow

Each milestone should first compare the smallest output that establishes its contract. For m1 that is a list of IDs from identical input bytes. For m2 it is a tensor directory, not tensor arithmetic. For m3 it is decoded row data. For m4 and m5 it is a vector at a specified stage. For m6 it is the selected next-token ID, supplemented by narrower checks when it differs. For m7 it is a kernel or operation against the CPU reference before the full model is exercised. For m8 it is which prefix is retained and which suffix is evaluated.

This order prevents a later mismatch from hiding its cause. If m6 fails and the only evidence is a final different token, there are many possible sources: tokenization, file parsing, weight decoding, normalization, attention state, layer scheduling, or argmax. If m3 already has a direct comparison to the checked decoder, the search can start later in the data flow. The same principle applies to speed: a whole-model rate tells whether the user-visible path changed, while a targeted kernel timing explains where.

Test data and reproducibility

Keep fixed examples alongside each implementation, not embedded only in a notebook or a one-off terminal history. Tokenizer fixtures should preserve exact bytes; a printed Unicode string can hide normalization or newline changes. GGUF checks should record the local file identity and tensor subset. Decoder fixtures should include type and shape metadata, since the same bytes have no meaning without a layout. Cache tests should initialize state explicitly, so an old slot is not mistaken for the expected value of a fresh cache.

For full forward checks, keep the prompt or token IDs, beginning-of-sequence behavior, context position, and cache format. A comparison against a different quantized file is not a regression comparison

PREPRINT

unless that difference is intentional. For Metal, record the compile definitions and path used; a CPU-only binary and a Metal-enabled binary can exercise different code. These records mirror the book's W0/W1/W2 distinction: identify whether a check needs no weights, metadata, or actual inference before running it.

The checkpoint table is deliberately independent of a particular output transcript. Greedy text is a convenient smoke check, but it does not say which internal value first diverged. A rebuild is most useful when each checkpoint can be repeated with the same inputs and its failure can be reduced to one boundary. That is the practical meaning of an incremental rebuild path.

What is outside the promise?

The milestones rebuild a teaching path, not every optimization in the production implementation. m7 is a handful of kernels, not a complete performance-equivalent Metal backend. m8 demonstrates a cache-prefix rule with fake logits, not a full chat application with image and audio lifetimes. The optional vision and audio encoders are separate chapters and are not prerequisites for the text-only m1–m6 path.

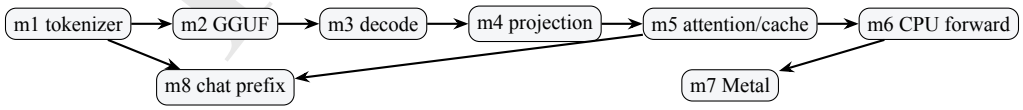
The proposed mini-engine would assemble checkpoint work into a single CPU-only, text-only program. The plan estimates its size, but does not treat it as present or validated. It would need its own end-to-end correctness checks, packaging choices, and documented limitations. A reader who completes m6 has already reached the core learning goal: a compact program that computes a complete next-token decision and can compare it with the reference engine.

Suggested work sessions

The milestones can be grouped into work sessions without changing their dependency order. The first session can focus on m1 and m2 because both establish how the model file becomes usable information. The second can focus on m3 and m4, ending with one checked projection result. The third can implement m5 with deliberately small cache fixtures before attempting real model tensors. The fourth can assemble m6 and compare a short prompt set. The last sessions can explore m7 and m8 independently: one studies the GPU boundary, the other studies chat state.

This grouping is a practical suggestion rather than a schedule promise. A learner may already have a parser or a Metal harness, or may want to stop after understanding one format. The important property is that a later checkpoint consumes a result whose earlier contract has been checked. If one part takes longer, preserve its test boundary and resume there instead of moving a mismatch into the next stage.

A dependency sketch



The arrows show the main data dependencies rather than an enforced project-management order. m8 needs token and cache concepts, but it does not need m6's complete model arithmetic because its planned fake model isolates prefix reuse. m7 follows a checked CPU result so it can use that path as a reference. This is why the milestones are a ladder with one side path, not eight chapters that must all be finished before any result is useful.

Choose the right stopping point

If the goal is reading a model file, m2 is a complete stopping point: it produces a tensor inventory and tests directory interpretation. If the goal is understanding numerical formats, m3 is complete when its

PREPRINT

row comparison is reproducible. If the goal is seeing a working model prediction, continue through m6. If the goal is GPU programming, m7 inherits a meaningful CPU answer to compare against. If the goal is interactive state, m8 demonstrates the prefix rule without pretending to implement inference.

Write down the stopping point explicitly when sharing a checkpoint with another reader. "I rebuilt the engine" can mean a tokenizer, a directory reader, a complete CPU forward path, or a GPU kernel subset. Naming the milestone tells the next reader exactly what is established and which checks remain. This clarity is the same reason the book labels experiments W0 through W3: it states the dependency before anyone starts a run.

For a handoff, include the smallest input that demonstrates the checkpoint and the expected comparison boundary. A new reader should be able to reproduce the same test without guessing which model file, tensor row, prompt flags, or cache state was intended. If a model file is required, identify it by its local file identity and keep it outside the book's source snapshot. If the milestone is metadata-only, say so plainly; it prevents someone from loading weights just to inspect a tensor directory.

When a check fails, preserve the first mismatch before changing the implementation. A later run may differ for another reason, and editing several layers at once destroys the evidence needed to isolate the original fault. Keep the known-good lower checkpoint available as a reference while adding the next piece. This makes each milestone a useful rollback point for reasoning, even though the project itself remains a set of small programs rather than a single production binary.

PREPRINT

Appendix E

Glossary

This glossary gives the meaning used in this book. A term may have a broader meaning elsewhere; here it is tied to the C and Metal engine described in the chapters. Related terms are grouped so a reader can follow the path from model-file bytes to a generated token. Chapter references point to the first or most useful detailed treatment.

