FlashMLA

Deep Dive into DeepSeek — Chapter 22

Yin Yang

Foundation Books Project

The one rule of this chapter

A faster path is interpretable only when it returns the declared output and log-sum-exp within tolerance.

- MLA reduces decode-time cache traffic. A serving kernel still has to map each logical position to physical cache bytes and preserve the attention result.
- FlashMLA has four distinct attention paths: dense and sparse FP8 MLA decode, sparse MLA prefill, and dense MHA prefill.
- Cache addressing, FP8 representation, split reduction, seesaw scheduling, and Hopper overlap all change *how* that result is produced — never *what* it must equal.

The one rule of this chapter

A faster path is interpretable only when it returns the declared output and log-sum-exp within tolerance.

- MLA reduces decode-time cache traffic. A serving kernel still has to map each logical position to physical cache bytes and preserve the attention result.
- FlashMLA has four distinct attention paths: dense and sparse FP8 MLA decode, sparse MLA prefill, and dense MHA prefill.
- Cache addressing, FP8 representation, split reduction, seesaw scheduling, and Hopper overlap all change *how* that result is produced — never *what* it must equal.

Goal: follow one paged-cache decode call through addressing, layout, reduction, and scheduling — and see where every piece lives in the pinned FlashMLA source.

Roadmap

1. Why MLA needs a kernel, not just a formula

Roadmap

1. Why MLA needs a kernel, not just a formula
2. FP8 layout and Hopper crossover

Roadmap

1. Why MLA needs a kernel, not just a formula
2. FP8 layout and Hopper crossover
3. Split-KV reduction and seesaw scheduling

Roadmap

1. Why MLA needs a kernel, not just a formula
2. FP8 layout and Hopper crossover
3. Split-KV reduction and seesaw scheduling
4. Repository surfaces and the decode/prefill matrix

Roadmap

1. Why MLA needs a kernel, not just a formula
2. FP8 layout and Hopper crossover
3. Split-KV reduction and seesaw scheduling
4. Repository surfaces and the decode/prefill matrix
5. Correctness, benchmarking, and who owns what

Why MLA needs a kernel, not just a formula

The one equation, the seven-part contract

$$\text{Attention}(q, K, V) = \text{softmax}\big(\alpha q K^\top\big)\, V, \qquad \text{plus a returned log-sum-exp } \ell.$$

- An attention kernel is a compiled implementation of this map with fixed shape, layout, masking, address metadata, dtype, scales, and reduction semantics.

The one equation, the seven-part contract

$$\text{Attention}(q, K, V) = \text{softmax}\big(\alpha q K^\top\big)\, V, \qquad \text{plus a returned log-sum-exp } \ell.$$

- An attention kernel is a compiled implementation of this map with fixed shape, layout, masking, address metadata, dtype, scales, and reduction semantics.
- Absorbed MLA has many query heads read one latent-plus-RoPE row — compression can raise arithmetic intensity as fast as it lowers bytes.

The one equation, the seven-part contract

$$\text{Attention}(q, K, V) = \text{softmax}\big(\alpha q K^{\top}\big)\, V, \qquad \text{plus a returned log-sum-exp } \ell.$$

- An attention kernel is a compiled implementation of this map with fixed shape, layout, masking, address metadata, dtype, scales, and reduction semantics.
- Absorbed MLA has many query heads read one latent-plus-RoPE row — compression can raise arithmetic intensity as fast as it lowers bytes.

At h_q=128, s_q=1, d_k=576, d_v=512: about 241.8 FLOPs/byte [DERIVED: FlashMLA §1 model]; the recorded discussion rounds this to about 256 [REPORTED: FlashMLA dense-decode deep-dive].

The one equation, the seven-part contract

$$\text{Attention}(q, K, V) = \text{softmax}\big(\alpha q K^{\top}\big)\, V, \qquad \text{plus a returned log-sum-exp } \ell.$$

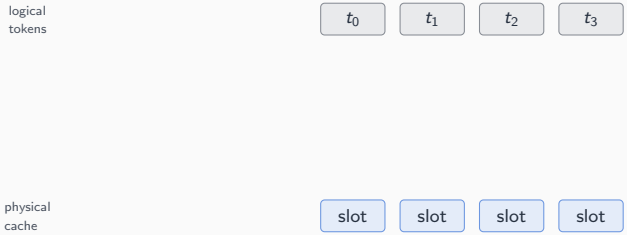
- An attention kernel is a compiled implementation of this map with fixed shape, layout, masking, address metadata, dtype, scales, and reduction semantics.
- Absorbed MLA has many query heads read one latent-plus-RoPE row — compression can raise arithmetic intensity as fast as it lowers bytes.

