27. Attention and GEMM Kernels

Chapter Abstract. A scheduled request row can survive batching and still be lost at launch time. An attention kernel may visit KV pages in tile order; a GEMM kernel may pad rows, reinterpret packed weights, or apply FP8 scale blocks. The runtime has to prove that those physical choices still write the output for the same logical request. The account follows two launches, a decode-attention row and a projection GEMM, from backend metadata to kernel evidence. The attention-backends chapter in Volume I chooses a family. The launch-level question is what one concrete path must record before speed can count as the same operation rather than a different one.

27.1. The Launch-Level Problem

The plain picture is that attention kernels read request-specific K/V history, while GEMM kernels multiply scheduled activation rows by projection weights. Ragged KV pages mean different requests have different cache lengths; scale records say how low-precision matrix blocks should be interpreted; padded rows exist for launch alignment and must be hidden from the request. A GEMM *epilogue* is the post-matmul step that applies scale, bias, dtype conversion, or output slicing before the result is written.

A decode row reaches an attention launch with ragged KV pages, causal masks, and a selected backend. A prefill row reaches a projection GEMM with packed weights, scale records, and maybe extra padded rows. Both launches can be fast and still be wrong. The attention path can read the wrong KV row. The GEMM path can apply the wrong FP8 scale block, meaning one scale shared by a tile of FP8 values. An epilogue can also return padded rows as if they were real tokens.

The launch-level problem begins where a backend-family decision becomes an individual device launch. Volume I supplies live request rows, cache visibility, masks, and output ownership. Here the runtime chooses a concrete tile and storage plan. It records page tables and scale records. It also records workspace and output slices, then checks that the physical plan still implements the scheduled attention or GEMM operator.

Fast attention and GEMM paths must preserve serving semantics.

Figure 27.1 makes the decode half visible: the kernel is free to stream KV pages in tile order, but the final row must still belong to the selected request. The same rule applies to the prefill-heavy half. Its projection GEMMs may pack operands, split work, or pad rows, but the output rows still have to match the scheduler's request order. Those obligations come from the request-row and KV-slot contracts in the KV cache chapter in Volume I and from the attention-backend abstraction in the preliminaries chapter in Volume I.

Here a *projection GEMM* is the GEMM form of a transformer linear projection. If the scheduled activation rows are $X \in \mathbb{R}^{M \times K}$ and the projection weight is $W_\pi \in \mathbb{R}^{K \times N}$, the logical layer asks for $Y = XW_\pi \in \mathbb{R}^{M \times N}$, possibly with bias or quantization metadata. The phrase emphasizes that the model requests one

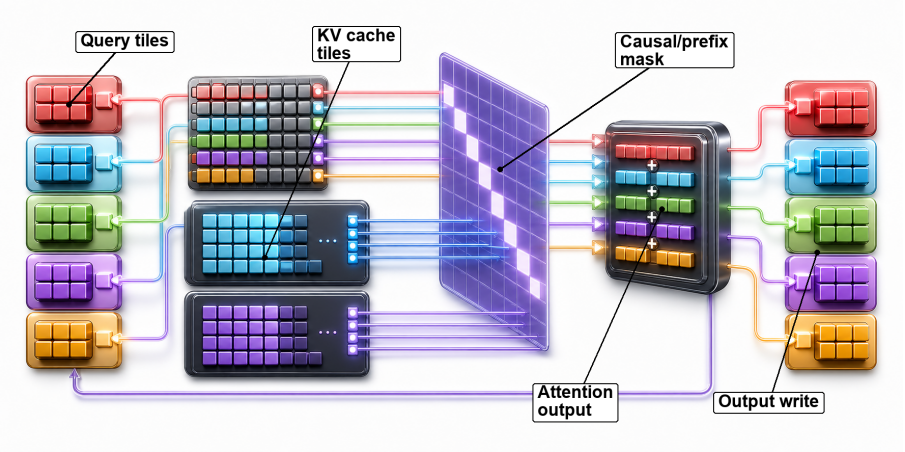

Figure 27.1.: Attention kernel dataflow. Query tiles reading key and value cache tiles, applying causal or prefix masks, accumulating attention output, and writing the result for one request batch.

Table 27.1.: Running example ledger. The two halves exercise different kernels, but both carry request-owned metadata through a kernel-facing shape.

Half	Request rows	Kernel-facing view
Decode-heavy	A and B, one new row each	Attention over long KV histories, with context lengths, masks, cache pages or KV slots, and output rows as metadata.
Prefill-heavy	C, 128 scheduled prompt-suffix rows	Projection GEMM with $K = N = 4096$, plus scheduled suffix span, padding, packed-weight, or scale-layout metadata.

projection map, while the backend sees a matrix multiply whose M rows must remain the scheduler's token rows.

The running example is a two-part `meta-llama/Llama-3.1-8B-Instruct` serving step. In the decode-heavy half, two resident requests A and B each contribute one new query row while reading long KV histories through paged-cache metadata.

The example uses four local terms. *Request metadata* means the positions, masks, KV slots, and output rows that identify the serving step. An *attention kernel* consumes query, key, value, and cache-location state. A *GEMM kernel* consumes matrix operands and scale or packing metadata. A *workspace* is temporary backend-owned storage needed to run the kernel.

In the prefill-heavy half, request C contributes 128 prompt-suffix rows, the scheduled part of its prefill suffix as described in the chunked-prefill chapter in Volume I, to a projection with $K = N = 4096$. The attention kernel may split KV reads or visit pages out of request order, and the GEMM kernels may pad rows or use packed low-bit weights (Definition 19.2.4), but both halves must return activations that still belong to A, B, and C in the scheduler's order.

Table 27.1 records the running-example state before the section follows its later transitions.

Across both halves, tile order, split work, padding, and packing may change the physical launch shape. The outputs still have to attach to the scheduled request rows. The broader source-coverage boundary appears later in Section 27.10; the opening keeps only the two running cases the chapter will reuse. Readers who are not tracing kernel implementation should track three fields: which request row

Table 27.2.: Local notation for attention and GEMM kernel shapes. The serving-shape symbol is S_t, not K_t, because K is already the GEMM contraction dimension and the key tensor. Backend field names such as `kv_indices` and `block_table` are introduced in the sections that use those metadata records directly.

Symbol	Meaning	Role in the serving account
S_t	Serving kernel shape at step t	The concrete launch record together with the logical attention or GEMM operation.
$\mathscr{B}_t$	Scheduled token rows for serving step t, including row order and per-row request identity	Rows whose outputs must remain attached to the runtime's request order.
$\mathbf{g}_t$	Operator geometry	$(H_Q, H_{\mathrm{KV}}, d_h)$ for attention or (M, K, N) for GEMM.
τ_{tile}	Tile or split policy	Query/KV tile shapes, GEMM tile shapes, or split metadata used by the backend.
κ	Layout and metadata bundle	Cache indices, page tables, scale layout, padding, workspace, and output slicing.
$I(S_t)$	Arithmetic intensity	$\mathrm{FLOPs}(S_t)/\mathrm{bytes}(S_t)$ for the concrete serving kernel shape.

owns the output, which metadata gives the bytes meaning, and what baseline makes a latency number comparable.

The two-part serving step needs notation for launch records as well as operators. Table 27.2 names the recurring serving-shape, row-order, geometry, tile, metadata, and measurement symbols that make an attention or GEMM launch comparable.

Running-example labels remain prose labels: A and B denote the decode rows, and C denotes the scheduled prefill suffix. The serving-shape definition in the next section carries the remaining tuple fields locally.

One decode row from scheduler state to output.

For the decode-heavy half, make the row contract concrete before considering backend-specific kernels. Suppose request A has 96 visible history tokens before the step and request B has 2048. After each request contributes one new query and its K/V entry has been written, the attention view records `seq_lens` = $[97, 2049]$. A Triton-style flat-index view can record the segment boundaries as `kv_indptr` = $[0, 97, 2146]$, so row 0 owns the first 97 K/V slots and row 1 owns the next 2049. With page size $P = 16$, those same visible histories occupy 7 pages for A and 129 pages for B, although the concrete page identifiers remain allocator choices.

This trace is intentionally smaller than a full backend record. The detailed FA3 and FlashMLA examples in Section 27.5 use the same request rows and output ownership, then add the backend scheduler metadata or latent block tables needed by those kernels. The important point is already visible here: tile splitting changes how memory is visited, not which request row a query belongs to.

On Hopper and Blackwell, some selected CUDA kernel paths can also expose programmatic dependent launch as a launch-overlap optimization. The serving contract is the one defined in Chapter 26: PDL may move independent same-stream work earlier, but it cannot let a later attention or GEMM launch consume values before the required dependency wait [25].

Dataflow stage	Request A	Request B	Must preserve
Scheduler row	row 0, one decode query after 96 tokens	row 1, one decode query after 2048 tokens	row order
Forward-batch fields	`seq_lens`[0] = 97, output row 0	`seq_lens`[1] = 2049, output row 1	lengths
KV-index view	`kv_indices`[0:97] names A's visible slots	`kv_indices`[97:2146] names B's visible slots	segments
Tile loop	query row 0 streams only A's K/V tiles	query row 1 streams only B's K/V tiles	mask state
Output write	softmax accumulator writes A's activation to row 0	softmax accumulator writes B's activation to row 1	owner

Table 27.3.: Attention dataflow stages for two requests. Each stage preserves request-visible ownership while page identifiers remain allocator choices.

For CUTLASS specifically, the name expands to NVIDIA's CUDA Templates for Linear Algebra Subroutines: a C++/CUDA template library for composing architecture-specific GEMM and related tensor-core kernels [27]. Later references to a CUTLASS path mean SGLang selected a kernel path whose legality depends on shape, dtype, layout, scale metadata, architecture support, and epilogue behavior; model semantics stay fixed.

Example 27.1.1 (Shape-preserving attention metadata bug) The tensor extents can still look plausible when the metadata is wrong. If the same decode step above accidentally records `kv_indptr` = $[0, 96, 2145]$, each row silently loses the newly written K/V slot, so the new query cannot attend to the cache entry that should complete its visible history. If `kv_indptr` remains $[0, 97, 2146]$ but one entry in `kv_indices`[97:2146] points into A's cache slots, request B's query reads a foreign history while the output shape remains two rows. Both failures preserve a reasonable launch shape; both violate attention dataflow because the metadata no longer denotes the scheduled requests.

Concept-to-code map

SGLang makes this contract visible at two boundaries. On the attention side, `AttentionBackend.forward` dispatches decode, extend, and mixed modes, while concrete backends such as `TritonAttnBackend` and `CutlassMLABackend` build KV indices, page tables, split counts, workspaces, and CUDA-graph metadata before invoking kernels. On the GEMM side, kernels such as `fp8_blockwise_scaled_mm` and QServe W4A8 kernels reject incompatible strides, dtypes, scale layouts, alignments, or compute capabilities before launching. These checks make a mathematical operator executable for a particular serving shape [42]. Sections 27.4, 27.5, 27.6, and 27.7 expand the attention and GEMM execution paths.

27.2. Serving Kernel Shape, Layout, and Cost Vocabulary

The launch-level problem needs a vocabulary that is wider than tensor shape but narrower than a source-code tour. A serving kernel shape names the logical operator and scheduled rows first. It then records sequence state, geometry, tile policy, numeric policy, and layout metadata for one device launch. A layout view explains how those logical indices become bytes, pages, packed values, scale records, workspace, and output slices. This section is a reference layer: the serving argument depends on the logical row-to-output mapping, while CTA, warp, MMA, and shared-memory terms explain how one legal launch may implement that mapping. Arithmetic intensity then gives the same launch a cost vocabulary before the chapter compares attention and GEMM bottlenecks.

Kernel vocabulary names launch geometry; it does not change the serving row semantics.

Tile hierarchies from logical rows to GPU work.

Definition 27.2.1 (GPU execution primitives) *A GPU* thread block*, also called a cooperative thread array or CTA, is a group of threads scheduled together on one streaming multiprocessor. A* warp *is the smaller SIMD-style group that executes one instruction stream. A* tensor core *is a matrix-multiply unit inside the GPU. An* MMA *instruction is a matrix-multiply-accumulate instruction issued by warps to those tensor cores. A* register accumulator *is per-thread fast storage holding a partial output value, and* on-chip shared memory *is block-local storage used to reuse operand tiles before they return to global memory.*

A serving step names logical rows, but a GPU kernel executes a hierarchy of tiles. For GEMM, a thread-block or cooperative-thread-array (CTA) tile owns a rectangular region of C. It streams the K dimension through operand tiles, stages reusable A and B fragments in on-chip storage, and lets warps issue tensor-core matrix-multiply-accumulate (MMA) instructions into register accumulators. For attention, the outer tile owns query rows and streams visible K/V tiles or pages. The row's running softmax state, defined in Section 27.4, travels with that query tile until the final output row is written. The symbol τ_{tile} in the serving kernel shape is therefore a compact handle for the physical work hierarchy. It stands for block ownership and warp fragments. It also covers on-chip staging, register state, and epilogue mapping [2, 27].

Table 27.4.: Tile-level vocabulary for attention launches. The tile objects explain which kernel role each launch-level unit plays.

Tile level	Attention launch role
Logical serving rows	Scheduled query rows read visible K/V history and later write row-aligned output.
CTA or thread-block tile	Owns a query block and visits K/V blocks or pages in an implementation-chosen order.
Warp and MMA tile	Computes qk fragments and value contributions while preserving mask state and accumulator dtype.
On-chip staging and registers	Carry K/V tiles plus running softmax statistics until the output row is ready.

Layout question	Attention view	GEMM view	Failure if wrong
Which indices become neighboring bytes?	K/V rows are normalized to token, KV-head, and head-dimension views, but the cache may be flat, paged, or latent.	Operands may be row-major, column-major, transposed, or packed before the tile loop sees them.	The kernel reads the wrong vector, column, or page.
Which work unit is contiguous?	A query tile streams visible K/V slots or pages while keeping per-row softmax state.	A CTA tile owns an output rectangle and streams K_b-sized operand tiles.	Coalescing or reuse assumptions no longer match the operands.
Which indirection names the live rows?	`kv_indptr`, `kv_indices`, page tables, or block tables map scheduled rows to visible history.	Padding, packed-weight permutations, and output slices map the physical launch back to M scheduled rows.	The output extent looks plausible while row ownership is stale.
Which metadata interprets stored values?	FP8 K/V scales, cache dtype views, MLA latent geometry, and mask state tell the kernel what the cache bytes mean.	FP8, INT8, W4A8, or other scale and zero-point layouts tell the GEMM how to dequantize tiles.	The arithmetic uses plausible bytes with the wrong numerical meaning.