Text and model output

Token. One vocabulary ID consumed or produced by the language model. A token can represent a word, part of a word, punctuation, a control marker, or a byte fallback. It is not the same as a word or a Unicode character. See Chapter 2.
Tokenizer. The metadata-driven conversion between input bytes and token IDs. In this engine, the text path uses byte-pair merges and can optionally recognize selected special spellings.
Logit. A model's raw score for one possible next token. The output head produces one logit per vocabulary ID; greedy generation selects the highest score. See Chapter 15.
Greedy decoding. Selecting the largest logit at every step, with the engine's deterministic tie rule. It is simple and repeatable, but it is not temperature or top-k sampling.
Perplexity (PPL). A summary of how much probability a model assigns to a sequence of text. Lower is better only when the compared runs use compatible tokenization, text, and scoring conditions. See Chapter 20.
KL divergence (KLD). A measure of how two probability distributions differ. In the book it compares model output distributions at scored positions; it can expose a numerical change even when the top token stays the same. See Chapter 20.
Top-token agreement. The fraction of scored positions at which two runs choose the same highest-scoring token. It reports decision agreement, not equality of the complete distributions.
Chat template. The protocol that wraps roles, messages, channels, and turn-ending markers around ordinary text. It controls which strings become special token IDs. See Chapters 2 and 16.
Special token. A vocabulary ID used as a control marker, such as a message start or end. The same visible spelling can be ordinary text when the caller has not enabled special-token recognition.

Transformer data and attention

Activation. A temporary value produced while evaluating the model: for example, a token embedding, normalized row, or MLP hidden vector. Activations are not learned weights.
Attention. A layer operation in which a query compares with stored keys and uses the resulting weights to combine values. Chapters 10–12 follow its projections, cache state, and Metal kernels.
Grouped-query attention (GQA). An arrangement with fewer key/value heads than query heads. Several query heads share one key/value head; the E4B's 8 query heads and 2 KV heads form four-query groups. See Chapter 10.

PREPRINT

Draft book, October 2026. Not peer reviewed.

KV cache. Stored key and value vectors from earlier positions, reused so generation does not recompute their projections. It can use F16 or an 8-bit representation, and some model layers share another layer's cache. See Chapter 11.
Ring buffer. A fixed-capacity array reused by wrapping a logical position modulo its size. The E4B uses rings for sliding-window cache state; overwriting an old row limits how far a chat rollback can reuse state. See Chapter 11.
Sliding window. Attention restricted to a recent range of positions. Its cache can be bounded by the window instead of growing with the complete conversation.
Global attention. Attention over the retained full history rather than just the recent sliding window. Its cache use grows with context length.
RoPE. Rotary position embedding: position-dependent rotations applied to pairs of query and key coordinates. It lets attention scores reflect relative position without adding a separate position vector. See Chapter 10.
RMS norm. Root-mean-square normalization of a vector, followed by learned scaling. The engine fuses or reuses some normalized inputs to avoid standalone work. See Chapters 4, 9, and 13.
GeGLU. A gated feed-forward operation: a GELU-transformed gate is multiplied elementwise by an up projection before a down projection. The routed 26B path applies this inside each chosen expert as well as in its shared branch. See Chapter 13 and Appendix C.
PLE (per-layer embeddings). Learned token-dependent contributions used by the E4B at multiple layers. The large table remains CPU-side and supplies the row selected by a token ID. See Chapter 13.

PREPRINT

Weights and storage

GGUF. The model container format read by the engine. Its header and metadata describe the model; its tensor directory gives names, shapes, types, and offsets into the payload. See Chapter 3.
Tensor. A named multi-dimensional array in the model file. Matrix orientation matters: in this engine's GGUF reader, `ne[0]` is the row width and `ne[1]` is the row count.
Quantization block. A group of weight values encoded together with shared scale or metadata. The code must decode the block layout before using its values. Appendix A is the byte-offset reference.
Super-block. A larger quantized unit, commonly 256 values, that groups eight 32-value sub-blocks and shares additional scale metadata. Q4_K, Q5_K, Q6_K, and IQ4_XS use such layouts in the supported decoder.
Dequantization. Expanding stored quantized values into numerical weights, either into a temporary array or as part of a kernel's multiply. See Chapter 4.
F16 (half precision). A 16-bit floating-point storage and compute format with a 5-bit exponent and 10-bit fraction. It has less range than F32 but more fraction detail than BF16.
BF16. A 16-bit floating-point representation formed from the upper half of an F32 bit pattern. It keeps the wider exponent range of F32 with fewer fraction bits.
Mapped file. A file made addressable in a process's virtual memory. Mapping does not mean every page has been read into physical memory.
Wired memory. Pages made resident and available to the GPU through Metal. The wired-weight banner counts selected GPU-accessible model ranges, not the complete file or total process memory. See Chapters 3 and 23.
Page cache. The operating system's cached file pages. Repeated reads may be served from memory, but a working set larger than the cache can be evicted and reread from storage.
Expert cache. In the 26B MoE path, a wired pool holding a subset of routed expert weights. It is distinct from the KV cache; Appendix C maps its fields and read path.

GPU execution and measurement

Kernel. A function launched over many GPU work items. In this book the Metal source defines kernels for matvecs, attention, normalization, and other steps. See Chapters 6–12.
Threadgroup. A set of GPU threads scheduled together for one portion of a kernel's work. Threads can cooperate through threadgroup memory and synchronization.
Simdgroup. A smaller cooperating group of GPU lanes with hardware matrix operations on Apple GPUs. The exact mapping is part of the kernel design; it is not interchangeable with a CPU thread group.
Command buffer. A sequence of encoded GPU operations submitted for execution. Submissions can create synchronization boundaries; reducing unnecessary dependent submissions can lower latency. See Chapter 6.
Prefill. Processing multiple prompt positions together. Matrix-matrix kernels can reuse weights across rows, so its bottleneck differs from one-token decode. See Chapters 1 and 8.
Decode. Generating one next token at a time after the prompt. Each step reads much of the model's weights, which makes memory traffic and dependent GPU steps important. See Chapters 1 and 7.
Roofline / bandwidth floor. A way to bound performance by bytes moved divided by sustained bandwidth. It is a lower bound on transfer time, not a complete prediction when arithmetic, synchronization, or cache misses also cost time. See Chapters 7 and 21.
Wall time. Elapsed time observed by a user, including CPU scheduling and waits. It is not the same as accumulated GPU kernel time.
Idle gate. The load and utilization checks used to decide whether a shared machine is quiet enough for a measurement. Chapter 21 explains the project's capture procedure.
Interleaved run. A comparison ordered A, B, A, B so heat and background load affect both variants more evenly than running all A tests before all B tests.