At h_q=128, s_q=1, d_k=576, d_v=512: about 241.8 FLOPs/byte [DERIVED: FlashMLA §1 model]; the recorded discussion rounds this to about 256 [REPORTED: FlashMLA dense-decode deep-dive].

Neither number classifies a call — that needs the measured GPU ridge and clocks.

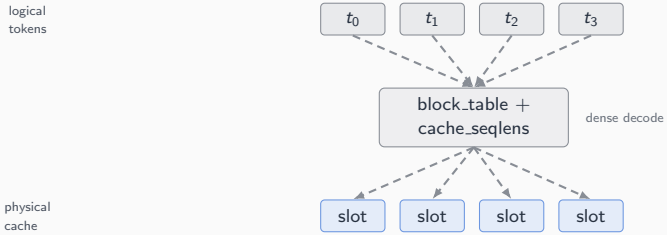
Two coordinate systems, one block table

Token $t \to$ logical block $\lfloor t/P \rfloor$, offset $t \bmod P \to$ physical slot $b_{\text{phys}} P + (t \bmod P)$.



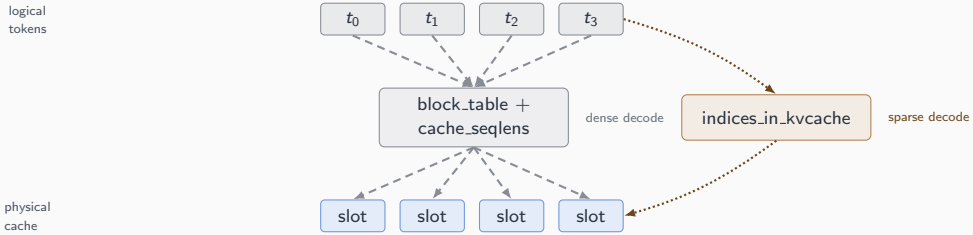
Two coordinate systems, one block table

Token $t \to$ logical block $\lfloor t/P \rfloor$, offset $t \bmod P \to$ physical slot $b_{\text{phys}} P + (t \bmod P)$.



Two coordinate systems, one block table

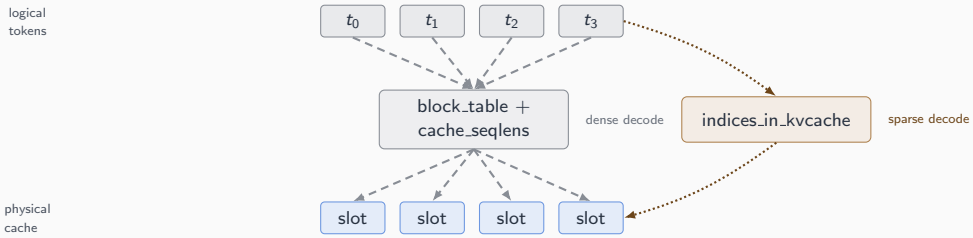
Token $t \to$ logical block $\lfloor t/P \rfloor$, offset $t \bmod P \to$ physical slot $b_{\text{phys}} P + (t \bmod P)$.



Sparse decode already receives encoded physical entries — invalid entries are -1, masked to $-\infty$ before reduction, never simply skipped.

Two coordinate systems, one block table

Token $t \to$ logical block $\lfloor t/P \rfloor$, offset $t \bmod P \to$ physical slot $b_{\text{phys}} P + (t \bmod P)$.



Sparse decode already receives encoded physical entries — invalid entries are -1, masked to $-\infty$ before reduction, never simply skipped.

On SM100, every byte of a sparse cache tensor's address range must be valid — a slice of one allocation qualifies; disjoint pages do not, even with an identical logical page table.