Table 27.5.: Layout questions for attention and GEMM. Each view has a different failure mode when layout metadata is wrong.

The GEMM-specific version of the same CTA-to-warp hierarchy appears in Section 27.6; here the point is only that the serving shape must name enough tile policy for the runtime to validate the launch it actually issues.

Layout is the bridge between this tile hierarchy and the bytes a kernel actually touches. A logical attention operand may be described as tokens, heads, and head dimension, but the decode kernel may see those same entries through flat slots, paged blocks, MLA latent pages, or a compact `kv_indices` list. A logical GEMM may be described by M, N, K, but the implementation may require row-major activations, column-major weights, packed low-bit weights, padded rows, or scale tiles. The shape says how many values exist; the layout view says where the kernel finds them and which metadata gives them meaning.

Definition 27.2.2 (Kernel layout view) *A kernel layout view is the metadata that maps logical operand indices to physical storage and interpretation metadata for one launch. It includes axis order, strides, contiguous dimensions, flat-slot or paged indirection, packed-weight ordering, scale or zero-point grouping, tile ownership, workspace shape, and output slicing whenever those fields affect which bytes the kernel reads or writes.*

Example 27.2.1 (Same logical shape, different layout views) Suppose a decode row needs a K/V history exposed logically as [tokens, H_{KV}, d_h]. A flat-slot cache can store token slot 37 directly, while a paged cache with page size 8 reaches the same logical token through page 4, offset 5. The logical K/V shape is unchanged, but the layout view changes the address calculation and the page-table metadata the attention kernel must consume. For request C's projection, $A_{128\times4096}B_{4096\times4096}$ likewise names a logical GEMM shape, not a complete launch: a row-major A, column-major B, packed FP8 scale-block layout, and padded output slice are separate facts. A kernel that infers them from M, N, K alone is missing part of the operator.

Tensor shape alone is ambiguous at this boundary. The same output extent can come from different request rows, cache pages, split policies, packed-weight layouts, scale blocks, or padded launch buckets. The chapter therefore names the whole kernel-facing bundle that must stay aligned with the scheduled serving step.

Definition 27.2.3 (Serving kernel shape) *A serving kernel shape is the tuple*

$$S_t = (o_t, \mathcal{B}_t, \mathbf{s}_t, \mathbf{g}_t, \tau_{\text{tile}}, \Delta_{\text{dtype}}, \kappa)$$

presented to a device kernel in a serving step. Here o_t names the logical attention or GEMM operator, $\mathcal{B}_t$ names the scheduled request rows, $\mathbf{s}_t$ records sequence or context lengths, $\mathbf{g}_t$ records operator geometry such as (H_Q, H_{KV}, d_h) for attention or (M, K, N) for GEMM, τ_{tile} is tile shape, Δ_{dtype} is dtype and accumulation policy, and κ collects layout, cache-index, scale, workspace, padding, and output-slicing parameters.

Example: Running decode shape as a concrete launch view. For the decode half of the running `meta-llama/Llama-3.1-8B-Instruct` step, requests A and B have post-write sequence lengths 97 and 2049. In a one-GPU grouped-query-attention instance with $H_Q = 32$, $H_{KV} = 8$, $d_h = d_v = 128$, and BF16 cache values, the tuple in Definition 27.2.3 binds to concrete buffers rather than just names:

$$o_t = \text{decode attention}, \quad \mathcal{B}_t = [(A, 0), (B, 1)],$$
$$\mathbf{s}_t = [97, 2049], \quad \mathbf{g}_t = (32, 8, 128).$$

The same logical decode call can therefore appear as flat `kv_indices`, a padded page table, or backend-owned scratch. The visible result stays tied to the scheduled rows: row 0 writes A's activation, row 1 writes B's activation, and no workspace or page padding creates an extra request row.

The prefill projection for request C binds the same tuple differently: o_t is GEMM, $\mathcal{B}_t$ is C's 128 scheduled suffix rows, $\mathbf{g}_t = (128, 4096, 4096)$, and κ carries packed-weight, scale-layout, padding, and output-slicing metadata. Section 27.7 fills in that GEMM-side scale record. The symbol S_t is therefore the concrete launch record together with the algebraic operation.

The serving shape fixes the concrete operands and metadata a kernel instance will see. Arithmetic intensity uses that same instance as the accounting unit

so work and memory traffic are compared for the actual serving launch rather than for an abstract operator.

Definition 27.2.4 (Arithmetic intensity) *Arithmetic intensity for a kernel instance S_t is*

$$I(S_t) = \frac{\text{FLOPs}(S_t)}{\text{bytes}(S_t)},$$

the ratio of arithmetic work to operand, cache, metadata, workspace, and output memory traffic for that concrete serving shape.

Low intensity means the launch is likely to spend most of its time moving bytes; high intensity means it has enough arithmetic to keep compute units busy. This is the roofline picture used later in Section 27.9: a first lower bound is max(bytes/bandwidth, FLOPs/throughput).

For a dense GEMM of shape $M \times K$ by $K \times N$, the logical work is about $2MKN$ floating-point operations; the leading 2 counts a multiply and an add.

Example 27.2.2 (Small GEMM intensity ledger) For a dense FP16 GEMM $A_{2\times4}B_{4\times4} \to C_{2\times4}$, the logical work is $2MKN = 2 \cdot (2) \cdot (4) \cdot (4) = 64$ floating-point operations. Reading A and B and writing C touches $8 + 16 + 8 = 32$ FP16 values, or 64 bytes before reuse and metadata. The toy intensity is therefore about one FLOP per byte. A real serving launch may improve reuse through tiling or lower it with scale tables, page indices, split buffers, and output slices; the serving kernel shape determines which ledger is relevant. Figure 27.2 shows that the arithmetic and traffic accounts both belong to one concrete serving kernel shape.

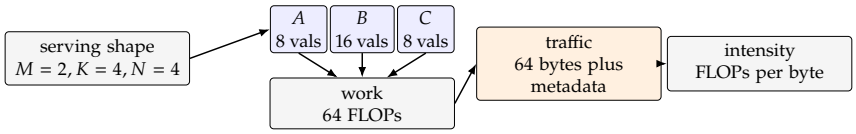

Figure 27.2.: Arithmetic intensity is computed for a concrete serving kernel shape. The same algebraic operator can have a different traffic ledger when metadata, cache pages, scale layouts, or output slicing change.

With $I(S_t)$ fixed as the ledger for one serving kernel shape, the important distinction is how different kernels fill that ledger. For a dense GEMM of shape $M \times K$ by $K \times N$, the logical work is about $2MKN$ floating-point operations, while the byte term includes the actual operands, the output, and implementation metadata such as FP8 scale blocks or packed low-bit weights. Decode attention has a different ledger: its work grows with heads, head dimension, and the live context length, but its bytes are often dominated by KV-cache reads, page-table or index traffic, split-state buffers, and output writes. This contrast is why a projection GEMM can be compute-dense while a one-token decode attention kernel can be limited by metadata and memory movement even when both are called from the same forward pass. The serving-shape state attached to the ledger determines whether SGLang can batch it, test it, and measure it without confusing speed with correctness.

Table 27.6.: Launch tuple binding for the running shape. Each tuple field fixes a concrete shape or byte consequence.

Tuple field	Concrete binding	Shape and byte consequence
$\mathscr{B}_t$ and output slice	Row 0 belongs to A; row 1 belongs to B.	The query and output tensors have shape $[2, 32, 128]$ in BF16, or $2 \cdot 32 \cdot 128 \cdot 2 = 16{,}384$ bytes each.
$\mathbf{s}_t$ and flat KV view	`seq_lens` is $[97, 2049]$; `kv_indptr` is $[0, 97, 2146]$.	`seq_lens` is `int32[2]` and `kv_indptr` is `int32[3]`. The Triton-style `kv_indices` buffer has 2146 int64 physical slot ids, or 17,168 bytes.
$\mathbf{g}_t$	$H_Q = 32$, $H_{\mathrm{KV}} = 8$, $d_h = d_v = 128$.	The visible K cache has logical shape $[2146, 8, 128]$, and V has the same shape. At two bytes per value, K plus V reads account for $2146 \cdot 8 \cdot 128 \cdot 2 \cdot 2 = 8{,}790{,}016$ bytes before cache-line, reuse, or split effects.
τ_{tile} and Δ_{dtype}	A split decode path streams K/V tiles while keeping FP32 softmax scratch.	With configured split capacity $S_{\max}$, Triton allocates `attn_logits` with shape $[2, 32, S_{\max}, 128]$, `attn_lse` with shape $[2, 32, S_{\max}]$, and `num_kv_splits` as `int32[2]`. For $S_{\max} = 4$, those scratch buffers occupy 132,104 bytes.
κ as page/-workspace view	A CUTLASS MLA-style page-table view uses 128-token pages and pads to the longest row.	$\lceil 2049/128 \rceil = 17$, so `block_kv_indices` has shape $[2, 17]$ in int32, or 136 bytes including padding. The opaque workspace is a `uint8` buffer sized by `cutlass_mla_get_workspace_size` with arguments 2176, 2, and `num_kv_splits=1`; $2176 = 17 \cdot 128$ is the longest sequence length rounded to page-block geometry. It changes launch legality, not request ownership.

27.3. Attention Metadata: ForwardBatch, KV Writes, and Row Ownership

Kernel inputs are bundles of tensors, layout, and launch metadata.

A kernel never sees "request A" directly. It sees tensors, offsets, lengths, cache locations, and output slots that must still denote A. `ForwardBatch` ties an attention launch back to the scheduler's row order. It carries request-pool rows and sequence state, then adds positions, cache-write locations, and optional speculative state. `AttentionBackend` then builds the ordinary forward view plus the CUDA-graph capture and replay views consumed by the backend path. CUDA-graph capture records a fixed sequence of launches once, and replay submits that recorded graph cheaply for a later shape bucket; the bucket is safe only when its buffers and output slice are refilled for the current request rows. Attention backend code therefore belongs in kernel reasoning: the launch is correct only when those model-executor records denote the same request state the scheduler selected. Those records instantiate the Volume I KV-cache-layout definition at the kernel boundary.

For one scheduled row i, write the kernel-visible metadata bundle as

$$M_i = (r_i, p_i, \ell_i, s_i, b_i, a_i),$$

where r_i is request identity, p_i is the position account, ℓ_i is the cache-write location, s_i is any scale record, b_i is backend metadata, and a_i is the launch argument view. Source fields such as `cache_loc` are concrete carriers of this bundle, not independent ownership facts.

Example 27.3.1 (One scheduled row at the kernel boundary) Suppose the scheduler admits request A's next decode row. In scheduler space, A is a request row with a sequence length, position id, and reserved cache location. At the kernel boundary, the same row is represented by M_A: forward-batch fields, cache location, scale metadata when the cache is quantized, backend metadata, and launch arguments. The row identity is correct only if every component of M_A still points to A's physical cache slot and output slice. A launch can have legal tensor shapes and still be wrong if one metadata record names another request's slot. Figure 27.3 draws the same ownership chain from scheduler row to kernel-visible state.

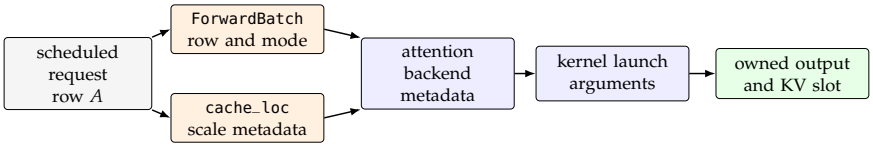

Figure 27.3.: Row ownership survives the kernel boundary only when scheduler state, forward-batch metadata, cache locations, scale records, and launch arguments all name the same request row and physical storage.

Row-ownership source concordance

- **Runtime row metadata.** `python/sglang/srt/layers/attention/` connects runtime attention metadata to backend implementations, including the forward-batch and backend metadata views that carry request rows into launches.

- **Attention launch implementations.** `sgl-kernel/csrc/attention/` contains lower-level attention kernels, including the CUTLASS MLA decode implementation behind `cutlass_mla_decode`.
- **Scaled GEMM and quantized variants.** `sgl-kernel/csrc/gemm/` and `sgl-kernel/csrc/gemm/fp8_blockwise_gemm_kernel.cu` contain GEMM kernels whose launch records must preserve scale layout and tile ownership.
- **Correctness tests.** `sgl-kernel/tests/test_flash_attention.py` and `sgl-kernel/tests/test_fp8_blockwise_gemm.py` check attention and blockwise FP8 GEMM behavior against reference expectations.

In this map, GEMM means general matrix multiplication. FP8 names an 8-bit floating-point data path with explicit scale metadata; E4M3 is the common FP8 format with 4 exponent bits and 3 mantissa bits. A blockwise scale is one scale shared by a tile of stored values, and a zero-point is the integer offset used by asymmetric quantization. W4A8 means 4-bit weights and 8-bit activations. The quantization chapter gives the full scale-layout account. MLA means multi-head latent attention, introduced in the attention-family chapter in Volume I. CUTLASS and FlashMLA name kernel families rather than request semantics.

The runtime Triton decode attention implementation used here lives under `triton_ops/`. The serving anchor is therefore the runtime attention backend and its Triton ops package.

The Triton ops package also contains helper kernels whose job is not to compute attention, but to write cache state in the representation that later attention will consume. For an FP8 cache write, the helper must preserve the same metadata bundle M_i: layout, optional K/V scales, byte-storage view, and physical cache location. In source, `trtllm_fp8_kv_kernel.py` defines `fused_fp8_set_kv_buffer`, its Triton kernel, and a naive fallback for writing K/V tensors into FP8 paged cache. That source path supports byte views for FP8 storage compatibility and writes under the same page-size contract as the ordinary KV-buffer path.

Read the Triton grid here with the GPU vocabulary from the Volume II preliminaries: a program block owns a small tile of the token, KV-head, and K-or-V side space. That block may visit physical cache bytes in a different order from request order, but it must use `cache_loc` to recover the same flat slot or (page, offset) that the scheduler reserved. In the notation above, that field is ℓ_i. Triton is therefore the launch language for the tiled store, not a different cache semantics.

Algorithm 30 writes FP8 KV-cache data by validating scale metadata and layout before storing quantized keys and values into the page-owned slots. It first validates the cache layout and normalizes the K/V views. It then prepares optional inverse scales. When the fused Triton path is available, that path maps cache locations and stores FP8 values; otherwise the fallback quantize-and-write path preserves the same physical-location contract.