Putting the terms together

A text prompt becomes tokens; the model looks up token embeddings and evaluates layers. Attention reads cached keys and values, while the feed-forward path multiplies activations by quantized tensor rows. A Metal kernel may decode a quantization block as it streams bytes, and a command buffer orders that kernel with the other steps. The final logits are reduced to the next token ID, which becomes input to another decode step. During prefill, the same weights can serve many rows at once, so the speed limit and useful kernel shape change.

The words "cache" and "memory" deserve particular care. The KV cache holds attention history; the expert cache holds selected MoE weights; the operating system's page cache holds file pages; Metal wired memory makes chosen pages resident for GPU access. They have different owners, lifetimes, and meanings in a memory report. Chapters 11, 21, and 23 distinguish them in context, and the command reference in Appendix B identifies the control that applies to each.

A few easily confused pairs

Weight vs activation. A weight is a learned value stored in the model file. An activation is a value produced while evaluating a prompt or token. Weights can be quantized on disk; activations are commonly represented in F16 or F32 during the computation.
Tensor shape vs byte layout. A tensor shape describes logical dimensions. Its type determines how those values are serialized. A quantized tensor's row length and block format together determine row bytes; multiplying dimensions alone is not sufficient. See Chapters 3–4 and Appendix A.
File size vs wired size. File size counts bytes in the GGUF. Wired size counts selected pages made resident for Metal. Neither number alone is total process memory. Chapter 23 reconciles these quantities.
GPU time vs wall time. GPU time measures device execution under the timer's rules. Wall time includes the complete observed interval, including CPU work, waiting, and scheduling. A speed claim must name

PREPRINT

Draft book, October 2026. Not peer reviewed.

which one it reports.

Prefill vs decode. Prefill handles many input rows together; decode advances one position at a time. A kernel or optimization that helps one phase may not help the other.

Exactness vs agreement. Bit identity means equal stored result bits at a chosen boundary. Top-token agreement means equal winning IDs. KLD measures distribution difference. These are distinct checks and answer different questions.

Additional engine vocabulary

Embedding. A learned vector selected by token ID. The token embedding begins the forward pass; the E4B also has per-layer embeddings added inside later layers.

Projection. A matrix multiplication that maps a vector or row to a new width or role, such as query, key, value, or output. Projection names in the GGUF identify these learned matrices.

Head. One attention subspace. A query head forms scores and combines values; grouped-query attention allows several query heads to share one key/value head.

Mask. A rule deciding which positions an attention query may see. A causal mask prevents looking forward; a sliding-window mask limits attention to recent positions.

Flash attention. An attention implementation that processes score/value tiles without materializing the full attention matrix. The book's Metal kernels use tiled online accumulation; see Chapter 12.

Online softmax. A way to accumulate a softmax-weighted result a block at a time while keeping a running maximum and normalization sum. It avoids storing every score at once.

Residual. The path that adds a block's result back to its input representation. Residual scaling and normalization are part of the model's layer behavior, not merely a kernel optimization.

Argmax. The operation that returns the index of the largest score. The engine performs this reduction on the GPU in the pipelined generation path; see Chapter 15.

Prefetch. Reading data before the immediate consumer needs it, to overlap I/O with useful computation. Incorrect predictions or cache eviction can make prefetching slower; Chapter 22 records the measured dead end.

Checkpoint. A small rebuild program and a comparison rule that establish one piece of understanding before the next is added. Appendix D lists m1–m8.

When a term appears in a source comment with a narrower meaning than its general GPU or machine-learning usage, follow the surrounding data flow. For example, "cache" may refer to position-indexed attention state or expert-weight residency; "scale" may mean a quantization factor, normalization coefficient, or model output multiplier. Read the variable's lifetime and the values that index it before carrying a familiar definition into the code.

PREPRINT

Appendix F

Exercise hints

F.1 Chapter 1

1. Convert 4.72 GB to bits, then divide by 4.25 bits per weight. The result is an order-of-magnitude estimate: metadata, padding, block scales, and nonuniform tensor types make the average approximate.
2. Divide bytes per token by bytes per second to estimate minimum seconds per token, then invert. The real engine also spends time on attention, selection, and command dispatch, and its sustained bandwidth is below the illustrative 100 GB/s.
3. Start with the reader and mapping code around `gemma4.c:313--359`; then follow `map_model` around line 4518 and the Metal host's `metal_map`. A no-copy buffer refers to the mapped file pages in unified memory.
4. The renderer produces role and turn markers along with message text. If the new rendered token sequence shares an initial run with the cached conversation, only the unmatched suffix needs fresh prefill.
5. The logits are scores; greedy argmax needs only the winning token ID. The chosen ID is turned into its text piece using the tokenizer vocabulary and streamed to the terminal. Check the Metal scoring kernel and the C generation loop.
6. The W0 build has no model argument and the usage routine exits 1 after printing help. Any prompt-generation, benchmark, perplexity, or KL run that opens a model would be a weighted experiment and is not part of this chapter's W0 check.