FP8 layout and Hopper crossover

The 656-byte ledger, and why it is mixed

Part	Bytes	
non-RoPE FP8	512	quantized, local scales

[INSPECTED: FlashMLA sparse FP8 layout]

The 656-byte ledger, and why it is mixed

Part	Bytes	
non-RoPE FP8	512	quantized, local scales
scales	16	4 FP32 values

[INSPECTED: FlashMLA sparse FP8 layout]

The 656-byte ledger, and why it is mixed

Part	Bytes	
non-RoPE FP8	512	quantized, local scales
scales	16	4 FP32 values
RoPE BF16	128	never quantized

[INSPECTED: FlashMLA sparse FP8 layout]

The 656-byte ledger, and why it is mixed

Part	Bytes	
non-RoPE FP8	512	quantized, local scales
scales	16	4 FP32 values
RoPE BF16	128	never quantized
Total	**656**	per cached token

[INSPECTED: FlashMLA sparse FP8 layout]

The 656-byte ledger, and why it is mixed

Part	Bytes	
non-RoPE FP8	512	quantized, local scales
scales	16	4 FP32 values
RoPE BF16	128	never quantized
Total	**656**	per cached token

[INSPECTED: FlashMLA sparse FP8 layout]

"FP8 KV cache" hides three separately accounted objects. Hopper reconstructs BF16 operands from this layout before any matrix instruction runs — the cache format is FP8; the MMA input format is BF16.

Crossover shares the dequantization work

- Two CTAs split a 64-position block: each owns 32 rows, converts its full 512-coordinate payload, and publishes to its partner via distributed shared memory.

Crossover shares the dequantization work

- Two CTAs split a 64-position block: each owns 32 rows, converts its full 512-coordinate payload, and publishes to its partner via distributed shared memory.
- Without sharing: $2 \cdot 64 \cdot 512 = 65{,}536$ conversions. With crossover: $32 \cdot 512 + 32 \cdot 512 = 32{,}768$.

Crossover shares the dequantization work

- Two CTAs split a 64-position block: each owns 32 rows, converts its full 512-coordinate payload, and publishes to its partner via distributed shared memory.
- Without sharing: $2 \cdot 64 \cdot 512 = 65{,}536$ conversions. With crossover: $32 \cdot 512 + 32 \cdot 512 = 32{,}768$.

Both CTAs still attend to the identical caller-selected token set after the barrier — crossover changes conversion work and movement, never attention semantics. The pinned source reports 410 versus 250 TFLOP/s for one specified shape. [REPORTED: FlashMLA sparse-FP8 deep-dive, H800-specific]

Split-KV reduction and seesaw scheduling

One shared normalizer across every split

A split-KV schedule partitions cache rows into ranges, each producing a partial (o_s, ℓ_s). The stable merge:

$$m = \max_s \ell_s, \qquad \ell = m + \log \sum_s e^{\ell_s - m}, \qquad o = \sum_s e^{\ell_s - \ell} o_s.$$

One shared normalizer across every split

A split-KV schedule partitions cache rows into ranges, each producing a partial (o_s, ℓ_s). The stable merge:

$$m = \max_s \ell_s, \qquad \ell = m + \log \sum_s e^{\ell_s - m}, \qquad o = \sum_s e^{\ell_s - \ell} o_s.$$

Adding partial outputs without this rescale implements a different operator — the result is independent of how the kernel partitioned the work only because every split shares the same eventual global normalizer.

Seesaw shares output storage, not the softmax state

- One 64×512 FP32 output already holds 32,768 registers; two full copies would consume the Hopper SM's entire 65,536-register file.

Seesaw shares output storage, not the softmax state

- One 64×512 FP32 output already holds 32,768 registers; two full copies would consume the Hopper SM's entire 65,536-register file.
- Seesaw splits the output into $O_L, O_R \in \mathbb{R}^{64\times256}$, one per warpgroup, and reverses which warpgroup does CUDA-core softmax work versus Tensor Core matrix work.

Seesaw shares output storage, not the softmax state

- One 64×512 FP32 output already holds 32,768 registers; two full copies would consume the Hopper SM's entire 65,536-register file.
- Seesaw splits the output into $O_L, O_R \in \mathbb{R}^{64\times256}$, one per warpgroup, and reverses which warpgroup does CUDA-core softmax work versus Tensor Core matrix work.

Output ownership never tips. Both halves share one online-softmax maximum and normalizer — their concatenation is one attention output, not two independently normalized ones.