The serving hazard is broader than this one helper. A quantized cache write saves resident bytes only when the later attention read can consume the stored representation without moving the cost somewhere worse. Table 27.7 keeps the backend question visible.

Algorithm 30: FP8 KV-cache write helper. Reader takeaway: a decode append is correct only when the right K/V value is written to the right physical slot under the right scale convention.

Input: Key tensor K; value tensor V; FP8 K/V caches; cache locations ℓ; optional scales s_K, s_V; page size P
Output: FP8 cache entries written at the physical locations ℓ

validate that the cache is either flat-slot layout or paged layout with page size P;
normalize K and V to [tokens, kv heads, head dim] views and derive cache strides;
if *both scales are provided* **then**
    create GPU-resident inverse-scale scalars $1/s_K$ and $1/s_V$;
if *the Triton path is enabled and the token count is nonzero* **then**
    launch a fused grid over tokens, KV-head blocks, and the K/V side;
    foreach *program block* **do**
        map ℓ to a flat slot or (page, offset);
        apply the inverse scale when present, convert to FP8 storage, and store;
else
    quantize K, V with the provided scales when the cache dtype requires it;
    view FP8 tensors as `uint8` if the cache stores FP8 bytes that way;
    write the tensors by ℓ, using page and offset for paged caches;

Figure 27.4 gives the same address calculation visually. A logical token row becomes a physical slot through either a flat index or a paged lookup; padding entries such as -1 are not extra visible tokens.

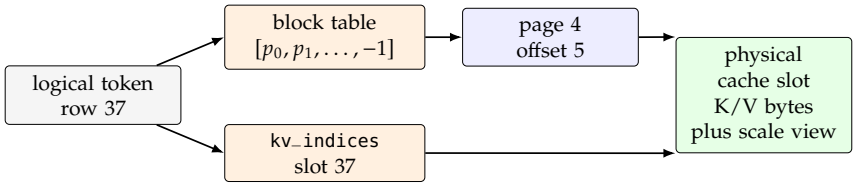

Figure 27.4.: Paged KV metadata maps logical token rows to physical cache slots. The -1 block-table entry is padding; it must not become visible history.

Algorithm 30 answers the common decode-cache question directly: a decode step appends a new KV entry, but the append is a typed, scaled, layout-aware cache write rather than a logical list push. The helper's Triton and naive paths preserve the same page-location contract, which is what lets later attention kernels read the entry through ordinary KV metadata.

Attention-metadata implementation anchors

These attention anchors show two concrete metadata shapes at the serving boundary. In `TritonAttnBackend.init_forward_metadata`, decode metadata includes `kv_indptr`, `kv_indices`, split counts, log-sum-exp scratch, and optional sliding-window buffers. In `CutlassMLABackend`, decode for multi-head latent attention (MLA) uses a fixed page size, builds `block_kv_indices`, obtains a CUTLASS MLA workspace size, and passes the page table plus workspace into `cutlass_mla_decode`, whose CUDA implementation is `sgl-kernel/csrc/attention/cutlass_mla_kernel.cu`. Both paths implement attention, but their serving kernel shapes are different because their metadata, cache layouts, and workspace requirements differ.

The recurring terms are compact. `kv_indptr/kv_indices` encode a ragged KV lookup as prefix-sum offsets plus a flat physical-slot list. Split counts partition a long attention read across kernel work units. Log-sum-exp scratch carries stable

Table 27.7.: Quantized KV-cache read risk. Cache dtype changes both the resident bytes and the attention-backend contract; correctness requires the same request history under the same scale interpretation.

Cache dtype	Write path	Later read path	Risk
BF16 or model dtype	Store K/V rows in the ordinary cache view.	Attention reads the same dtype expected by the backend.	Capacity is limited by two-byte elements.
FP8	Apply the selected K/V scale convention and store FP8 bytes.	A compatible backend reads FP8 directly, or a fallback dequantizes before attention.	Missing or stale scales preserve shape but change values.
FP4	Store packed FP4 data plus scale-side buffers or method-specific scale state.	The backend consumes a supported FP4 path or reads from a dequantization workspace.	Scale buffers and dequantization can erase part of the memory win.

Table 27.8.: Quantized KV write and read trace. The rows keep token ownership aligned with the representation later decode reads.

Token row	Cache write view	Representation	Later decode read
A's new row	$\ell = 37$, page 4, offset 5 for $P = 8$.	K/V converted with the recorded FP8 scale convention and stored through the cache dtype view.	`kv_indices` contains slot 37, so attention reads the value as part of A's history.

softmax reductions across splits or chunks. Sliding-window buffers translate full-history positions to local-window positions. Workspace is temporary backend storage, not request-owned state.

The ordinary Triton decode path can therefore be read as a field trace rather than as a list of tensor names. The forward batch identifies the live request rows and their cache state; backend metadata construction turns that state into prefix sums, flat KV-slot lists, split counts, and scratch buffers; the decode launch then consumes those fields. Table 27.9 shows the trace for the fields a reader should check first.

GEMM launch implementation anchors

The GEMM anchors show the same contract from the matrix-multiply side. `fp8_blockwise_scaled_mm` checks CUDA residency, row-major and column-major strides, FP8 E4M3 inputs, output dtype, 16-byte alignment, and scale shapes in `sgl-kernel/csrc/gemm/fp8_blockwise_gemm_kernel.cu` before dispatching SM90, SM100, or SM120 CUTLASS paths. The QServe W4A8 files encode another family of packed-weight and scale layouts. These kernels are not interchangeable merely because they compute AB: the runtime must supply the dtype, packing, scale granularity, and alignment that the kernel was written to consume.

The tests are part of the evidence because they expose the intended contract. The flash-attention test `test_flash_attention.py` compares cache, page-table, rotary, left-padding, variable-length, local-window, multi-query/grouped-query attention (MQA/GQA), and FP8 cases against a reference attention computation. The FP8 blockwise-GEMM test `test_fp8_blockwise_gemm.py` compares the kernel against a scaled-matrix baseline across many M, N, K shapes and both FP16 and BF16 outputs with explicit tolerances. These files are evidence for what must remain invariant without becoming a file-order checklist.

ForwardBatch state	Triton metadata	Kernel-facing meaning	Failure if stale
`req_pool_indices` with `req_to_token_pool.req_to_token`	`kv_indices` built by `create_flashinfer_kv_indices_triton`.	Maps each scheduled request row to the physical KV slots that form its visible history.	The query reads another request's cache rows.
`seq_lens` and `seq_lens_sum`	`kv_indptr` and `num_kv_splits`.	Bounds each ragged KV segment and chooses how long reads are split across work units.	The kernel overreads, underreads, or reduces the wrong split range.
`out_cache_loc` with `token_to_kv_pool`	`set_kv_buffer` before the decode launch; later reads through `kv_indices`.	Stores the new K/V vectors in the physical slots that subsequent attention will address.	The next step misses the token or reads a value stored for another row.
Sliding-window or speculative fields	`window_kv_indptr/window_kv_indices` or `spec_info.kv_indptr/spec_info.kv_indices`.	Replaces the full-history view with the local-window or draft/verify visibility that the mode selected.	The tensor shape survives while the visible history changes.

Table 27.9.: ForwardBatch-to-Triton metadata trace. The check is whether each runtime field still denotes the scheduled request rows when the Triton decode kernel consumes the compact KV view.

27.4. Attention Tile State and Online Softmax

Attention launch families differ in page tables, split metadata, and workspace, but the row-local mathematical state is smaller: a running maximum, a running normalizer, and an output accumulator. This section uses (m, ℓ, u) only for that online-softmax state. The full serving launch remains S_t from Definition 27.2.3.

Online softmax keeps a tile-local summary of the same causal attention row.

The attention tile state is small but semantically loaded. For one query row or query block, the kernel keeps a running maximum m, a running normalizer ℓ, and an unnormalized output accumulator u. When the next visible score tile $s_j = qk_j$ and value tile v_j arrive, masked-out positions behave as $-\infty$ scores and the online update is

$$m^+ = \max(m, \max_j s_j), \qquad \ell^+ = e^{m-m^+}\ell + \sum_j e^{s_j - m^+},$$

$$u^+ = e^{m-m^+}u + \sum_j e^{s_j - m^+} v_j.$$

The final output is u^+/ℓ^+ after all legal K/V tiles have been visited. This is the mechanism behind the phrase "maintains softmax state across tiles": the kernel avoids writing a full score matrix while the running max, normalizer, output accumulator, masks, and segment boundaries continue to describe the same query rows [2, 44].

Example 27.4.1 (Two-tile online softmax) Take one query row, scalar values, and two score tiles. The first tile has scores $[1, 0]$ and values $[2, 4]$. Starting from $m = -\infty$, $\ell = 0$, and $u = 0$, the first tile gives $m = 1$, $\ell = 1 + e^{-1} \approx 1.368$, and $u = 2 + 4e^{-1} \approx 3.472$. The second tile has score $[2]$ and value $[6]$, so the maximum rises to $m^+ = 2$. The old state is rescaled by $e^{1-2} = e^{-1}$: $\ell^+ = e^{-1}(1.368) + 1 \approx 1.503$ and $u^+ = e^{-1}(3.472) + 6 \approx 7.277$. The final output is $u^+/\ell^+ \approx 4.84$, matching the ordinary softmax over all three scores without ever materializing the full score row.

This small example proves that online softmax can reproduce the ordinary softmax row while streaming tiles. It does not license mixing state between request rows; the registers still belong to one query row until the final write.

Example 27.4.2 (Four-query online-softmax tile trace) For the decode half of the running example, take four query rows attending over sixteen visible KV slots, split into four 4-slot tiles. Use scalar FP32 values $v_j = 0.1j$ only to make the accumulator readable; a real head carries a vector value at each slot. For query q_0, the tile scores are $[1.0, 0.0, -1.0, 0.5]$, $[1.5, -0.5, 0.25, 0.75]$, $[0.0, 1.25, -0.25, 0.5]$, and $[2.0, 1.0, 0.0, -1.0]$. The online state after tile 0 is $m = 1.0000$, $\ell = 2.1097$, $u = 0.2458$, and $u/\ell = 0.1165$. After tile 1 it is $m = 1.5000$, $\ell = 3.1738$, $u = 1.1193$, and $u/\ell = 0.3527$. After tile 2 it is $m = 1.5000$, $\ell = 4.7174$, $u = 2.5772$, and $u/\ell = 0.5463$. After tile 3 it is $m = 2.0000$, $\ell = 4.4143$, $u = 3.5055$, and $u/\ell = 0.7941$.
The other three query rows carry their own (m, ℓ, u) state in parallel: their final $(m, \ell, u/\ell)$ values are $(0.8, 10.5243, 0.8364)$, $(1.75, 4.9755, 0.9271)$, and $(2.0, 4.4205, 0.6338)$. This is the 4-query by 16-KV-slot version of the same update rule: tile order changes storage traffic, but each query row keeps its own max, normalizer, and accumulator.

This larger trace proves that the same three-field state scales from one toy row to multiple query rows. It does not claim a particular tile order is fastest; it shows which state must survive any tile order the backend chooses.

Figure 27.5 turns the same numeric walkthrough into the state that travels from tile to tile.

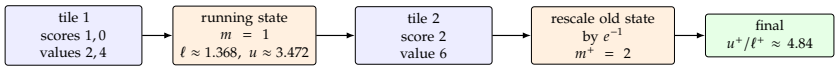

Figure 27.5.: Online softmax carries only the running max, normalizer, and output accumulator across tiles. When a later tile raises the maximum, the previous state is rescaled before the new tile contributes.

In the CUDA-style listing below, a warp is a group of 32 lanes executing the same instruction stream. The primitive shfl_xor(t, δ) means that each lane reads the value t from the lane whose id differs by the bit mask δ. The masks $16, 8, 4, 2, 1$ therefore implement a butterfly reduction across all 32 lanes.

The shuffle masks $16, 8, 4, 2, 1$ progressively exchange candidates across half-warps, 8-lane groups, 4-lane groups, lane pairs, and adjacent lanes; after the final exchange, all lanes have seen the warp-wide maximum. Algorithm 31 exposes the register-level state hidden behind the launch-family descriptions.

Algorithm 31: CUDA-style decode-attention warp loop. The listing is explanatory pseudocode for the online-softmax state; SGLang launch families may implement the same state with Triton, FA3, FlashMLA, or CUTLASS-specific schedules.

Input: One CTA's query row group Q; paged K/V slots for one decode row; tile width $B_k = 16$; value width d_h
Output: Attention output for the query row group

assign one warp to one query row and its visible K/V stream;
initialize register state $m \leftarrow -\infty$, $\ell \leftarrow 0$, and vector accumulator $u \leftarrow 0$;
foreach *K/V tile T of 16 visible slots* **do**
    lane λ loads its assigned q components and K/V components for tile T;
    compute one or more partial dot products $s_\lambda = q \cdot k$ in registers;
    set masked or out-of-range scores to $-\infty$;
    $t \leftarrow s_\lambda$;
    for $\delta \in \{16, 8, 4, 2, 1\}$ **do**
        $t \leftarrow \max(t, \text{shfl_xor}(t, \delta))$;
    set $m_T \leftarrow t$, the warp-wide maximum for this tile;
    $m^+ \leftarrow \max(m, m_T)$;
    rescale old state: $u \leftarrow \exp(m - m^+)u$ and $\ell \leftarrow \exp(m - m^+)\ell$;
    compute $a_\lambda = \exp(s_\lambda - m^+)$ for this lane's valid scores;
    reduce the lane-local a_λ values across the warp to update ℓ;
    accumulate $u \leftarrow u + \sum_{\lambda \in \text{warp}} a_\lambda v_\lambda$ in registers;
    set $m \leftarrow m^+$;
divide u by ℓ and write the row-aligned output;

The important serving fact is still row ownership: the warp may visit K/V tiles in memory order and reduce across lanes, but its (m, ℓ, u) registers belong to one query row group until the final write.

A single occupancy line shows why that register state is part of the launch account. On an NVIDIA H100-class SM, use 65,536 32-bit registers and a 2,048-thread resident limit as a budgeting row [25]. A decode CTA with 128 threads and 64 registers per thread consumes

$$128 \cdot 64 = 8{,}192$$

registers, so the register file admits $\lfloor 65{,}536/8{,}192 \rfloor = 8$ resident CTAs, or 32 resident warps. The thread limit would admit $\lfloor 2{,}048/128 \rfloor = 16$ CTAs, or 64 warps, so registers are the tighter bound in this sketch. The scalar m and ℓ state is small, but the vector accumulator u grows with value-fragment ownership; more query heads or larger head fragments can raise registers per thread and lower the resident-warp count without changing the logical attention formula.