F.2 Chapter 2

1. Start with the space marker, then `t`, `h`, `e`, and another space marker. Merge `t+h` (rank 1), then `th+e` (rank 2), then the left marker with `the` (rank 3), and finally the result with the right marker (rank 4). The final piece includes both space markers.
2. A candidate records its left position and the expected left and right IDs. After a merge, the node length or its neighbour link changes; compare those stored IDs with the current linked sequence and discard a mismatch. This avoids maintaining arbitrary deletion positions in a heap.
3. Starting with whole UTF-8 characters prevents a valid multi-byte character from being split before merges are considered. If that character has no vocabulary piece, the output loop recovers each original byte and emits its `<0xXX>` ID so no byte is lost.
4. Look up each ID in `tk.vocab` to print the spelling. A byte token is a visible hex spelling; `U+2581` stands for a space in token notation, even though detokenization writes an ordinary ASCII space.
5. Count the input's encoded bytes and the resulting IDs, then divide IDs by elapsed seconds. Use the same timer boundary for every repetition; if timing only tokenization, read the file before starting the clock.
6. Keep the vocabulary's special-token ID list separate. With recognition enabled, choose the longest listed spelling at the current byte offset, flush preceding plain text through BPE, and append one

PREPRINT

Draft book, October 2026. Not peer reviewed.

control ID. With recognition disabled, pass the same characters through ordinary text tokenization.

F.3 Chapter 3

1. Read the tensor record in this order: a length-prefixed name, a dimension count, that many dimension values, a type ID, and a 64-bit offset. The offset is relative to the aligned tensor-data start, so the file address is the mapping base plus data start plus offset.
2. The row length is 2,560, which is `ne[0]` and the input-feature count. There are 2,048 rows, which is `ne[1]`.
3. Add the desired key to the probe's small `wanted` set. The main loop parses that value and calls `skip_value` for all other metadata; this is important for large vocabulary arrays, which should not be retained as Python objects.
4. Compute `absolute = data_start + offset`; check `absolute % alignment == 0` for this file's 32-byte alignment, then verify `absolute + size <= file_size`. Tensor byte counts must use the recorded block format and shape.
5. The first run may need the operating system to fetch metadata pages; a later run may reuse them. The timed work is parsing the directory and touching metadata pages, not reading tensor payloads. State whether the timer includes process startup and file open.
6. The per-layer token table is CPU-read with random-access advice; shared-layer K/V weights are not used by those layers; F32 originals of converted F16 matrices have F16 copies; expert weights use the expert cache. Each reason is visible in the `map_model` exclusion construction and its preceding loader code.

F.4 Chapter 4

1. Divide each block's complete storage by its value count. IQ4_XS is 136/256 = 0.53125 bytes per weight; Q4_K is 144/256 = 0.5625; Q5_K is 176/256 = 0.6875. The metadata is why those values exceed the ideal code-only rates.
2. The low nibble of `0x9E` is 14 and the high nibble is 9. Q4_0 subtracts eight from each code, then multiplies by the fp16 scale. These are positions 0 and 16 because the two nibble halves are separated by sixteen weights.
3. Reuse the directory parser to find the tensor's absolute data address, then slice just the requested super-block. Check that the selected format's byte size matches the bytes read; do not turn a tensor selector into a whole-payload scan.
4. The Q4_K and Q5_K local scale/minimum values are six-bit fields. The upper groups borrow two high bits from other bytes in the 12-byte metadata area; see the CPU decoder's masks and shifts.
5. For IQ4_XS, the first code byte is `0x89`; its low nibble 9 selects 13. Multiply by the effective local scale -0.001329422 to get about -0.017282486, matching the first W1 value.
6. The CPU path fills a decoded row; Metal decodes 32 weights into a small half tile for the current multiply. Inspect the tile's bounded dimensions. The per-layer F32 matrices are a distinct exception because the engine keeps F16 working copies for repeated token use.

F.5 Chapter 5

1. Tile index is `18/16 = 1`; the second tile owns rows 16–18 and writes three valid rows. It has 16 slots in its temporary tile, but only three correspond to real output rows.
2. Static chunks avoid an atomic operation for each task, but a slow chunk can leave other threads idle. The increasing-duration callback reveals this imbalance; repeat with fixed inputs and separate scheduler overhead from the callback work.

PREPRINT

3. Keep the allocation size and indexing consistent: each thread needs K floats for a decoded row plus K times NB floats for the transposed tile. Compare repetitions at each setting and report the median rather than the fastest run.
4. A common scalar preserves direction. The coordinate-specific learned weights stretch or shrink coordinates by different amounts, so they can change direction as well as length.
5. The direct implementation can decode a quantised value inside the inner loop. The row version first expands the row, then takes its dot product. Keep both on the same input and check their outputs before comparing times.
6. Fix the model, prompt IDs, initial cache contents, and precision settings. Save the activation entering the layer and vectors after normalization, attention, residual, feed-forward, and final projection. Report a numeric error measure and top ID; a top ID records only the maximum coordinate and discards all other values.

F.6 Chapter 6

1. In a 128-thread group, lane indices 0, 32, 64, and 96 are the first lanes of the four SIMD groups. Each such lane writes the subtotal reduced by its own 32-lane group. The second barrier ensures every lane has finished reading the shared subtotal array before any lane leaves the helper.
2. `metal_map` rounds the mapped address down and its end up to page boundaries. The buffer offset passed by `find_region` is the original tensor pointer minus the start of that page-rounded region, so the shader sees the tensor's exact first byte.
3. Update the number of elements, the mapping file's length and contents, the second input, the output allocation, and the dispatch count together. Keep the mapped base and length page-aligned even though the kernel reads only eight floats.
4. The second kernel must not observe unfinished output from the first. If they share one encoder, the ordinary memory barrier between encoded steps establishes the dependency. A no-barrier `metal_group` is valid only when the steps are independent. Separate command buffers on this queue are submitted in order, but the host still needs a wait before it reads the output itself.
5. Separate compilation and pipeline creation from the repeated dispatch loop; they are startup costs. Record whether the timer includes command-buffer creation, commit, and wait. A wall clock around a batch measures host plus GPU and queue overhead; GPU timestamps measure execution intervals but not all host costs. Tiny dispatches mostly expose those fixed costs, so report the batch size and avoid treating one kernel's time as a stable throughput number.
6. The CPU path must not race the previously submitted GPU operation on shared activations. A counter can increment at the wrapper boundary whenever a supported call returns zero, before the CPU fallback executes; report per-operation counts so one unsupported shape does not disappear inside a total.

