Kernel Design

Prateek Shukla

General format of CUDA kernels

There are 2 types of CUDA kernels

- Compute bound kernels (limited by the rate of arithmetic operations)
- Memory bound kernels (it's limited by the rate of data movement)

The crossover point is the arithmetic intensity(AI) = FLOPs / Bytes moved

Ridge point = Peak FLOPS / Peak BW

≈ ~295 FLOP/Byte(for FP16 tensor ops) ≈ ~20 FLOP/Byte(for FP32 CUDA cores)

- If your kernel's AI < ridge point ⟶ memory-bound

- If your kernel's AI > ridge point ⟶ compute-bound

Compute bound kernels

A compute-bound CUDA kernel is one where the limiting factor is math throughput rather than moving data. These are the characteristics of a compute bound kernel

1. High Arithmetic Intensity: Lots of FLOPs per byte loaded/stored, Reuse is strong: data is kept in registers / shared memory and used many times before being evicted etc.
2. Producers are Idle: The most distinct characteristic is the Producer/Consumer imbalance. The producer warpgroups are sitting idle at barriers, waiting for the consumer warpgroups to catch up
3. Occupancy is “necessary but not sufficient”. Many compute-bound kernels can run near peak with moderate occupancy if they launch enough WGMMA instructions. Launching compute instructions is not issue, register utilization is.

Optimizing compute bound kernels

These are some of the optimizations methods for compute bound kernels

- Warp Specialization
- Persistent Kernels + Tile Scheduling
- Circular buffer of shared memory stages with explicit synchronization
- Cluster-Level Optimizations
- Register Pressure Management
- Megakernels
- Epilogue Fusion

Warp Specialization

On Hopper, an SM can host many active warps, but only a small number can issue each cycle

If threads inside a warp take different branches, the warp serializes those paths (worst-case ~32×). But if warp 0 does loading of the data while warp 1 does compute on that data, those are different warps with independent execution contexts so you avoid the SIMT penalty.

With warp specialization we are intentionally giving different warps in the same thread block different jobs, typically:

“producer” warps: move/prepare data

“consumer” warps: compute on that data

Why warp specialization helps

There are 3 reasons why warp specialization is almost mandatory for fast kernels -

1. Resource constraints force it: You can’t fit the whole live state (regs/preds/etc.) per-thread/warp without spilling, so you split the work across warps
2. Variable-latency ops are hard to schedule statically: Memory and other variable-latency operations make it difficult for a compiler/static schedule to keep all units busy.
3. Blocking synchronization would otherwise stall issue: If a warp must wait for barriers etc, specializing lets other warps run immediately, so the SM doesn’t waste issue slots.

Resource constraints

Resource constraints primarily register file capacity and latency hiding requirements are the driving force behind Warp Specialization.

The primary bottleneck in wgmma kernels is the Register File. To maximize compute throughput, threads must hold a large tile of the output matrix in registers. However, allocating ~200+ registers per thread drastically reduces the number of warps that can fit on an SM.

WS decouples the work into two roles, allowing asymmetric resource allocation

Producers: They issue cp.async instruction, requiring lowest number of registers

Consumers: These perform the WGMMA instructions, requiring maximum registers

setmaxnreg

```
setmaxnreg.action.sync.aligned.u32 imm-reg-count;

.action = { .inc, .dec };
```

The setmaxnreg instruction allows a warp to dynamically change the number of registers it owns during execution.

You can launch a kernel with a low register count. This allows the GPU to fit many active warps on the SM, maximizing memory bandwidth utilization.

Just before entering the heavy compute section, the consumer warp executes setmaxnreg to request more registers, producers launch this to use less registers

Warp group count selection

For producers always allocate exactly 1 Warp Group. A single thread can issue a cp.async instruction that moves gigabytes of data.

Producers do very little math; they just manage barriers (mbarrier). You can cap them at ~32 registers using `set_maxnreg`.