The worked rows prove why an attention kernel may change memory traversal order without changing request semantics. They do not prove that one family is faster; they define the state that the launch-family contracts in the next section must preserve.

27.5. Attention Launch-Family Contracts

The scheduler fixes request semantics before an attention backend chooses a physical itinerary. The online-softmax state from Section 27.4 explains what travels with each query row. This section explains which launch-family metadata turns those rows into a legal Triton, FA3, FlashMLA, or CUTLASS MLA launch.

Table 27.10.: Attention launch families and ownership invariants. Each family represents a launch decision that must preserve request ownership.

Family	Decision it represents	Ownership invariant
Triton decode	Use a flat-index decode view when ordinary paged K/V rows and split buffers are the executable representation.	`kv_indptr` and `kv_indices` may change traversal order, but row i still writes the output for request row i.
CUTLASS MLA	Use a fixed-block MLA page table and workspace when latent-cache decode should run through a CUTLASS-backed path.	Padding and workspace sizing shape the launch, not the request histories that own the returned rows.
FA3	Use cumulative-length and scheduler metadata when ragged decode or prefill segments should become a FlashAttention-family tile itinerary.	Segment boundaries, masks, and optional page tables preserve the same causal visibility as the scheduled batch.
FlashMLA	Use latent-KV block tables and split metadata when MLA decode or prefill should read compressed cache pages directly.	Latent page order and split counts may change physical reads, but not which request owns each attention output.

Definition 27.5.1 (Launch family) *A launch family is a reusable backend path whose entrypoints, metadata builder, tile schedule, workspace rules, and fallback behavior are shared across a class of serving kernel shapes.*

Decode attention is dominated by KV interpretation and memory traffic.

In the decode-heavy half of the running example, SGLang has already chosen the active requests. The attention backend now has to make them executable. A Triton decode path turns scheduled rows $\mathcal{B}_t$ into contiguous KV index arrays, using the row-to-cache relation from the KV cache chapter in Volume I. A CUTLASS MLA path instead builds a fixed-block page table and a workspace sized for the padded maximum sequence length. Both paths can visit memory in tile order rather than request order; neither may change which request receives the output [32, 27].

The rows in this table are launch families in the sense of Definition 27.5.1: each one packages entrypoints, metadata construction, tile scheduling, workspace ownership, and fallback behavior for a class of serving kernel shapes. Their shared contract is the reader's through-line for this section: the physical itinerary may change, but causal visibility and row-aligned output may not.

FA3 and FlashMLA entry paths.

FA3 abbreviates FlashAttention-3. At this chapter's level, FA3 is a FlashAttention-family exact-attention launch path that tiles the query-by-key work, maintains softmax state across tiles, and avoids materializing the full attention matrix. FlashMLA names the kernel path for multi-head latent attention (MLA): it reads compressed latent KV pages through block-table metadata rather than ordinary K/V head rows.

FA3 and FlashMLA are important enough to name separately because they expose two different answers to the same serving question: how does a ragged request batch become a legal tiled-attention launch? The motivation for FA3 is IO-aware attention over query and KV segments whose lengths, masks, optional page tables, and rotary state are already fixed by the scheduler. The motivation for FlashMLA is the MLA case, where the readable cache is a latent page table plus

positional state rather than ordinary K/V rows. Both kernels can be fast only after SGLang has made that metadata executable [2, 44, 6].

The intuition is an itinerary. For a decode row with a long history, the kernel receives sequence lengths, page entries, split counts, scheduler metadata, and workspace instead of a Python request object. FA3 precomputes tile schedules that the forward operator can consume across layers. FlashMLA builds split metadata and reads latent KV through page tables. MLA-style prefill has separate dense and sparse entry paths depending on whether sparse indices select the visible cache neighborhood.

The FA3 `sgl_kernel.fwd` operator contract.

The source-level call path is short but load-bearing. The backend builds metadata, the Python wrapper normalizes optional defaults and contiguous views, and the registered schema binds the public PyTorch operator to the SGLang torch shim. Thus the public extension boundary for the FA3 forward path is a contract surface, not the name of one CUDA template or one serving mode.

The detailed field list is reference material for kernel implementers, so Section 27.10 collects it with the source-coverage material. The main reading path needs three checks: the payload names the query rows and K/V state; the segment and placement fields name legal request histories; and semantic or launch modifiers such as `softcap`, `sinks`, scheduler metadata, and split counts match the reference path.

Separating `get_scheduler_metadata` from `sgl_kernel.fwd` is therefore an execution-contract optimization. In ordinary decode, the backend can compute the FA3 tile itinerary while initializing forward metadata and reuse it across layers; CUDA graph paths can also preallocate a buffer for the same shape family. This saves repeated scheduler preparation, but it does not relax the correctness contract. If the cumulative lengths, cache sequence lengths, page table, or mask modifiers disagree with the scheduled batch, `sgl_kernel.fwd` will compute quickly for the wrong attention problem.

The FlashMLA decode operator contract.

The analogous FlashMLA boundary is the latent-cache decode wrapper. Its metadata initializer builds a 64-token block table from the position-to-slot map and stores scheduler metadata, split prefix sums, and the block table in the decode metadata object. During decode, the backend writes newly produced layer state when needed, obtains the layer's latent cache buffer, reshapes the query rows, and dispatches either to the ordinary latent-cache decode operator or to the dense FP8 query-and-cache variant; the registered schemas live in the FlashMLA extension source.

The detailed FlashMLA field groups are also reference material and are collected with the source-coverage material later in the chapter. The durable reader question is not whether `tile_scheduler_metadata` or `descale_q` appears in a particular row, but which part of the attention contract that field protects: payload ownership, request-history placement, tile scheduling, semantic modifiers, FP8 scale state, or return alignment.

The failure modes follow from those field groups. A stale block table can make a decode row attend to another request's latent pages. A stale sequence-length record can overread or underread the visible history. Split metadata that does not match the sequence lengths can reduce the wrong ranges. In the FP8 path, mismatched query and cache descaling tensors change the numerical attention problem; in the sparse path, the index vector owns visibility, so the wrapper rejects combining sparse selection with causal masking.

Figure 27.6 summarizes the shared row-aligned output obligation behind the FA3 and FlashMLA metadata paths.

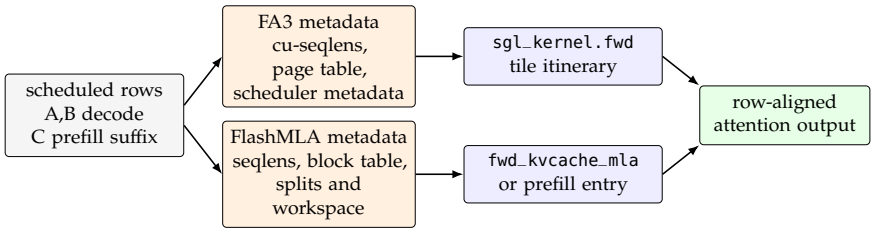

Figure 27.6.: FA3 and FlashMLA use different kernel metadata for the same serving obligation. FA3 turns ragged/page-table attention into a scheduled tile itinerary; FlashMLA turns latent-cache pages and split counts into an MLA decode or prefill launch.

Example 27.5.1 (Triton and FA3 boundary views) Suppose requests A and B each decode one token with histories 96 and 2048, while request C has a 512-token prefix and contributes a 128-token suffix. Assume the decode K/V rows have already been written before attention. The Triton decode view for A and B concatenates each selected request row's physical cache slots into a flat cache-index array and records row boundaries separately. The toy record uses `seq_lens` $[97, 2049]$ and `kv_indptr` $[0, 97, 2146]$. The split-softmax buffers `attn_logits` and `attn_lse` may divide the long read, but they are scratch, not request-owned state.
A valid FA3 view preserves the same logical segments for A, B, and C while changing the entrypoint-facing metadata. One toy FA3-facing record has

$$\mathrm{cu}_q = [0, 1, 2, 130], \qquad \mathrm{cu}_k = [0, 97, 2146, 2786].$$

Those cumulative lengths say that the two decode queries and the 128-query prefill suffix remain separate segments. Tiles may visit memory in another order, but the decode outputs still belong to A and B, and the suffix queries for C see only the prefix plus earlier suffix positions.

This boundary example proves that backend metadata can change the entrypoint view without changing request ownership. It does not claim FA3 and Triton have the same cost model; it isolates the row-alignment contract they must share.

Metadata view	Toy record	Failure guarded
Triton decode	`seq_lens` [97, 2049]; `kv_indptr` [0, 97, 2146].	Flat indices may split reads, but row 0 writes A's output and row 1 writes B's output.
FA3 varlen	$\mathrm{cu}_q = [0, 1, 2, 130]$; $\mathrm{cu}_k = [0, 97, 2146, 2786]$.	Cumulative boundaries preserve two decode rows plus C's causal prefill suffix.
FlashMLA decode	`seqlens_k` [97, 2049]; `num_splits` [0, 1, 5].	Latent page-table and split metadata cover the same two decode histories without merging them.
CUTLASS MLA decode	Sequence lengths [97, 2049]; padded block count 17; workspace input 2176, 2, 1.	Padding and fixed-size page-table rows shape the launch, not request ownership.

Table 27.11.: Metadata views for row-alignment checks. Each toy record shows the failure a launch-family boundary must guard.

Example 27.5.2 (MLA page-table views) For the same decode rows, FlashMLA passes `seqlens_k`, a latent `block_table`, tile scheduler metadata, and split boundaries to the MLA entry point. With a 64-token decode page size, A's 97 visible rows occupy two latent pages and B's 2049 visible rows occupy thirty-three. If B's long history receives more split slots, that is still an execution choice, not permission to merge A's and B's logical histories.
FlashMLA prefill answers a related but different question: which suffix queries and visible latent rows should enter the prefill kernel? Dense prefill presents request C's 128 suffix queries and 640 visible latent rows as one varlen segment; sparse prefill presents the same request-owned suffix through explicit sparse indices. CUTLASS MLA decode uses the same selected rows as FlashMLA decode but builds a fixed-size page table and sizes a workspace from the padded maximum sequence length. In all three cases, the metadata may change physical reads and padding, but it must still encode legal causal visibility and the same request row ownership.

This MLA example proves that latent page tables, split counts, and workspace sizing are launch metadata for the same scheduled rows. It does not make FlashMLA, sparse prefill, and CUTLASS MLA interchangeable performance baselines.

Implementation constraints that leak into serving.

The book does not need to reproduce the CUDA templates behind these kernels, but a few implementation facts are part of the serving contract because they change backend eligibility, page geometry, or the tensors a profile will show. FlashMLA decode uses 64-token pages with `seqlens_k`, a latent `block_table`, scheduler metadata, and split counts. CUTLASS MLA uses 128-token pages, builds `block_kv_indices`, and allocates workspace from padded maximum sequence length, batch size, and `num_kv_splits`. The exposed CUTLASS MLA wrapper also checks DeepSeek-style latent dimensions such as $D_{\text{latent}} = 512$, $D_{\text{rope}} = 64$, and query-head limits.

SMNN names an NVIDIA GPU architecture generation; a tensor-core kernel is written against one generation's instruction and scheduling rules, so the runtime must choose a variant the device supports. That is why the CUTLASS MLA CUDA path can require a CUDA build new enough for the kernel and gate the

in-tree decode implementation to SM100-class execution, while FlashMLA has a separate FP8 decode dispatch with descaling tensors. Helper kernels such as `concat_mla_k` and `concat_mla_absorb_q` are visible for the same reason: they do not change the attention rule, but they determine whether a profile materializes MHA-shaped K/V state or keeps the latent form.

These constraints are the appropriate low-level boundary for this chapter. The reader should understand why `flashmla`, `cutlass_mla`, FA3, or a fallback backend may be selected for the same logical attention step. The default-selection policy itself remains the job of the attention-backends chapter: its server-argument table explains why specialized names such as `flashmla` and `cutlass_mla` may be legal choices without being the generic MLA default. Lower-level kernel-engineering details such as warp scheduling, TMA, PTX/SASS, fatbins, cubins, or driver JIT compilation enter only when they change a visible launch contract, architecture eligibility, first-use latency, benchmark reproducibility, or fallback decision.

27.6. GEMM Tile Loop and Shared-Memory Staging

A GEMM launch has its own physical hierarchy. This section uses CTA tile, warp fragment, shared-memory stage, and epilogue slice for the kernel mechanics. The next section uses *launch record* for quantized metadata such as packed weights, scale layouts, padding, and architecture dispatch.

GEMM staging changes data movement, not the matrix product the layer owes.

GEMM tile loop and scale tiles.

For a dense $C = AB$ launch with $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$, the physical kernel usually decomposes C into CTA tiles such as $M_b \times N_b$, then walks the shared K dimension in K_b-sized chunks. Each iteration brings an $A[M_b, K_b]$ tile and a $B[K_b, N_b]$ tile through global memory, packed memory, or on-chip staging, warps map subtiles to tensor-core MMA fragments, and registers hold partial C accumulators until the epilogue writes or slices the final output. The serving runtime sees the same logical M, N, K, but the kernel is legal only if the physical tile loop agrees with layout, alignment, dtype, accumulator, and epilogue rules [27].

Figure 27.7 shows the K-loop visually: activation and weight tiles feed accumulator blocks before the epilogue writes the output tile.

First, the CTA maps its block coordinates to a fixed output tile and divides that tile among warp-owned subtiles. Then, the mainloop alternates shared-memory stages so the next A/B tile can be prefetched while the current stage feeds tensor-core fragments. Finally, the epilogue applies the selected scale, bias, activation, and dtype rules before storing the tile, which is why dispatch must bind the launch record to a legal layout and scale record.

The illustrative eight-warp layout is a 2×4 partition of the CTA output tile: each warp owns one 64×32 subtile until the epilogue. The exact partition can vary by kernel family, but the serving-visible rule does not: a warp may own a physical fragment, while the CTA still owns the logical output rectangle that will be sliced back to request rows.