F.7 Chapter 7

1. A Q4_0 block has its two-byte scale at offsets 0–1, then the packed code bytes at offsets 2–17. Lanes 0 and 1 take byte ranges 2–9 and 10–17 from block 0; each half supplies 16 codes and therefore 16 activation positions. Lanes 2 and 3 repeat those byte ranges for block 1, starting 18 bytes later.
2. Multiply the number of rows by the stored row bytes, and account for both matrices in the fused gate/up call. Convert microseconds to seconds before dividing bytes by time; decimal GB/s uses 1,000,000,000 bytes per second. A close result checks the arithmetic, not that all traffic reached DRAM exactly once.
3. For `blk.0.attn_q.weight`, K=2,560 gives 80 blocks per row and ten blocks per lane. The kernel uses four rows per SIMD group, four SIMD groups per group, and 16 rows per threadgroup, so 2,048 rows need 128 groups. Count code bytes separately from repeated scale metadata loads.
4. Q4_0 represents each weight as scale times (code minus 8), so distribute the subtraction across the dot product as scale times (sum of code-products minus 8 times sum of activations). Q4_1 represents a

PREPRINT

weight as scale times code plus a shared minimum, so the correction is minimum times the activation sum.

5. The dispatch uses the Q4 branch for Q4_0/1, the Q4K branch for Q4_K, and the IQ4NL branch for IQ4_NL or aligned IQ4_XS. Check the final branch for the generic matvec fallback.
6. A weight reused for 32 token rows supports 64 floating-point operations per stored weight. Divide by the IQ4_XS byte cost per weight to estimate intensity; the estimate excludes activation and output traffic and is a roofline comparison, not a measured rate.

F.8 Chapter 8

1. The operation count is $2 \times 512 \times 2{,}560 \times 10{,}240 = 26{,}843{,}545{,}600$ FLOP. At 2.2×10^{12} FLOP/s the ideal time is about 0.0122 s; this ignores attention, memory staging, launch costs, and the rest of the model.
2. The decoded weight tile is shared by the token fragments mapped to simdgroups. Widening the output tile can reduce repeated activation loads, but it increases the live matrix accumulators and can cause register spills.
3. At T=48, TOK_SG=2: a group covers two 32-row token tiles, or 64 padded rows, and two row blocks. At T=128, TOK_SG=4: it covers four token tiles, or 128 rows, with one row block.
4. Compare every result element, not just a checksum. A changed loop order can still be correct while producing a different checksum only if the arithmetic itself changed; for integer inputs the result should be exact.
5. Attention and the many other projection or residual steps contribute to prefill time; the documented optimization that skips unneeded rows after layer 23 also affects the end-to-end rate. The README comparison is not a matmul-only benchmark.
6. At 128 token rows, the ideal IQ4_XS weight-only traffic is about 0.0021 byte per FLOP. It ignores scale metadata, activations, output stores, and any weight tile fetched again.

F.9 Chapter 9

1. Count `metal_matmul` calls inside the Q/K/V group as concurrent work, then count the following preparation, attention, output projection, residual, FFN, and optional per-layer embedding steps. The source accounting in ENGINE-MATH §5.14 gives about 12 dependent steps for a decode layer.
2. 500 steps times 10 microseconds is 5 milliseconds. Removing one dependent step from each of 42 layers would save about 420 microseconds at that assumed cost; this is an estimate, not a measured speedup.
3. The separate version materialises the normalised branch, the residual sum, and the normalised next input. In the fused row kernel, each thread keeps its assigned coordinates locally across the reductions and writes the final residual and optional fp16 next input.
4. The attention call produces `a->a`; `add_norm` combines it with `a->x` and writes `a->xn` with `L->ffn_norm`. The FFN reads `a->xn`. Layer zero still needs the initial norm of the embedding because there is no preceding layer tail.
5. Score can compute log-sum-exp statistics, save quantised log probabilities, and compare against a base distribution for KL statistics. Standalone argmax only chooses the maximum ID and writes it to the token destinations.
6. The toy dispatches touch distinct synthetic array entries and have no model weights, no large matrix traffic, and no inter-layer dependencies. Their launch-to-work ratio and command-buffer setup differ from the engine, so only the checksum and general launch-cost observation transfer.

PREPRINT

F.10 Chapter 10

1. Subtract the maximum 2 to get (0,-1,-2). Exponentiate to get approximately (1,0.3679,0.1353), whose sum is about 1.5032; divide each by that sum.
2. Heads 0,1,2,3 use KV head 0; heads 4,5,6,7 use KV head 1. Their queries and resulting score lists remain distinct even though each group reads the same K/V sequence.
3. Replacing B's key with (2,0) gives scores (1,2,1). After subtracting 2, exponentiate (-1,0,-1) to get approximately (0.3679,1,0.3679); dividing by their sum gives weights (0.2119,0.5761,0.2119).
4. Eight independent queries require eight output vectors, but two KV heads hold the shared key and value streams. Compare eight K/V vectors per position with two: one quarter as many K/V vectors.
5. The global layers are 6, 12, 18, 24, 30, 36, and 42. At position 700, each global layer can still see position 0; a sliding layer sees positions 189 through 700.
6. With window 4, the first visible position is max(0,p-3). So the visible ranges are 0–2, 1–4, and 3–6; the query position is included, making four positions once the window fills.