Repository surfaces and the decode/prefill matrix

From concept to code: the whole operator table

from `FlashMLA/csrc/api/api.cpp`

```
m.def("sparse_decode_fwd", &sparse_attn_decode_interface);
m.def("dense_decode_fwd", &dense_attn_decode_interface);
m.def("sparse_prefill_fwd", &sparse_attn_prefill_interface);
m.def("dense_prefill_fwd", &FMHACutlassSM100FwdRun);
m.def("dense_prefill_bwd", &FMHACutlassSM100BwdRun);
```

From concept to code: the whole operator table

from `FlashMLA/csrc/api/api.cpp`

```
m.def("sparse_decode_fwd", &sparse_attn_decode_interface);
m.def("dense_decode_fwd", &dense_attn_decode_interface);
m.def("sparse_prefill_fwd", &sparse_attn_prefill_interface);
m.def("dense_prefill_fwd", &FMHACutlassSM100FwdRun);
m.def("dense_prefill_bwd", &FMHACutlassSM100BwdRun);
```

Five pybind exports, four lifecycle surfaces — this is the entire Python-visible operator boundary of the package. [INSPECTED: FlashMLA `csrc/api/api.cpp`]

Dense and sparse decode contracts, side by side

	Dense decode	Sparse decode
Address metadata	`block_table`, `cache_seqlens`	`indices_in_kvcache`

Dense and sparse decode contracts, side by side

	Dense decode	Sparse decode
Address metadata	`block_table`, `cache_seqlens`	`indices_in_kvcache`
Mode requirement	Sparse-only args absent	`causal=False`, `is_fp8_kvcache=True`

Dense and sparse decode contracts, side by side

	Dense decode	Sparse decode
Address metadata	`block_table`, `cache_seqlens`	`indices_in_kvcache`
Mode requirement	Sparse-only args absent	`causal=False`, `is_fp8_kvcache=True`
Constraints	SM90a; BF16/FP16; $P{=}64$	SM90a/SM100f; BF16 query; $H_{\mathrm{KV}}{=}1$

Dense and sparse decode contracts, side by side

	Dense decode	Sparse decode
Address metadata	`block_table`, `cache_seqlens`	`indices_in_kvcache`
Mode requirement	Sparse-only args absent	`causal=False`, `is_fp8_kvcache=True`
Constraints	SM90a; BF16/FP16; P=64	SM90a/SM100f; BF16 query; H_{KV}=1

Same return shapes on both sides, o and ℓ — only address metadata and mode requirement differ. Neither mode name is a latency result until shape, dtype, layout, hardware, and a correctness gate are all recorded together.

Two prefill surfaces, two different savings

- DSA owns the learned choice of the selected set S_i.

Two prefill surfaces, two different savings

- DSA owns the learned choice of the selected set S_i.
- Sparse MLA prefill (`flash_mla_sparse_fwd`) executes attention over that caller-selected set.

Two prefill surfaces, two different savings

- DSA owns the learned choice of the selected set S_i.
- Sparse MLA prefill (`flash_mla_sparse_fwd`) executes attention over that caller-selected set.
- Dense MHA prefill (variable-length wrappers) evaluates every position in the declared domain D_i, tiled and fused, never selecting fewer positions.

Two prefill surfaces, two different savings

- DSA owns the learned choice of the selected set S_i.
- Sparse MLA prefill (`flash_mla_sparse_fwd`) executes attention over that caller-selected set.
- Dense MHA prefill (variable-length wrappers) evaluates every position in the declared domain D_i, tiled and fused, never selecting fewer positions.

A shared word like "attention" never tells you which of the two you are calling — the two-versus-four score count in this chapter's worked example does not predict a matching latency ratio either.

SM100: `gather4` and lazy-anchor softmax

- A TMA `gather4` issue names four noncontiguous selected rows at once, split across a two-CTA cooperating pair.

SM100: `gather4` and lazy-anchor softmax

- A TMA `gather4` issue names four noncontiguous selected rows at once, split across a two-CTA cooperating pair.
- Moving a softmax anchor here means touching a 512-coordinate accumulator in tensor memory — expensive, so the pinned source lets the anchor *lag* the true maximum within six base-2 units.

SM100: `gather4` and lazy-anchor softmax

- A TMA `gather4` issue names four noncontiguous selected rows at once, split across a two-CTA cooperating pair.
- Moving a softmax anchor here means touching a 512-coordinate accumulator in tensor memory — expensive, so the pinned source lets the anchor *lag* the true maximum within six base-2 units.

Deferred rescaling changes tensor-memory traffic, never the returned normalized output or LSE — only floating-point rounding distinguishes it from eagerly rescaling after every increase.