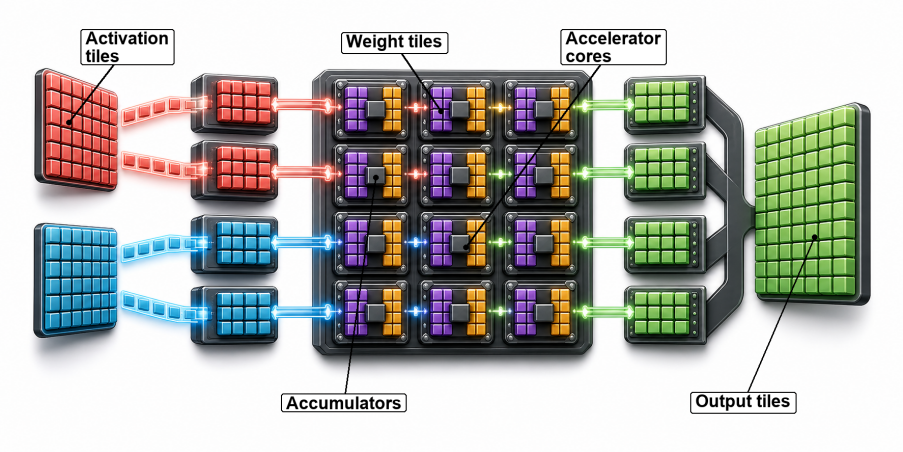

Figure 27.7.: GEMM tiling on accelerator cores. Matrix tiles flow through accelerator compute units, with input activation tiles, weight tiles, accumulation blocks, and output tiles arranged along the K-loop data path.

GEMM tile object	Kernel role	Request-visible check
CTA tile $M_b \times N_b$	Owns one output rectangle and chooses the epilogue slice.	Padded rows and columns must be hidden from request-owned output.
K_b operand tile	Controls reuse of activation and weight fragments.	Packed strides and alignment must describe the same logical K.
Warp or MMA fragment	Maps subtiles to tensor-core instructions and register accumulators.	Accumulator and output dtype must match the tolerance policy.
Scale tile or scale vector	Supplies blockwise FP8, W4A8, or INT8 dequantization metadata.	Scale grouping must match the selected kernel's tile interpretation.

Table 27.12.: GEMM tile objects and kernel roles. Each object has a request-visible check that keeps tiled work tied to the requested operation.

Algorithm 32: CTA GEMM tiling sketch. The concrete tile sizes are illustrative of a tensor-core mainloop; SGLang dispatch still chooses only legal templates for the actual dtype, layout, architecture, and scale record.

Input: Logical GEMM $C = AB$; CTA output tile 128×128; K-chunk $B_k = 32$; eight warps
Output: One 128×128 output tile

map CTA (b_m, b_n) to output rows $128b_m$:$128b_m + 127$ and columns $128b_n$:$128b_n + 127$;
assign eight warps to a 2×4 grid of 64×32 output subtiles;
allocate two shared-memory stages for $A[128, 32]$ and $B[32, 128]$;
prefetch the first A and B tiles from global or packed memory into stage 0;
for $k_0 = 0; k_0 < K; k_0 \leftarrow k_0 + 32$ **do**
    prefetch the next A/B tile into the other shared-memory stage when it exists;
    synchronize the CTA so the current stage is visible;
    foreach *warp-owned* 64×32 *subtile* **do**
        load 16×16 accumulator fragments for that subtile into registers;
        for $k_f = 0; k_f < 32; k_f \leftarrow k_f + 16$ **do**
            load A and B fragments for a $16 \times 16 \times 16$ MMA;
            execute one `mma.sync`-style fragment multiply accumulate;
    swap the shared-memory stage index;
apply scale, bias, activation, or dtype epilogue rules and store the CTA tile;

The sketch in Algorithm 32 is the low-level version of the launch-record contract. The CTA owns a stable output rectangle, the K-loop streams operand tiles through shared memory, and each warp accumulates fragments in registers. Double buffering hides some global-memory latency only when the operand layout and shared-memory staging avoid conflicts and satisfy the selected MMA template.

Example 27.6.1 (Shared-memory bank conflict in a 32×32 GEMM tile) The smallest bank conflict example is a 32×32 staged tile with 32 shared-memory banks and 4-byte bank words. Let lane λ write logical tile coordinate $(r, c) = (0, \lambda)$. The logical tile is the same in every row below; only the shared-memory address calculation changes. This is the conflicting-versus-conflict-free layout check hidden behind the shorter phrase "double buffering hides latency only when staging avoids conflicts."
In a naive column-major tile with leading dimension 32, the word index is $w = 32c + r = 32\lambda$, so bank $= w \bmod 32 = 0$. All 32 lanes target one bank, and the warp is serialized. With a +1 padded column-major tile and leading dimension 33, the word index is $w = 33c + r = 33\lambda$, so bank $= w \bmod 32 = \lambda$. Lanes 0:31 distribute across banks 0:31. With a swizzled row index over the same logical tile, $w = 32c + (r \oplus c)$; for $r = 0$, bank $= \lambda$. The kernel restores the logical coordinate while using a conflict-free staging order.

This staging example proves that a physical tile layout can be performance critical while leaving the logical GEMM unchanged. It does not make the swizzle a request-visible transformation; the request-visible object is still the output rectangle owned by the CTA.

The tile-loop examples define the physical work hierarchy that a valid GEMM launch may use. They do not by themselves choose a quantized path; the scale and dtype record in the next section decides whether a rowwise FP8, blockwise FP8, INT8, W4A8, or fallback path is legal.

27.7. Quantized GEMM Launch Records and Scale Layouts

This section keeps S_t as the full serving kernel shape and uses *launch record* for the GEMM-side metadata that makes a packed or quantized matrix multiply legal. The logical product is still $C = AB$; the record says how the kernel interprets rows, columns, packed weights, scales, padding, output dtype, and architecture support.

Quantized GEMM correctness depends on scale layout matching the packed values.

In the prefill-heavy half, projection and feed-forward GEMMs see a different pressure point. The input token dimension is large enough to keep tensor cores busy, but only when layout and dtype match the implementation. Blockwise FP8 GEMM pads rows, carries per-block scale tensors, chooses an architecture-specific CUTLASS dispatch, and slices the padded output back to the original row count. W4A8 kernels add packed low-bit weights, zeros, and group or channel scales. The bytes change; the requested matrix product must not. W4A8 means 4-bit weights with 8-bit activations, and a zero-point is the integer offset used when an

Record field	Kernel-facing metadata	Failure if omitted
Scheduled token rows	Physical M' after padding plus the slice back to M.	Padded rows can be returned as if they were request tokens.
Packed weights	W4A8, FP8, or other packed layout plus zero-point policy.	The kernel may dequantize the wrong columns or groups.
Scale layout	Activation and weight scale tensors with their grouping convention.	The product can keep the right shape while applying scales to the wrong tile.
Architecture path	SM family, scheduler choice, alignment, and output dtype.	A path may be fast on one target and illegal or numerically different on another.

Table 27.13.: Quantized launch record fields. The kernel-facing metadata records the information that would otherwise be omitted from the launch.

asymmetric quantized value is reconstructed. A blockwise scale is one scale shared by a tile of values; the quantization chapter develops the full scale-layout model.

The lower-level dispatch makes that requirement concrete. FP8 GEMM entrypoints branch by SM family, then specialize for bias presence, scale-tensor layout, scheduler choice, and output dtype. INT8 GEMM has its own input, scale, and output constraints. Thus "CUTLASS-backed GEMM" is not a single implementation. The runtime presents a matrix problem plus scale records, bias state, architecture capability, and desired output type; SGLang chooses the legal kernel variant for that bundle. Here SMNN names an NVIDIA GPU architecture generation. A tensor-core GEMM kernel is written against a specific generation's instructions and layout rules, so a path that is legal on one device can be unavailable or numerically different on another.

Definition 27.7.1 (Quantized GEMM launch record) *A quantized GEMM launch record is the kernel-facing tuple of logical M, N, K, activation and output dtypes, packed-weight representation, scale and zero-point layout, optional bias, architecture dispatch, padding, and output slice that must still denote the scheduled rows of the requested matrix product.*

Example 27.7.1 (Scale-layout mismatch with a valid output shape) For request C's $M = 128, K = N = 4096$ projection, a blockwise FP8 path may expect activation scales grouped over 128-element K blocks and weight scales grouped over 128×128 tiles. A scale tensor grouped over 64-element blocks can still broadcast to a plausible output shape if checked too late, but it describes a different dequantization ledger. The launch record must reject that case or route it to a kernel whose scale layout matches.

Inside the FP8 scaled GEMM boundary.

SGLang's rowwise `fp8_scaled_mm` entrypoint is a fused scaled-GEMM boundary, not a generic matrix multiply. The caller supplies $A_q \in \text{FP8}^{M\times K}$ in row-major layout, $B_q \in \text{FP8}^{K\times N}$ in column-major layout, one FP32 activation scale per output row, one FP32 weight scale per output column, an optional bias vector, and an FP16 or BF16 output request. The CUDA entrypoint checks those shapes,

Table 27.14.: Quantized implementation choices. Each choice has an internal role and a request-visible condition for correctness.

Implementation choice	Kernel-internal role	Request-visible condition
Row and column scale vectors	Broadcast s_A over output columns and s_B over output rows in the epilogue.	The scale at address i still belongs to scheduled row i, and the scale at address j still belongs to output column j.
Architecture launch family	Selects SM89, SM90, SM100, or SM120 CUTLASS templates, with different tile and scheduler choices.	The fallback and eligibility record must name the actual device capability, not just the logical matrix shape.
Bias and output dtype branch	Installs an epilogue variant that either multiplies scales only or performs the scale-and-bias writeback.	The bias dtype and output dtype must agree before the kernel is allowed to run.

strides, dtypes, scale lengths, 16-byte alignment, and bias dtype before selecting an architecture-specific CUTLASS launch family [27]. The numerical contract is

$$Y_{ij} = \mathrm{cast}_{\mathrm{out}}\left(\left(\sum_{k=0}^{K-1} A_{q,ik} B_{q,kj}\right) s_{A,i} s_{B,j}\right) + b_j,$$

with $b_j = 0$ when no bias is attached. The important implementation point is where the scales are applied: the tensor-core mainloop accumulates the FP8 product, and the CUTLASS epilogue broadcasts the row and column scales while writing the FP16 or BF16 output. SGLang therefore avoids a separate dequantize-then-matmul sequence while preserving the same dequantized matrix product that the tests compare against.

The blockwise sibling, `fp8_blockwise_scaled_mm`, changes the scale addressing contract. Its activation scales are indexed by row and 128-wide K group, and its weight scales are indexed by 128×128 blocks. In index form, a product using inner coordinate k and output column n reads activation scale group $g_K = \lfloor k/128 \rfloor$ and weight block (g_K, g_N) where $g_N = \lfloor n/128 \rfloor$.

The kernel pads the activation rows and scale rows to the required alignment, dispatches through SM90, SM100, or SM120 blockwise CUTLASS paths, and slices the padded output back to the original M rows. In the Python runtime, `apply_fp8_linear` calls `fp8_scaled_mm` for compatible rowwise per-channel FP8 weights, while the block-FP8 dispatcher chooses among architecture-specific, vendor-tuned, and fallback GEMM paths according to the configured backend and current device. The custom part is therefore the whole executable boundary: activation quantization, scale layout, architecture eligibility, epilogue semantics, and fallback policy have to describe the same logical GEMM.

Packed values and scales travel together through the launch record even when kernels load them separately.

Example 27.7.2 (FP8 blockwise CTA scale use at $k = 256$) Suppose request C's projection uses $M = 128$, $K = N = 4096$, blockwise FP8 scales, and a CTA that owns output rows 0:63 and columns 128:255. When that CTA reaches the K-loop slice 256:383, the activation-scale group is $g_K = 256/128 = 2$, and the weight-scale block is $(g_K, g_N) = (2, 1)$. For output element $(17, 140)$, the mainloop consumes FP8 products from $A_q[17, 256{:}383]$ and $B_q[256{:}383, 140]$, then the epilogue applies the activation scale $s_A[17, 2]$ and the weight-block scale $s_B[2, 1]$. If $s_A[17, 2] = 0.03125$ and $s_B[2, 1] = 0.046875$, a raw accumulator of 512 contributes

$$512 \cdot 0.03125 \cdot 0.046875 = 0.75$$

to the dequantized partial sum for that output element. The next K-slice uses $s_A[17, 3]$ and $s_B[3, 1]$. Reusing $s_B[2, 0]$ would keep the output shape valid while applying the scale for columns 0:127, which is the wrong matrix product.

This ledger proves that scale indices are semantic metadata: a wrong block scale keeps the output shape plausible while changing the matrix product. It does not claim that this FP8 shape is generally faster than another GEMM path.

INT8 uses the same launch-record idea, but the scale convention is rowwise and columnwise rather than blockwise FP8. In SGLang's `int8_scaled_mm` interface, the caller supplies INT8 activation and weight matrices with row and column scale vectors: $(X_q, W_q, s_X, s_W, \text{out_dtype}, b)$. The CUDA entrypoint checks that both matrices are two-dimensional CUDA INT8 tensors, that K is compatible with tensor-core alignment, that N is aligned for the output stride, that the activation and weight scales are contiguous FP32 tensors of lengths M and N, and that any bias matches the output dtype. The tests compare the kernel with

$$Y_{ij} = \left(X_q W_q\right)_{ij} s_{X,i}\, s_{W,j} + b_j$$

for FP16 and BF16 outputs across many M, N, K shapes. The contrast with blockwise FP8 is the scale address: INT8 uses one activation scale per output row and one weight scale per output column, while blockwise FP8 uses scale tiles whose grouping must match the selected FP8 kernel.

Example 27.7.3 (INT8 row and column scale ledger) Suppose request C's projection is routed through INT8 scaled GEMM with $M = 128$, $K = N = 4096$. The activation quantizer produces $X_q \in \text{INT8}^{128\times4096}$ and a scale vector $s_X \in \mathbb{R}^{128}$. The loaded weight gives $W_q \in \text{INT8}^{4096\times4096}$ and $s_W \in \mathbb{R}^{4096}$. A launch with the right output shape can still be wrong if s_X is flattened from a padded token bucket, if s_W is indexed as a group scale rather than an output-channel scale, or if a BF16 bias is attached to an FP16 output request. The logical product is unchanged, but the launch record no longer describes the same dequantized matrix product.

Implementation bridge from wrapper to CUDA.

All three scaled-GEMM operators enter SGLang through the same implementation ladder. The public Python wrapper calls the corresponding custom op; the extension schema and CUDA binding connect that op to device code. The CUDA entrypoint then checks tensor ownership and scale metadata before handing the

Algorithm 33: Dispatching one attention or GEMM kernel view. The pseudocode distinguishes the logical operation from the packed, padded, or tiled kernel representation that the backend actually executes.