F.11 Chapter 11

1. A sliding owner's K row has 2 heads × 256 values = 512; K and V together have 1,024 values. A global owner has 2 × 512 = 1,024 per K or V, or 2,048 combined.
2. There are 20 sliding owners and 4 global owners: 20 × 1,024 + 4 × 2,048 = 28,672 values. At 2 bytes each that is 57,344 bytes; with one fp16 scale per 32 int8 values it is 28,672 + (28,672/32 × 2) = 30,464 bytes. Sliding storage has a fixed ring row count; only global rows continue to grow with context.
3. The loader chooses an earlier matching layer as `kv_src`; state construction assigns the shared layer's K/V and scale pointers from that source and skips allocation. It also does not load that layer's K/V projection tensors.
4. Rows 0 through 3 hold positions 6, 7, 8, 9; rows 4 and 5 hold positions 4 and 5. Row index is position modulo 6.
5. With `n_high`=1300 and ring 1600, no position has yet been overwritten, so both prefixes 700 and 900 remain valid. Try a later `n_high`, such as 2300: prefix 700 needs positions from 188 and fails, while prefix 900 needs from 388 and also fails under this state; compare the needed first position with `n_high - ring`.
6. At c=512, the README reports 0.000217 for the 8-bit cache and 0.000012 for fp16. Same-top counts only the winning ID; KL compares the probability distributions across all candidate IDs and can reveal shifts even when the winner stays the same.

F.12 Chapter 12

1. After score 1: $m = 1$, $l = 1$, $o = 10$. After score 3, rescale those by e^{-2}, then add the weight-one item with value 20. After score 2, keep maximum 3 and add a weight e^{-1} item with value 40. Divide the final numerator by the final denominator and compare with a two-pass calculation.
2. Divide 32,768 by 32 to get 1,024 chunks. To stay within 64, a chunk needs at least 512 positions; the host rounds that size to a multiple of 32. The exact 32,001-position walkthrough uses 512-key chunks and 63 chunks.
3. Head 7 has KV head $7/4 = 1$ and is the fourth query head in that group. Grid X is 1; its SIMD group index is 3. Grid Y is the chunk. The lane owns dimensions lane times 16 through lane times 16 plus 15 for the 512-wide head.
4. A chunk numerator and denominator are scaled to that chunk's maximum. If its maximum is 2 and the shared maximum is 5, multiply both by $e^{2-5} = e^{-3}$ before adding them to the other chunk.
5. There are $256/32 = 8$ scales per K vector. Q8 stores 256 bytes plus 16 scale bytes, or 272 bytes for K and another 272 for V: 544 bytes total for the pair.

PREPRINT

6. Keep the score function deterministic and use the same payload in both methods. An error near floating-point rounding is the expected check; a large discrepancy usually means the running numerator or denominator was not rescaled when the maximum changed. Runtime is incidental to this correctness test.

F.13 Chapter 13

1. Gate and up take 2,560 input features and make 10,240 output features each. Down takes 10,240 and returns 2,560, so the result can join the residual stream.
2. One token row contains 10,752 Q5_K values. Divide by 256 to get 42 blocks, then multiply by 176 bytes: 7,392 bytes. There are 42 adjacent slices, each 256 values wide.
3. The two PLE matrices in each layer each have $2{,}560 \times 256 = 655{,}360$ values. Across 84 matrices, F32 storage is $84 \times 655{,}360 \times 4 = 220{,}200{,}960$ bytes. F16 copies use half: 110,100,480 bytes.
4. Use the exact same input and matrices in both paths. Compare all 10,240 output values after GELU and multiplication; the synthetic test should not rely on model inference. The diagnostic array should hold the activated gate values, while the reference hidden value is that array multiplied elementwise by the up projection.
5. The measurement is specific to the snapshot, hardware, GPU idle state and prompt. Log wall time and GPU time separately, and inspect the wired-memory banner. Skipping PLE is not an equivalent model: it removes a learned branch and changes subsequent values.

F.14 Chapter 14

1. Begin at the normed input. Q/K/V projection and preparation lead to attention; output projection forms its branch; post-attention norm and add make the FFN input; gate/up with GELU and multiply, then down, make the FFN branch; post-FFN norm and add update the stream; the PLE gate, multiply, projection, post norm, add and layer scale finish the layer. Decode groups its independent Q/K/V matmuls, uses a fused GLU kernel for gate/up plus activation/product, and combines residual with norm in row kernels. The down and output projections remain their own matrix operations.
2. Multiply each category's per-layer total by its count and add: sliding owners 20, global owners 4, sliding shared 15, global shared 3. Shared layers still read Q and output matrices; only their own K/V projection weights and cache writes disappear. The sum is about 2.18 billion bytes of layer matrix weights.
3. The attention branch is added after post-attention norm; the FFN branch is added after its post-FFN norm. PLE adds after its projection and PLE post norm. Its tail then applies the layer output scale.
4. The gather is eligible only for causal calls with extra rows at the first layer that has no owned cache. Disabling it keeps all 512 rows through layers 24–41 instead of only the requested final row. Compare raw logits from the same run, not just generated text, and keep GPU conditions fixed.
5. Group the descriptors by the metadata's sliding-window pattern and head dimensions. For each tensor, multiply its quantisation block count by encoded block bytes; do not use bits-per-weight alone because block scales and metadata are part of storage.
6. If every layer owns its cache, `l == m->n_kv` is never true within the loop: `n_kv` equals the layer count, while the loop's last index is one less. The gather path is skipped and `Tl` remains the original row count through the final layer.

F.15 Chapter 15

1. ID 1 wins: the maximum 1.2 appears at IDs 1 and 2, and the reference keeps the first maximum. On the GPU, each lane retains a score and ID, and `better` selects the lower ID when scores tie; the group reduction repeats the same comparison.

PREPRINT

Draft book, October 2026. Not peer reviewed.

2. The GPU argmax writes into `a->tokens` and the generated output array. Synchronous mode waits before the next iteration and calls back with the current ID. Pipelined mode submits the following step and reports the prior iteration's token while it runs; it flushes the last token after the loop.
3. Keep the logits on the GPU for a production sampler if possible. A CPU prototype must wait for the logits, read their values, choose a filtered candidate with a reproducible random stream, then provide the selected token for the next call. Time the wait/read, filtering and sampling separately from forward; also compare total throughput with the greedy GPU path. A precise cost needs a quiet-GPU measurement.
4. A changed lower-ranked score can leave the winning ID unchanged while altering the distribution substantially. KLD compares probability distributions; same-top checks only one ID. Compare the full token-ID arrays from synchronous and pipelined decode, and repeat a statistical distribution check over many scored positions.
5. Persist a seed or generator state across steps and define exactly how it advances. The interface needs the logits (or sufficient candidate scores), sampling settings, generator state, and destination token slot. To reproduce, reset the same seed and inputs; test CPU and GPU with the same random stream. Ensure command-buffer order makes the chosen ID visible before the next forward reads it.