Correctness, benchmarking, and who owns what

Correctness gates timing; five owners share the result

- A timing row is comparable only when kernel mode, shape, cache format, hardware, and correctness gate all match. Local H200 rows for dense decode, sparse decode, and sparse prefill each passed their *out*, ℓ gates [OBSERVED: VOL2_KERNEL_SWEEP_H200_342882 and successors] before any timing was recorded.

Correctness gates timing; five owners share the result

- A timing row is comparable only when kernel mode, shape, cache format, hardware, and correctness gate all match. Local H200 rows for dense decode, sparse decode, and sparse prefill each passed their out, ℓ gates [OBSERVED: VOL2_KERNEL_SWEEP_H200_342882 and successors] before any timing was recorded.
- Five separate owners share responsibility for the served answer: the **serving runtime** (batching, cache allocation, sparse-index construction), the **FlashMLA call site** (converts runtime state into documented tensors), the **kernel** (computes output and LSE), a **correctness harness** (compares returns under tolerances), and a **benchmark harness** (times a correctness-gated call).

Correctness gates timing; five owners share the result

- A timing row is comparable only when kernel mode, shape, cache format, hardware, and correctness gate all match. Local H200 rows for dense decode, sparse decode, and sparse prefill each passed their out, ℓ gates [OBSERVED: VOL2_KERNEL_SWEEP_H200_342882 and successors] before any timing was recorded.
- Five separate owners share responsibility for the served answer: the **serving runtime** (batching, cache allocation, sparse-index construction), the **FlashMLA call site** (converts runtime state into documented tensors), the **kernel** (computes output and LSE), a **correctness harness** (compares returns under tolerances), and a **benchmark harness** (times a correctness-gated call).

Kernel correctness establishes the attention result for one call. End-to-end serving correctness needs all five to hold together — SGLang's `FlashMLABackend` makes this split concrete, not hypothetical.

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`
- **the public operator boundary** → `FlashMLA/csrc/api/api.cpp`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`
- **the public operator boundary** → `FlashMLA/csrc/api/api.cpp`
- **dense-decode eligibility gate** → `FlashMLA/csrc/api/dense_decode.h`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`
- **the public operator boundary** → `FlashMLA/csrc/api/api.cpp`
- **dense-decode eligibility gate** → `FlashMLA/csrc/api/dense_decode.h`
- **SM100 gather4 and lazy-anchor state** → `FlashMLA/csrc/sm100/prefill/sparse/fwd/head128/phase1.cuh`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`
- **the public operator boundary** → `FlashMLA/csrc/api/api.cpp`
- **dense-decode eligibility gate** → `FlashMLA/csrc/api/dense_decode.h`
- **SM100 gather4 and lazy-anchor state** → `FlashMLA/csrc/sm100/prefill/sparse/fwd/head128/phase1.cuh`
- **dtype-aware correctness tolerances** → `FlashMLA/tests/test_flash_mla_sparse_decoding.py`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`
- **the public operator boundary** → `FlashMLA/csrc/api/api.cpp`
- **dense-decode eligibility gate** → `FlashMLA/csrc/api/dense_decode.h`
- **SM100 gather4 and lazy-anchor state** → `FlashMLA/csrc/sm100/prefill/sparse/fwd/head128/phase1.cuh`
- **dtype-aware correctness tolerances** → `FlashMLA/tests/test_flash_mla_sparse_decoding.py`
- **the benchmark card, as code** → `FlashMLA/benchmark/bench_flash_mla.py`

Concept-to-code map for the chapter

- **addressing and sparse indices** → `FlashMLA/tests/quant.py`
- **split-KV local online-softmax update** → `FlashMLA/csrc/sm90/decode/dense/splitkv_mla.cuh`
- **cp.async staging (SM80 analog of TMA)** → `FlashMLA/csrc/kerutils/.../sm80/intrinsics.cuh`
- **the public operator boundary** → `FlashMLA/csrc/api/api.cpp`
- **dense-decode eligibility gate** → `FlashMLA/csrc/api/dense_decode.h`
- **SM100 gather4 and lazy-anchor state** → `FlashMLA/csrc/sm100/prefill/sparse/fwd/head128/phase1.cuh`
- **dtype-aware correctness tolerances** → `FlashMLA/tests/test_flash_mla_sparse_decoding.py`
- **the benchmark card, as code** → `FlashMLA/benchmark/bench_flash_mla.py`

Store less, name a selection, reconstruct exactly.

Next: the DeepGEMM chapter applies this same method — kernel correctness, evidence-scoped benchmarking, an explicit responsibility split — to projection and grouped-expert matrix products.