Input: Logical operator o; scheduled rows $\mathcal{B}$; sequence lengths ℓ_{seq}; KV locations or matrix operands X; dtype and scale metadata D; hardware capability h
Output: Logical attention or GEMM output attached to the original rows

```
validate that B, ℓ_seq, masks, operands, and requested dtype describe a legal logical operation;
if o is attention then
    build the compact kernel view: request-row ids, KV indices or page tables, split metadata,
      and output locations;
else
    pad or pack matrix operands as required by the selected GEMM family;
    attach scale, zero-point, bias, and output-dtype metadata;
choose a legal launch variant for h, D, tile shape, and layout;
launch the kernel over tiles, allowing memory traversal order to differ from request order;
trim padded rows or columns and map the result back to the logical request rows;
return the row-aligned output tensor;
```

launch to an architecture-specific CUTLASS family. The ladder is the important part for the main text: Python wrapper, registered custom-op schema, CUDA entrypoint, architecture-specific CUTLASS family, then a rowwise FP8, blockwise FP8, or INT8 reference test. The detailed operator/source mapping is collected with the source-coverage tables in Section 27.10; keeping it there lets the worked scale ledgers carry this section's narrative.

The dispatch ladder below is the same for attention and GEMM even though their metadata differ. First validate the logical request rows and model operation. Then build either attention metadata, such as page tables and split buffers, or GEMM metadata, such as packed operands and scale records. Only after that can the runtime filter by hardware, dtype, tile, and layout legality, launch tiled work, and trim or map the result back to the original request rows.

Algorithm 33 dispatches an attention or GEMM launch by matching operation shape, quantization metadata, backend-family capability, and fallback rules. It first separates the attention and GEMM cases and assembles their metadata. It then chooses a legal launch variant, maps padded results back to request rows, and returns the row-aligned tensor consumed by the next serving stage.

Algorithm 33 is deliberately written at the boundary between model math and kernel mechanics. The kernel may use page tables, packed weights, FP8 scale blocks, or architecture-specific dispatch, but the observable contract is still row-aligned attention or a row-aligned matrix product for the selected requests. For a first pass, keep the Volume II serving boundary in mind: validate the logical operation, metadata, scale policy, and output mapping before trying to read the CTA, warp-fragment, or CUTLASS-template details from Section 27.6. Those details explain the selected fast path; they do not change the row-aligned serving result.

For the prefill-heavy half of the running example, request C feeds $M = 128$ token rows into a projection with $K = N = 4096$. The logical dense work is

$$2MKN = 2 \cdot 128 \cdot 4096 \cdot 4096 = 4{,}294{,}967{,}296$$

floating-point operations. The FP16 activation input is $128 \cdot 4096 \cdot 2 = 1$ MiB, the FP16 output is another 1 MiB, and an FP16 weight matrix is 32 MiB before

scale or packing metadata. A W4A8 path changes the weight bytes and adds scale/zero metadata; it does not change the logical $M \times N$ output or the request rows that own those 128 tokens.

27.8. Launch Correctness: Invariants, Buckets, and Semantic Modifiers

The previous sections split the launch story into attention tile state, attention-family metadata, GEMM tile mechanics, and quantized GEMM records. This section has the narrower job: state the checks that cut across those launch families. A kernel may change tile order, storage order, padding, graph capture, or scale layout. It may not change the logical request rows, visible attention history, matrix product, semantic modifiers, or output ownership.

Buckets and modifiers are launch contracts; each must preserve the operator's semantic shape.

Invariant 27.8.1 (Kernel step-alignment)**.** *A kernel may reorder tiles and memory accesses, but it must preserve the logical attention mask, sequence lengths, and matrix product semantics.*

The same rule covers both sides of the chapter. For attention, KV index generation, page size, split scheduling, semantic modifiers, and output slices must agree with the scheduled rows. For GEMM, tile shape, packing, scale broadcast, padding, architecture dispatch, and output dtype must agree with the requested product. Matching tensor extents is not enough: a wrong KV row, wrong FP8 scale block, stale graph bucket row, missing sink modifier, or invalid sparse compaction can preserve shape while changing the operation.

Launch contracts and graph-wrapped buckets.

The attention-backends chapter chooses an implementation family. The launch contract checks whether the selected family still denotes the scheduled work. The compact checklist is:

Definition 27.8.2 (Graph-wrapped launch bucket) *A graph-wrapped launch bucket is a captured attention or GEMM shape whose stable buffers, fill values, batch bucket, and replay metadata can be reused for later serving steps only when the replayed rows and output slice still denote the current scheduled requests.*

SGLang's attention backend interface separates ordinary forward metadata, CUDA-graph state initialization, capture metadata, replay metadata, and the sequence-length fill value. That separation is a correctness boundary before it is a compilation detail: replay can reuse storage and shapes, but it must not reuse stale request rows, stale page-table entries, or stale output ownership.

Table 27.15.: Launch correctness checklist. The fields that stay aligned determine whether a launch falls back or fails.

Launch surface	Fields that must stay aligned	Fallback or failure boundary
Attention	Query rows, sequence lengths, page tables or `kv_indices`, split buffers, positions, mask semantics, sink/logit-cap modifiers, workspace, and output slice.	Fallback when page layout, sparse selection, modifier support, workspace, or backend mode cannot express the scheduled rows.
Dense GEMM	M, N, K, operand layout, tile assumptions, dtype, architecture path, output stride, and output slice.	Reject or reroute when tensor-core alignment, dtype, stride, or architecture support is missing.
Quantized GEMM	Packed weights, scale tensors, group size, zero-point policy, activation dtype, dequantization location, tolerance, padding, and output dtype.	Reject or reroute when scale grouping does not match the packed values or the selected kernel's layout convention.
Graph-wrapped launch	Stable buffers, bucket size, replay metadata, fill values, live-row count, and request-visible output slice.	Replay only when dynamic metadata still fits the captured bucket; otherwise use an eager or newly captured path.

Example 27.8.1 (Replay bucket with fewer live rows) Suppose a decode graph was captured for a batch bucket of four rows, but the current step has only requests A and B. A legal replay may fill the two unused sequence-length slots, reuse stable KV-index buffers, and run the same captured graph body. It must still rebuild or overwrite the request-visible metadata for A and B, and the returned tensor slice must expose only those two rows. Returning all four bucket rows turns a launch optimization into a request-identity bug.

This example proves that graph replay is a metadata reuse contract. It does not make the captured bucket's padded rows part of the live serving step.

Invariant 27.8.3 (Kernel-metadata accounting)**.** *An attention or GEMM kernel account is valid only when logical arithmetic, metadata construction, semantic modifiers, padding, representation conversion, workspace ownership, output slicing, and numerical tolerance all describe the same scheduled request work.*

That accounting rule is the bridge from correctness to measurement. A latency row can count only after the record names the same logical operation, the same layout view, the same semantic modifiers, and the same tolerance policy. Otherwise a faster launch may only be faster because it computed a different attention distribution or a differently dequantized product.

Shape-compatible bug	Why shapes still pass	Semantic failure
Wrong KV row	`kv_indices` has the expected length.	The query reads another request's cache history.
Wrong FP8 scale block	The scale tensor broadcasts to the tile shape.	The product applies the wrong dequantization ledger.
Stale graph bucket row	The captured output has the right bucket extent.	The caller sees a row that no live request owns.
Missing logit cap or sink	The same query and cache tensors reach both kernels.	The attention distribution is different before value accumulation.
Invalid sparse page-table compaction	The compact destination buffer has the expected shape.	A valid block is dropped, an invalid entry survives, or visible-block order is changed.

Table 27.16.: Shape-compatible semantic bugs. Shapes can still pass while the launch changes the requested semantic operation.

Definition 27.8.4 (Attention sink) *For one query row and one attention head, an attention sink is an optional per-head scalar s_h that enters the softmax normalizer as an extra virtual logit with no value-vector contribution. If the visible token scores are $a_1, \ldots, a_n$, then a sink-aware attention row computes*

$$\alpha_j = \frac{\exp(a_j)}{\sum_{k=1}^{n} \exp(a_k) + \exp(s_h)}, \qquad o = \sum_{j=1}^{n} \alpha_j v_j.$$

The sink therefore absorbs probability mass without adding a KV row or an αv term to the output.

The sink definition is included here because it is easy to mistake it for a layout detail. It is not one. Logit capping transforms pre-softmax scores, XAI-style temperature changes the scale used by the kernel, and an attention sink changes the denominator. A backend comparison that omits one of those fields is comparing different attention functions, even if both paths read the same cache rows.

Example 27.8.2 (Sink mass without a sink value) For one head, suppose two visible token scores are 2 and 0, with values v_1 and v_2, and the sink scalar is $s_h = 1$. Without a sink, the softmax masses are about 0.881 and 0.119. With the sink, the denominator becomes $\exp(2) + \exp(0) + \exp(1)$, so the masses are about 0.665, 0.090, and 0.245 for token 1, token 2, and the sink. The output is $0.665v_1 + 0.090v_2$. The remaining 0.245 is not a hidden KV read; it is probability absorbed by the sink.

The sink example proves that semantic modifiers belong in the launch record. It does not add a new cache page, value vector, or request-owned output row.

Sparse attention makes the last row concrete. A page table row such as $[7, -1, 12]$ should compact to $[7, 12]$ with count two and the valid-block order preserved. Keeping the negative entry, dropping block 12, or reordering the valid blocks changes the visible KV history even when the destination tensor shape is unchanged.

Correctness tests make the accounting meaningful. FlashAttention tests stress cache update, page-table, left-padding, rotary, local-window, and variable-length

cases because each changes metadata while preserving the logical attention expression. FP8 blockwise GEMM tests construct explicit scale shapes and compare against a baseline with fixed tolerances. A faster backend path that corrupts row identity, shape fields, modifiers, scale layout, or output slicing is not an optimization; it is a semantic regression.

27.9. Performance Bounds: From Cost Ledger to Bottlenecks

Attention and GEMM bottlenecks move with batch shape and sequence length.

The central tradeoff is that decode attention often spends much of its time moving and interpreting KV-cache state, while large GEMMs can reach high arithmetic intensity if their layouts match tensor-core kernels. This explains why a serving runtime cannot treat "fast attention" and "fast GEMM" as one generic acceleration problem. Attention backends must optimize page-table construction, split choice, workspace reuse, and cache layout. GEMM kernels must optimize tile shape, architecture dispatch, packed weights, scale broadcast, and alignment. The common interface is the serving kernel shape S_t from Definition 27.2.3; the bottleneck terms are often different.

Chapter 3 supplied the serving-step ledger. $A(\mathcal{R})$ is the attention pair account induced by live requests and visible context, $D(\mathcal{R})$ is the dense-token work account, K_{live} is resident live state, and M_{cap} is resident capacity. At the launch boundary, that ledger supports a performance lower bound only after the backend translates counts into operational quantities. The launch record has to expose bytes, FLOPs, launches, copies, padding, and communication.

The roofline idea is simple: the step must take at least as long as the slower of moving its bytes and doing its math. Low arithmetic intensity from Definition 27.2.4 makes the memory term likely to win; high intensity makes the compute term more likely. A useful envelope is

$$t_{\text{step}} \gtrsim \max\left(\frac{R_{\text{bytes}}}{B_{\text{mem}}^{\text{eff}}}, \frac{W_{\text{flop}}}{F_{\text{comp}}^{\text{eff}}}\right) + n_{\text{launch}}\tau_{\text{launch}}^{\text{eff}} + t_{\text{collective}} + t_{\text{copy+pad}}.$$

Here R_{bytes} is backend-visible memory traffic, and W_{flop} is backend-visible arithmetic. B_{mem} is memory bandwidth in bytes/s, F_{comp} is compute throughput in FLOP/s, and τ_{launch} is per-launch overhead in seconds. The eff superscript means achieved value on this device, backend, graph mode, dtype, layout, and measurement window, not the spec-sheet peak. This is not an exact scheduler simulator. It is a way to ask whether a measured step is plausibly memory-bound, compute-bound, launch-bound, communication-bound, or dominated by graph padding and copies.

Table 27.17 records the ledger that the following discussion uses.

For the running serving shape, one-token decode for A and B is the traffic-heavy side of the account: KV pages, `kv_indices`, split scratch, and output rows dominate while query-side arithmetic is too small to amortize metadata well. The 128×4096 projection for C is the throughput-heavy side: weights, activations,

Chapter 3 account	Launch-level quantity	Hardware term	Why it is not exact
$A(\mathcal{R})$ and visible context	K/V bytes, page tables, split metadata, scratch, and output rows.	$B_{\text{mem}}^{\text{eff}}$	Backend tiling, cache locality, masks, and scratch add or remove traffic.
$D(\mathcal{R})$	GEMM and MLP FLOPs plus activation, weight, scale, and output traffic.	$F_{\text{comp}}^{\text{eff}}$ and $B_{\text{mem}}^{\text{eff}}$	Tensor-core eligibility, dtype, packing, and padding change achieved rates.
K_{live} and request slots	Admission feasibility and resident-object pressure.	M_{cap}	Resident state gates whether a step can run; time depends on bytes actually touched.
Graph buckets and padding	Padded rows, stable-buffer copies, replay hits, launches, and fallbacks.	$\tau_{\text{launch}}^{\text{eff}}$ and copy bandwidth	Graphs reduce launch overhead but may add padding, copies, and miss residuals.
Tensor-parallel boundary	All-reduce payload, rounds, backend, and possible overlap.	$t_{\text{collective}}$	Chapter 23 supplies the α, β account; overlap changes the observed residual.

Table 27.17.: Mapping the Chapter 3 serving ledger to launch-level performance bounds. The hardware terms become useful only after backend metadata gives the traffic, FLOP, launch, graph, and collective quantities operational meaning.

scales, and output writes are large but regular enough for tensor-core tiles to amortize metadata when dtype and layout are legal.

Example 27.9.1 (Decode KV-read traffic ledger) For the running decode half, requests A and B have visible lengths 97 and 2049. With $H_{\text{KV}} = 8$, $d_h = 128$, and a two-byte cache dtype, the K/V read term alone is

$$(97 + 2049) \cdot 8 \cdot 128 \cdot 2 \cdot 2 = 8{,}790{,}016 \text{ bytes} \approx 8.38 \text{ MiB}.$$

The two new query rows and two output rows are tiny by comparison. Split metadata, page tables, and log-sum-exp scratch add more traffic, so the ledger is dominated by reading and interpreting cache history rather than by the arithmetic for the new query rows.

This ledger proves why decode attention often starts from a memory-traffic question. It does not claim a measured latency by itself; launch overhead, split metadata, graph state, query/output traffic, and local memory behavior still have to be measured for a full row.