F.16 Chapter 16

1. List the control markers around the two user turns and completed model reply, then append the open model prefix. After the first reply, the cache contains the rendered prompt entries and the generated reply token IDs; the next rendering should reproduce that sequence before its new suffix.
2. The count should be the valid cached prefix length, including any template markers represented by entries. Recompute or shorten it after a history edit; set it to zero on clear. State whether your diagnostic is read before or after the open generation prefix is rendered.
3. Removing an earlier exchange changes the positions and context for all later hidden states. Re-render the retained transcript and find the common prefix, then recompute from that boundary; do not splice states from after the removed turn.
4. First synchronize any queued GPU reads. While media rows are still alive, render the remaining conversation and shorten the cache to the common prefix. Then free the rows. Freeing first risks reusing the same address for different media, which can fool pointer-based Entry equality.
5. Total prompt length includes the whole rendered conversation and template markers. New-token count is the suffix that must run after a valid cache prefix. Reply-token count is generated output. Measure elapsed wall time separately, and record GPU time only if the timing instrumentation provides it.

F.17 Chapter 17

1. The W0 example prepares a 1,024-by-768 image at 912 by 672. Dividing each side by the 16-pixel patch size gives 57 columns and 42 rows, or 2,394 patches. Groups of three patches per axis give 19 by 14 pooled locations, which is 266 output rows. The budget is a maximum, so alignment and aspect ratio can produce fewer than 280.
2. Divide the patch index by the grid width: 100 divided by 57 gives row 1 with remainder 43, so it is at row 1, column 43. The first half of each head rotates by column; the second half rotates by row. Patch indices are zero-based.
3. A 21-by-15 grid contains 315 patches. The 3-by-3 grouping has 7 columns and 5 rows, so it produces 35 output rows. The exercise's dimensions are already complete merge groups.
4. The intervals limit outliers in the learned encoder computation; they are not needed to avoid F16 overflow. `load_linear` reads optional input and output bounds from metadata. `clamp_in` applies input limits before the projection, and the matmul output path clamps stored results. The loader checks that projections sharing one clamped input really have the same bounds.

PREPRINT

5. Use the same image, prepared size, and activation in both runs. Dump the rows from each encoder, check the row count and width, and report cosine per row plus relative error over the complete arrays. The environment setting for the quick GELU must match on both sides if comparing against a converter that uses quick GELU. A caption can remain unchanged even when individual embedding values differ, because the language model may still choose the same words.

F.18 Chapter 18

1. Ten seconds at 16 kHz is 160,000 samples. The source framing rule gives about 1,000 feature frames, and two stride-2 stages leave about 250 rows. Exact counts depend on the actual sample length and chunk boundary.
2. Split 70 seconds into 30, 30, and 10 seconds. The approximate row counts are 750, 750, and 250. The causal attention history and five-tap convolution context restart at both chunk boundaries.
3. The window has 320 real samples and the 512-point transform zero-pads the remainder. Taking power instead of magnitude squares the spectral values and changes every mel feature supplied to the learned convolutions.
4. One first-convolution output location gathers a 3-by-3 neighborhood from the time/frequency plane. Each output channel accumulates its nine weighted values; then channel LayerNorm and ReLU run, and the row is stored as F16. Streaming a row avoids retaining a full F32 output tensor.
5. At frame 20, attention can see frames 9 through 20, inclusive. The learned relative term tells attention how far away a key is; the causal convolution mixes five neighboring rows channel by channel and cannot use future rows.
6. Use the same sample bytes, resampler, prompt, encoder files, and chunking. Compare row count, rowwise cosine and relative error, full-array error, and transcript. Record machine, date, encoder formats, GPU idle state, and median or p10 timing. A small row difference may leave the selected words unchanged.

F.19 Chapter 19

1. Multiply 10 frames by the 70-row ceiling, then add about 250 audio rows: at most 950 media rows. Multiply by 2,560 components and 2 bytes each for 4,864,000 bytes. This omits markers, timestamp and question tokens, KV cache, activations, model weights, and allocator overhead.
2. A rendered Entry with a non-null row pointer refers to encoded media and a row index; without one it contains a token ID. Media uses ID zero because it has no vocabulary token whose per-layer embedding should be added. Its supplied vector already fills the language-model-width input row.
3. Two calls: 256 rows in the first and 24 in the second. The call bound limits one staging/forward batch. The longer media sequence is divided across calls and accumulates in the KV cache.
4. The cap selects 32 frames instead of 300. It can miss a brief action between sample times, especially as the spacing becomes longer than one second on a long clip. The sampled images provide no direct evidence about intervals the decoder did not return.
5. Q8_0 changes eligible F16 matrices in the encoder GGUF file. Retained rows are converted from float32 to fp16 after encoding, while the staging buffer is a temporary copy for one forward call. The command-line ‘–quantize‘ option only makes the smaller encoder file.
6. Keep the main model, encoder file, exact prompt, media bytes, output limit, and build fixed. Record peak system-wide wired, anonymous, and compressed-memory rise with ‘memwatch.sh‘, the idle-GPU gate, date and machine. Compare repeated or interleaved trials and report a median or p10, not the best single run. Run text, image, and video cases separately so each added attachment is identifiable.