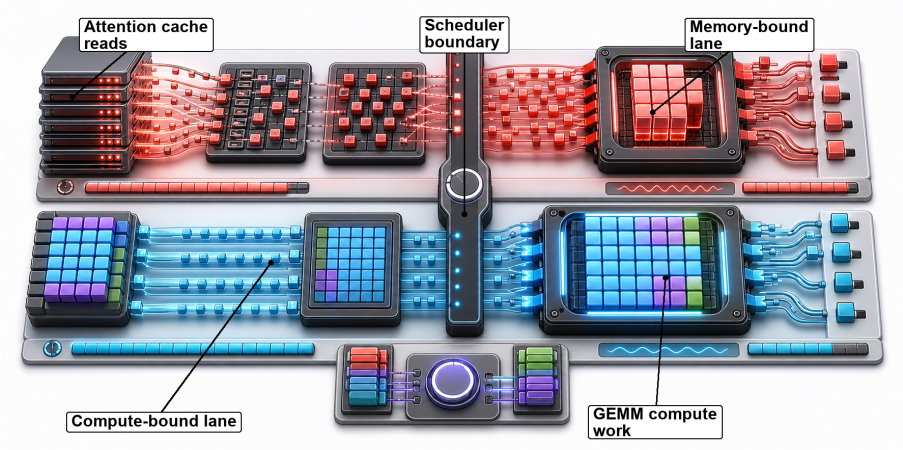

Figure 27.8.: Attention versus GEMM resource pressure. Systems illustration comparing memory-bound attention cache reads with compute-heavy GEMM work, using two adjacent accelerator lanes and a shared scheduling boundary.

Example 27.9.2 (Decode-attention occupancy estimate) Occupancy is not a latency claim; it is a resource budget that explains why a kernel shape may stop improving. For a representative NVIDIA H100 budgeting row, use 65,536 32-bit registers per SM, a 2,048-thread SM limit, and 64 resident warps as the architectural ceiling, then replace those constants with compiler output and device attributes for a measured row. For the running decode shape with $H_Q = 32$, $H_{\mathrm{KV}} = 8$, and $d_h = 128$, suppose a decode CTA has 128 threads, or four warps, and the compiled kernel uses 64 registers per thread for query fragments, page metadata, and the online-softmax state. The register file then admits

$$128 \cdot 64 = 8{,}192$$

registers per CTA, so the register budget allows $\lfloor 65{,}536/8{,}192 \rfloor = 8$ resident CTAs per SM. The thread budget would allow $\lfloor 2{,}048/128 \rfloor = 16$ CTAs, and the warp ceiling would allow $64/4 = 16$ CTAs, so register pressure is the tighter limit: 8 CTAs, or 32 resident warps. If larger head fragments or more split state raise the kernel to 96 registers per thread, the same arithmetic becomes $\lfloor 65{,}536/(128 \cdot 96) \rfloor = 5$ CTAs, or 20 resident warps. The online-softmax state (m, ℓ, u) is logically small, but u is a value-fragment accumulator. Serving more query-head fragments per CTA therefore increases per-thread accumulator state and can bound resident warps even when the request-level attention formula is unchanged.

Figure 27.8 gives the visual version of this split: the same scheduled step can be limited by cache interpretation on the attention side and by tensor-core eligibility on the GEMM side.

Quantized GEMM makes the tradeoff concrete. Lowering operand precision can reduce memory traffic and increase tensor-core throughput, but it introduces extra scale and zero-point metadata and stricter layout contracts. SGLang's FP8 blockwise path requires FP8 E4M3 inputs, scale tensors whose dimensions match 128-element groups, and architecture-specific CUTLASS support; QServe W4A8 paths distinguish per-group and per-channel scale layouts. These kernels can be excellent choices for a compatible model layer and a poor choice for a shape whose scale metadata or compute capability does not match.

Runtime attention integration has a parallel tradeoff. A backend registry can expose specialized attention kernels, device-specific branches, and native Torch fallback paths, but the selected backend still has to implement the metadata lifecycle promised by `AttentionBackend`. The Chapter 29 covers backend-specific contracts. Feature-rich backends may carry more workspace or planning state; simpler fallback paths may be slower but easier to validate. The practical rule is to place checks where metadata is consumed: before a GEMM launch for stride, dtype, alignment, and scale shape; before an attention launch for sequence lengths, page tables, split buffers, and cache locations. Earlier checks can miss scheduler-induced changes, and later checks may run after a kernel has already observed invalid state.

For example, an early validation step might approve a request before overlap scheduling changes the live sequence length or before a CUDA-graph bucket pads the batch. The launch boundary still has to validate the page table, split buffer, and output slice that the kernel will actually consume. Otherwise the kernel can observe stale metadata even though the original logical operation was legal.

27.10. Narrow Kernel Measurements

A useful kernel measurement changes one launch choice while keeping the serving case fixed. The model and request trace stay in place. Hardware, backend-selection rule, and measurement window stay fixed as well. The experiment changes a split policy, kernel family, packing layout, or dtype path. Otherwise a faster row may only be faster because it measured a different serving problem.

Kernel measurements must name the SGLang path that produced each result.

Attention and GEMM need different evidence because they can be wrong in different ways. Attention evidence has to prove that the same rows and visible KV history reached the kernel. It must also preserve masks, page tables, split work, semantic modifiers, and output slots. GEMM evidence has to prove that the same M, N, K problem survived dtype conversion and packed weights. Scale layout, padding, architecture dispatch, and epilogue slicing must match the same claim. In both cases, latency or bandwidth numbers are usable only after a reference check or validated baseline shows that the launch still computes the same operation.

Measurement slice	Evidence that must be fixed	Failure the record prevents
Decode attention	Live sequence lengths, page size, page-table or `kv_indices` shape, split count, selected backend, graph mode, semantic modifiers, and reference agreement.	A speedup that changed visible context, mask behavior, workspace ownership, or output row ownership.
GEMM	M, N, K, dtype and output dtype, packed-weight format, scale layout, padding, architecture path, achieved bandwidth or FLOP/s, and tolerance.	A speedup that used a different matrix, wrong scale block, invalid padding, or different epilogue slice.

Table 27.18.: Measurement-slice evidence requirements. Each slice fixes evidence that prevents an invalid kernel speed claim.

Table 27.19.: Ready-kernel measurements. The attention rows are CUDA-synchronized single-GPU measurements across anonymized hosts in the same NVIDIA device class; both shapes matched an eager attention reference at 2×10^{-2} tolerance. The eager reference validates semantics but is not a paired latency baseline; the following paragraph gives only a KV-read lower-bound frame.

Operator	Shape	Scope	Max diff	Mean ms
Decode attention	$\mathcal{B} = 1,\ L = 128$ $H_Q = 8,\ H_{\mathrm{KV}} = 8$ $d_h = 64,\ \text{splits} = 4$	Thor slice	3.7×10^{-4}	0.0784
Decode attention	$\mathcal{B} = 4,\ L = 512$ $H_Q = 16,\ H_{\mathrm{KV}} = 8$ $d_h = 128,\ \text{splits} = 8$	Thor slice	4.9×10^{-4}	0.1307

Table 27.20.: RTX 5090 attention and GEMM measurements. The FP8 rows use E4M3 inputs, BF16 output, per-token A scales, and 128-by-128 B scale blocks. Decode attention uses page size 1 and 2×10^{-2} relative and absolute tolerance.

Operator	Shape	Latency/rate	Agreement
FP8 blockwise GEMM	$(M, N, K) = (1, 128, 512)$	0.0220 ms; 0.0060 TFLOP/s	max abs 0, max rel 0.
FP8 blockwise GEMM	$(M, N, K) = (64, 512, 512)$	0.0063 ms; 5.3239 TFLOP/s	max abs 1.91×10^{-6}, max rel 0.0063.
FP8 blockwise GEMM	$(M, N, K) = (128, 1024, 1024)$	0.0150 ms; 17.9313 TFLOP/s	max abs 0.03125, max rel 0.0076.
Decode attention	$\mathcal{B} = 1,\ L = 128$ $H_Q = 8,\ H_{\mathrm{KV}} = 8$ $d_h = 64,\ \text{splits} = 4$	0.0209 ms	max abs 4.88×10^{-4}, max rel 0.2316.
Decode attention	$\mathcal{B} = 4,\ L = 512$ $H_Q = 16,\ H_{\mathrm{KV}} = 8$ $d_h = 128,\ \text{splits} = 8$	0.0222 ms	max abs 1.22×10^{-4}, max rel 0.9831.

The recorded SGLang codebase contains decode-attention and FP8 block-GEMM benchmark/test surfaces. The rows below are narrow measurements, not a complete backend sweep. They are included because they produced reviewed correctness and latency evidence under fixed shapes.

A narrower ready-kernel measurement then exercised the one-GPU paths that could run immediately on the same device class. Table 27.19 records the measured attention rows for the reviewed single-GPU device-class slice.

The 2×10^{-2} attention tolerance is a ceiling, not the achieved error. Attention kernels can legitimately accumulate more floating-point drift than a single GEMM because online-softmax reductions, exponentials, rescaling, split reductions, and accumulation order differ by backend. In these two rows, however, the observed maximum differences, 3.7×10^{-4} and 4.9×10^{-4}, are both below 10^{-3}.

The RTX 5090 follow-up adds FP8 blockwise GEMM rows and repeats the two decode-attention shapes on a Blackwell desktop GPU. Table 27.20 keeps the measured rows separate from the Thor slice because the hardware, SGLang kernel package, and CUDA stack differ.

Figure 27.9 gives a visual read of the same completed rows. It separates latency from throughput so the plot shows only within-metric structure: same-device-class spread for decode attention and the RTX 5090 FP8 GEMM size ramp.

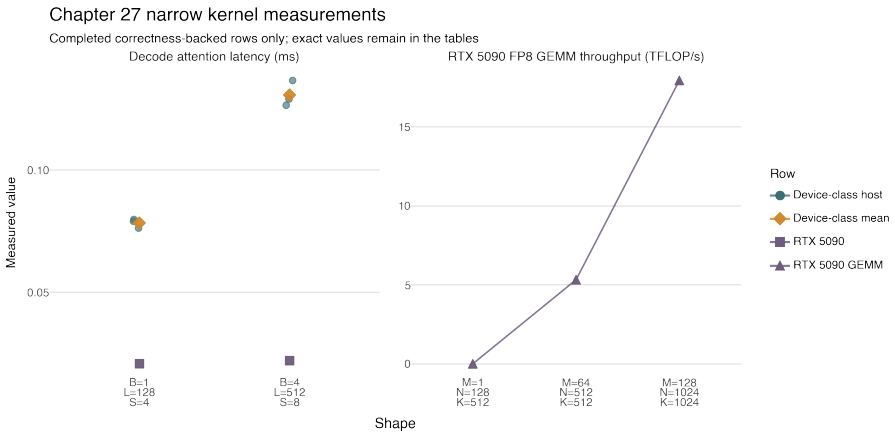

Figure 27.9.: Narrow attention/GEMM measurement view. The left panel plots the two decode-attention shapes from Tables 27.19 and 27.20 with individual rows from the same anonymized device class, their mean, and the RTX 5090 row. The right panel plots only the RTX 5090 FP8 blockwise-GEMM rows. The figure visualizes completed correctness-backed rows; it does not claim a backend sweep or a general cross-hardware speedup.

Table 27.21.: Scope of narrow-kernel evidence rows. The table separates what an evidence row proves from what it cannot claim.

Evidence row	What it proves	What it does not claim
Two Triton decode-attention shapes	Semantic agreement with the eager reference plus latency on the Thor and RTX 5090 slices.	General attention-backend speedup, prefill behavior, or all sequence shapes.
RTX 5090 FP8 GEMM rows	Scaled-reference agreement, latency, and FLOP/s for three blockwise FP8 shapes.	General FP8 GEMM performance over all shapes, packing layouts, or hardware targets.

The large relative-error entries in the decode-attention rows are not evidence that the kernels failed the agreement check. Relative error explodes when the reference value is near zero; the same rows have maximum absolute error around 10^{-4}, well inside the 2×10^{-2} relative-or-absolute tolerance used for the comparison. The agreement claim therefore rests on the paired absolute and relative tolerance, not on relative error alone.

A latency row also needs a lower-bound frame. For the first measured shape, the KV-read term alone is $1 \cdot 128 \cdot 8 \cdot 64 \cdot 2 \cdot 2 = 262{,}144$ bytes, or 0.25 MiB. For the second, it is $4 \cdot 512 \cdot 8 \cdot 128 \cdot 2 \cdot 2 = 8{,}388{,}608$ bytes, or 8 MiB. Those byte counts are not a hardware-speed claim by themselves. A calibrated bound would also need the local memory path, launch overhead, metadata and split handling, query/output traffic, online-softmax work, graph state, and measurement-window costs.

These measurements provide correctness and latency evidence for the Triton decode-attention path on Thor and RTX 5090, plus a narrow RTX 5090 FP8 blockwise-GEMM row. The same Thor run records FP8 blockwise GEMM as a blocked hardware/software boundary for compute capability 11.0, not as a failed correctness comparison. They still do not make a full attention/GEMM sweep claim: the result set would need broader shape, backend, tolerance, bandwidth, and FLOP/s columns before supporting a general cross-backend performance statement.

The running example changes one kernel-facing choice at a time: an attention split policy, a CUTLASS MLA path versus another validated backend, or a quantized GEMM packing layout. The workload keeps request mix, prompt-length distribution, scale calibration, and tolerance policy fixed. A result supports the kernel claim only when it reports both speed movement and the launch

record that made the path legal; a faster path that changes masks, scale blocks, or output slicing is a correctness failure even when the kernel metric improves.

27.11. Source Coverage for Attention and GEMM Kernels

The source-coverage material here keeps the chapter boundary explicit: this chapter explains attention and dense GEMM launch contracts, while quantization, MoE, fused operators, KV movement, and collectives keep their specialized coverage in the chapters that own those runtime surfaces.

Kernel reference tables.

The tables in this section are lookup material. They keep source-facing field groups attached to the contracts developed earlier, but they are not the narrative path through the measurement section.

Attention Launch-family Concept-to-code Map.

The concept-to-code map below is lookup material for the attention launch-family contracts in Section 27.5. It is collected here so that the narrative section can stay focused on row ownership, visibility, and metadata obligations.

Concept-to-code map

- **Triton decode metadata.** `python/sglang/srt/layers/attention/triton_backend.py`, `python/sglang/srt/layers/attention/utils.py`, and `python/sglang/srt/layers/attention/triton_ops/decode_attention.py` build and consume `kv_indptr`, `kv_indices`, `num_kv_splits`, `attn_logits`, and `attn_lse`.
- **FA3 extension registration.** `sgl-kernel/csrc/flash_extension.cc` registers `sgl_kernel.fwd` and `get_scheduler_metadata`, then binds `sgl_kernel.fwd` to `mha_fwd` for FA3-style launches.
- **FA3 scheduler metadata.** `python/sglang/srt/layers/attention/flashattention_backend.py` asks for FA3 scheduler metadata when the backend can use it.
- **FA3 Python and C++ API.** `sgl-kernel/python/sgl_kernel/flash_attn.py` and `sgl-kernel/include/sgl_flash_kernel_ops.h` expose FA3 forward and scheduler-metadata calls.
- **FlashMLA extension registration.** `sgl-kernel/csrc/flashmla_extension.cc` registers `get_mla_decoding_metadata`, `fwd_kvcache_mla`, `fwd_kvcache_mla_fp8`, `dense_prefill_fwd`, and `sparse_prefill_fwd`.
- **FlashMLA Python and backend metadata.** `sgl-kernel/python/sgl_kernel/flash_mla.py` and `python/sglang/srt/layers/attention/flashmla_backend.py` cover FlashMLA decode, sparse prefill, and dense prefill paths.
- **CUTLASS MLA decode metadata.** `python/sglang/srt/layers/attention/cutlass_mla_backend.py`,

`sgl-kernel/python/sgl_kernel/attention.py`, and `sgl-kernel/csrc/attention/cutlass_mla_kernel.cu` build the `block_kv_indices` page table, allocate CUTLASS MLA workspace, and implement the path behind `cutlass_mla_decode`.

- **MLA layout helper kernels.** `sgl-kernel/csrc/elementwise/concat_mla.cu` and `python/sglang/jit_kernel/concat_mla.py` cover MLA K packing and absorbed-query concatenation.

FA3 forward schema groups.

FlashMLA decode schema groups.

Table 27.24 summarizes the concrete rows used by the following discussion.

Kernel-surface boundary map

- **Attention metadata and launch entities.** `ForwardBatch`, `kv_indices`, `sgl_kernel.fwd`, `fwd_kvcache_mla`, `fwd_kvcache_mla_fp8`, `dense_prefill_fwd`, `sparse_prefill_fwd`, `cutlass_mla_decode`, `store_cache`, and `transfer_kv_*`.
- **Quantized GEMM entities.** `fp8_scaled_mm`, `fp8_blockwise_scaled_mm`, `int8_scaled_mm`, and `gptq_gemm`.
- **MoE and routing entities.** `fp8_blockwise_scaled_grouped_mm`, `cutlass_w4a8_moe_mm`, `topk_softmax`, `topk_sigmoid`, `moe_fused_gate`, `prepare_moe_input`, `moe_align_block_size`, `apply_shuffle_mul_sum`, and `moe_sum_reduce`.
- **Fused mutation entities.** `silu_and_mul`, `gelu_and_mul`, `gelu_tanh_and_mul`, `fused_add_rmsnorm`, `rotary_embedding`, and `fused_qk_norm_rope`.

This map keeps the chapter boundary explicit without becoming a book-wide custom-operator concordance. SGLang kernel history is clearest from the runtime shape backward. Ask which serving step produced the tensors, which metadata describes them, which backend or architecture gate selected the kernel, and which benchmark shape proves that the path is worth keeping.

Table 27.22.: FA3 forward schema groups. Representative fields show why each schema group matters for source coverage.

Schema group	Representative fields	Why it matters
Attention payload	`q`, `k`, `v`, optional `k_new`, `v_new`, `q_v`, and optional mutable `out`.	The operator receives the query rows, the readable K/V state, optional newly computed K/V rows, and the output buffer that should remain aligned with query segments.
Segment and length metadata	`cu_seqlens_q`, `cu_seqlens_k`, `cu_seqlens_k_new`, `seqused_q`, `seqused_k`, `max_seqlen_q`, and `max_seqlen_k`.	Packed rows are legal only because prefix-sum boundaries and maximum spans say where each request's query and key segment begins and ends.
Paged-cache placement	`page_table`, `kv_batch_idx`, and `leftpad_k`.	The same logical key segment may live in paged cache storage; these fields map the segment contract onto physical cache rows or pages.
Model and mask modifiers	`rotary_cos`, `rotary_sin`, `seqlens_rotary`, `q_descale`, `k_descale`, `v_descale`, `softmax_scale`, `is_causal`, window bounds, `attention_chunk`, `softcap`, and `sinks`.	Rotary state, FP8 scale state, causal or local visibility, softcapping, chunking, and sink tokens are semantic modifiers. They are part of the attention function, not optional profiling decorations.
Tile scheduling and launch control	`scheduler_metadata`, `num_splits`, `pack_gqa`, and `sm_margin`.	Scheduler metadata and split policy describe how the legal rows are divided into FA3 work. They may change tile order and occupancy pressure, but not request ownership or visibility.
Return surface	`out`, `softmax_lse`, and auxiliary return tensors.	The first return aliases a supplied output buffer when one is provided; `softmax_lse` records the row log-sum-exp needed by composed attention paths, tests, or later reductions.

Schema group	Representative fields	Why it matters
MLA payload	`q`, `kv_cache` or `kcache`, and `head_size_v`.	The query rows attend over latent MLA cache pages rather than expanded ordinary K/V rows; the value-head dimension fixes the output width.
Request-history placement	`seqlens_k` and `block_table` or backend-built `block_kv_indices`.	Each batch row names the visible latent pages for one request history; unused block-table entries are padding, not extra visible context.
Tile scheduling and splits	`tile_scheduler_metadata` and `num_splits`, returned by `get_mla_metadata` or the dense-FP8 metadata helper.	The metadata partitions the legal latent-cache read into FlashMLA work units without changing request ownership or causal visibility.
Attention modifiers and variants	`softmax_scale`, `is_causal`, `is_fp8`, and optional `indices`.	Scale, causal masking, FP8 cache interpretation, and sparse neighborhood selection are semantic inputs to the attention function, not profiling labels.
Dense FP8 scale state	`descale_q` and `descale_k` in `fwd_kvcache_mla_fp8`.	The FP8 entry point receives descaling tensors instead of the generic `is_fp8/indices` switches; the wrapper requires query and key scale state to be provided or omitted together.
Return surface	`out` and `softmax_lse`.	The first return is the batch-aligned attention output; `softmax_lse` records row log-sum-exp state for testing and composed attention paths.

Table 27.23.: FlashMLA decode schema groups. Representative fields show why each schema group matters for source coverage.

Table 27.24.: Implementation bridge for the dense scaled-GEMM operators. The table names the source-reading path that turns a Python-visible `sgl_kernel` function into a checked CUDA launch and a reference-tested operator contract.

Operator	CUDA anchor	Entry checks	Dispatch and output rule	Test oracle
Rowwise FP8	`fp8_gemm_kernel.cu`	Row-major FP8 E4M3 A, column-major FP8 E4M3 B, FP32 scale vectors of lengths M and N, 16-byte alignment, optional bias matching the output dtype.	SM89, SM90, SM100, or SM120 dispatch helpers choose output dtype, bias, and shape-bucket template variants; the result is written as the requested $M \times N$ tensor.	Row ref.
Blockwise FP8	`fp8_blockwise_gemm_kernel.cu`	The same FP8 layout and alignment checks, plus activation scale shape $(M, K/128)$, weight scale shape $(K/128, N/128)$, and scale strides matching the blockwise address convention.	Pads activation rows and scale rows, dispatches SM90, SM100, or SM120 blockwise CUTLASS paths, then slices the padded output back to the original M rows.	Block ref.
INT8	`int8_gemm_kernel.cu`	Row-major INT8 A, column-major INT8 B, K-dimension tensor-core alignment, N-dimension output-stride alignment, FP32 row and column scale vectors, optional bias matching the output dtype.	SM75, SM80, SM86, SM89, and SM90 CUTLASS paths choose output dtype and tile families; unsupported compute capabilities fail instead of silently changing semantics.	INT8 ref.

27.12. Summary

Attention and GEMM kernel choices are serving decisions, not isolated microbenchmarks. A faster launch is useful only when it still computes the same scheduled operation: the same request rows, visible KV history, masks, scale records, padding rules, workspace ownership, and output slices. In the running example, the decode rows for A and B may stream KV pages in the order the selected attention backend needs, but their outputs still belong to their own visible histories. The prefill suffix C may pass through a packed or quantized projection GEMM, but the returned rows still denote the scheduled suffix rather than padding or an altered matrix product.

For attention, the central risk is that a legal-looking launch shape reads the wrong cache history or writes the result to the wrong row. For GEMM, the central risk is that packing, quantization scales, padding, or the epilogue changes the matrix product the model requested. Layout metadata therefore acts as semantic state: page tables, index lists, tile shapes, scale blocks, and output slices are the evidence that physical bytes still denote the logical operator.

The measurement rows cover specific decode-attention and FP8 GEMM shapes. They report correctness-backed latency or throughput only after the step-alignment and metadata-accounting checks in Invariants 27.8.1 and 27.8.3 hold; broader backend and hardware claims stay out of scope. That is the durable review rule for this chapter: optimize the launch, but prove that the optimized path preserved row ownership, cache visibility, scale interpretation, and the serving-step boundary.

27.13. Exercises

1. Using Definition 27.2.3, define serving kernel shape for a decode attention step and for a prefill GEMM step.
2. Using I = FLOPs/bytes, explain why one-token decode attention can be memory-heavy while a large projection GEMM can be compute-dense. Then use the decode KV-read traffic ledger in Section 27.9 to compute a memory lower bound with a chosen $B_{\text{mem}}^{\text{eff}}$. Compare it with either a local decode-attention measurement or one row from Table 27.19 after recomputing that row's traffic from its shape, and classify the residual as launch, metadata, padding, non-KV traffic, communication, or measurement-window overhead.
3. Trace ForwardBatch sequence lengths, cache locations, and request-pool indices into Triton attention metadata. For two decode requests with histories 96 and 2048 and page size 16, construct `seq_lens`, `kv_indptr`, page counts, and the output-row mapping; then name one metadata bug that preserves tensor shape while changing the visible K/V history.
4. Compare FA3 scheduler metadata, FlashMLA block-table metadata, and a page-table CUTLASS MLA decode path.
5. List the dtype, stride, alignment, scale-shape, and architecture checks required before an FP8 blockwise GEMM launch.
6. Explain how QServe W4A8 packed weights and scale layouts change metadata without changing the logical matrix product.

7. Describe a padding or output-slicing bug that preserves tensor shape while exposing non-request rows.
8. State the boundary between this kernel chapter, the backend-selection chapter, the quantization chapter, and the CUDA-graph chapter.
9. Design an attention split-policy or GEMM packing experiment and name the metadata, semantic modifier, speed, bandwidth, and correctness fields to record.
10. Review a proposed kernel. List the attention-side checks it must preserve: serving shape, mask, sequence state, cache location, page table or KV-index, split count, workspace, semantic modifiers, and scheduler metadata. List the GEMM-side checks it must preserve: M, N, K, dtype, scale layout, packed weights, padding, architecture dispatch, tolerance, fallback, and output slicing. Classify the relevant `torch.ops.sgl_kernel` family as attention, GEMM, MoE routing, MoE execution, normalization, positional mutation, communication, cache movement, constrained sampling, or speculative bookkeeping; then state one shape-preserving failure the review must catch. Use that classification to scope the kernel review, and name the kernel test or measurement artifact that would make the failure visible. Record the failing tensor or request-row name.

CUDA Graphs, Piecewise Graphs, and Compilation 28.

Chapter Abstract. A decode step with 37 live request rows can reuse a graph captured for 40 rows only if the runtime keeps the three padded rows invisible. Before replay, the runtime refreshes input ids and positions. It also updates cache locations, output buffers, and attention metadata for the live batch. The returned tensors are sliced back to the scheduler's rows. CUDA graphs and compilation are valuable because they reduce host launch and framework dispatch overhead for repeated serving steps, but they do not make the model arithmetic cheaper. This chapter explains how SGLang makes replay a serving decision rather than a blind launch shortcut. The reusable shape is only the start; the runtime must prove graph-hit eligibility, keep replay buffers aligned with live rows, and fall back when a step no longer fits the captured path.

28.1. Graph Replay as Shape-Stable Serving State

Graph replay is useful only when the runtime can prove that a live serving step still means the same shape-stable state as a previously captured executable skeleton. CUDA graphs reduce host launch overhead. The serving problem is to keep request rows and cache writes aligned with metadata and output slices while the replay machinery sees fixed tensor extents and stable addresses.

Graph replay reduces launch overhead only when shapes stay compatible.

The running example is a decode-mode forward batch from the ModelRunner chapter in Volume I. It serves a `Qwen/Qwen2.5-Coder-7B-Instruct` decode trace with 37 live rows. The runtime can replay a graph captured for a padded bucket of 40 only if the bucket still describes the same live rows. The three padded rows are shape support, not new requests. A mixed prefill step may fall back to the ordinary path because its token count, input embeddings, log-probability requirements, or hidden-state mode do not match the captured path. Figure 28.1 shows that routing problem: compatible decode steps enter a shape bucket and replay, while incompatible shapes or metadata take the fallback path.

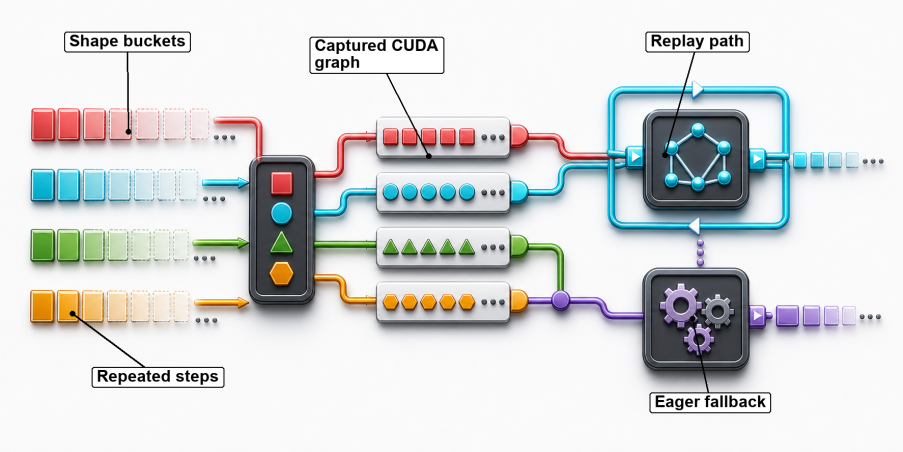

Figure 28.1.: Capture bucket and graph replay. Repeated model steps entering shape buckets, with compatible steps replaying a captured CUDA graph and incompatible steps falling back to eager execution.