PREPRINT

F.20 Chapter 20

1. The tokenizer may have different IDs that detokenize to the same bytes, or a whitespace/control detail may be hidden by the displayed sentence. Inspect the first differing position's IDs, token spellings, and high-precision logits.
2. Same-top says the top token agrees. It misses the large change in probability assigned to the winner and runner-up; KL compares the full distributions.
3. KLD is computed row by row against the saved base. If text, chunks, positions, or model differ, the rows are not corresponding predictions, so the number no longer measures the intended implementation difference.
4. Confirm rendered token IDs first; find the earliest score position that differs; compare CPU and GPU or intermediate outputs around RoPE; run the direct RoPE check; then rerun the model-level comparison after fixing the identified path.
5. Floating-point addition is order-dependent because each addition rounds. A repeatable small discrepancy with matching inputs and no overlap is consistent with reduction-order rounding; a large or timing-sensitive change that vanishes with synchronization points toward a race. Compare the same score rows and inspect buffer lifetimes.
6. Name the changed kernel and its observable output, run its direct input/output comparison, then run fixed greedy IDs or the KLD reference matching the path. Record binary, model, text, chunking, KV format, and logs. Block acceptance on unexplained token changes or a score difference outside the intended approximation.

F.21 Chapter 21

1. Use the same executable, model, text, options, warm-up, and generation length for every A/A run. Keep individual outputs and compare neighboring runs. A spread near three percent cannot establish a three-percent A/B gain; more controlled repetitions or a quieter interval would be needed.
2. One token at 24.7 tokens/s takes about 1/24.7 seconds, or 40.5 ms. Multiplying 2.61 GB of weight bytes per token by 24.7 tokens/s gives about 64.5 GB/s of weight traffic. The calculation omits the KV history read by attention and does not say what fraction of peak bandwidth is achieved.
3. Grouped trials put each candidate in a different part of a potentially warming or cooling time series. Alternation makes nearby A and B observations more comparable and exposes shared drift.
4. Report that the helper timed out, warned that the GPU was busy, and still launched the command. Capture a utilization trace over the timed interval, then compute its mean and maximum against the stricter limits. A pre-run gate alone does not prove the interval stayed quiet.
5. The profile committed and waited after each step, adding command-buffer round-trip and schedule effects. Batch repeated calls and synchronize after the batch, as kbench does, then report the per-call statistic and its boundary.

F.22 Chapter 22

1. Episode 1 removes late-layer work after the last KV-writing layer. Check the guard in `gemma4.c:2094` and trace what the `rows` gather copies before reducing `Tl`. Compare the logits and cached state needed by the next decode, not only a generated sentence.
2. The token table is selected by one token ID per decode step. Draw the dependency from GPU argmax to CPU row lookup to the next submitted GPU work. Mark which interval overlaps and which timing would reveal an exposed CPU delay.
3. F16 copies halve the per-token bytes of the 84 F32 matrices. Q8_0 gained about 2
4. A wider tile can reduce activation reads while increasing live accumulators. Inspect register use or spills, then compare interleaved timings for tile shapes; include a small-output case, not only a large synthetic matrix.

PREPRINT

5. E4B has four query heads per KV head and E2B has eight. Read the `Gs` and `SPL` mapping in the Metal source at lines 1282–1286, then sketch which threadgroups reuse a KV head and when a 512-dimensional head must split.
6. An optimization log entry should include the phase, baseline and candidate source identities, model and input, machine state, interleaved raw observations, statistic, exactness evidence, and any memory or quality cost. For W2/W3 measurements, describe the reader experiment and predicted result shape; keep dated project measurements labelled as historical.

F.23 Chapter 23

1. At 16K, use the cache totals already tabulated in Chapter 11. The wired weights, F16 copies, and tokenizer do not scale with context; global-layer KV grows while sliding-window rings retain their bounded size. Keep the process footprint and system-wide peak as separate estimates.
2. Compare the fixed wired-weight count with the variable cache and encoder working set. Then measure system-wide peak and process footprint on the same machine and workload; page cache alone does not show whether the GPU-wired model pages are included.
3. The 2.63 GB is the GPU-wired weight range. The 0.33 GB is the process's reported extra footprint. The 3.08 GB is a sampled system-wide rise in wired, anonymous, and compressed pages. They have different scopes, rounding, and sampling, so the residual is not evidence of a missing tensor.
4. Subtract the 8-bit total from the fp16 total at 32K using Chapter 11's values. Then place that saving beside the 8-bit cache's KLD and top-token results from Chapter 20; memory savings alone do not establish acceptable output behavior.
5. Inspect Chapter 17's activation-buffer notes and Chapter 14's prompt-row handling. A smaller call batch can reduce temporary activations while leaving the long-lived context cache at its allocated size.

F.24 Chapter 24

1. The build script resolves its location into `HERE`, sets the output directory relative to it, and runs the compiler from the snapshot. A local `./gemma4` destination would put a binary beside the source and could overwrite a checked-in or otherwise important file.
2. Use a timeout utility or a small supervisor that starts the child, waits at most ten seconds, then sends an interrupt and, if needed, terminates only that child's PID. Preserve stderr in a separate log and return a distinct timeout status so the caller can tell it apart from an engine error.
3. Define a versioned JSON object for each event or one complete reply record. Keep diagnostic messages on stderr; include a status field for completion, interruption, and errors. Specify escaping, newline framing, and whether partial text is emitted before an interrupted status.
4. Fix the exact model file, binary identity, prompt bytes and turn order, flags, context, output cap, and working directory. Save stdout and stderr separately, record the date and machine, and repeat the same commands in the same order. A W2 comparison is a reader experiment using a local model and the documented quiet-GPU procedure.
5. The launcher selects paths and defaults; the engine renders prompts and evaluates tokens; the voice layer captures and synthesizes audio and coordinates scheduling; the app presents controls and handles platform permissions. Media selection can be split: the launcher chooses the encoder file while the engine decodes a supplied image or audio item.

PREPRINT

Bibliography

Apple (2026). *Metal*. URL: https://developer.apple.com/metal/ (visited on 10/22/2026).

llama.cpp contributors (2026). *GGUF file format*. URL: https://github.com/ggml-org/llama.cpp/blob/master/docs/gguf.md (visited on 10/22/2026).

PREPRINT