For the Consumers, choose between 1, 2, or (rarely) 3 based on register pressure. The number of consumer WGs is determined by the size of your accumulator Tile.

Budget: ~232-240 registers (leaving room for barriers etc). We use this in two ways

Pingpong/Basic: Each WG works independently on a full tile. EffectiveThreads = 128

Cooperative: Two WGs split the same tile. EffectiveThreads = 256

Pipelining

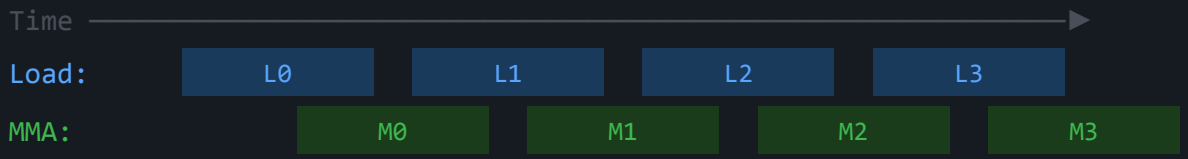
Pipelining is just overlap: you break work into stages and run different stages for different “items” at the same time, instead of doing one item start→finish before starting the next.

Without Pipelining (Sequential)

Load → Wait → Compute → Wait → Load → Wait → Compute → Wait ...

Each stage blocks until the previous one completes. Hardware sits idle during transfers.

With Pipelining (Overlapped)



Compute on tile N while loading tile N+1 — hardware stays busy

Why getting it right is Important

The Latency Gap: Global memory ~400-800 cycles. WGMMA ~30-60 cycles per operation. Without pipelining, compute waits 20x longer than it runs.

Without pipelining, your kernel looks like this: load a tile (400 cycles of compute sitting idle), compute on it (20 cycles of memory sitting idle), load next tile (400 more idle cycles). You're utilizing your compute hardware roughly 5% of the time.

With pipelining, while compute is chewing through tile N, memory is already fetching tile N+1. If the pipeline is deep enough, compute never stalls waiting for data. You go from 5% to near-100% compute utilization. That's the entire game.

Circular buffering

To overlap producers and consumers, you need a buffer between stages: a fixed set of shared-memory “slots” (stages) the producer fills and the consumer drains.

Circular buffering = reuse those slots round-robin:

Producer writes stage i, then i+1, ..., wraps back to 0.

Consumer reads stage i, then i+1, ..., wraps back to 0.

This keeps compute fed while hiding long memory latency (load ~400 cycles vs compute ~20 cycles): the buffer is what allows to load tile N+1 while computing tile N.

Stages and why we need them

Circular buffering with multiple stages is about keeping a pipeline busy, not waiting. If you have stages like Load -> Compute -> Store, a single buffer forces serialization. That wastes time because only one stage works at once.

Multiple buffers solve this by allowing multiple if flight chunks of data at once. While stage 1 is filling buffer A, stage 2 can process buffer B, and stage 3 can drain buffer C. The circular part means once a buffer finishes the full journey, it is reused for new input, wrapping around in a ring. So you don’t need unbounded memory, just enough buffers to keep all stages occupied.

The reason we need more than one buffer is ownership and overlap. A stage cannot safely overwrite data that another stage is still reading. Separate buffers let each stage own different data at the same time. In practice, each buffer moves through states, and barriers transfer ownership from one stage to the next.

The Two-Barrier Handshake

Each stage in the circular buffer has two signals, so producer and consumer never race:

1) FULL barrier: Consumer waits here before reading a stage. For TMA pipelines, it's a transaction barrier: producer calls mbarrier.arrive and the TMA engine signals completion when those bytes land in shared memory. Thus Consumer knows the stage is genuinely populated without polling.

2) EMPTY barrier: Producer waits here before overwriting a stage. Consumer signals it after it's done using that stage.

Producer waits EMPTY → declares expected TX on FULL → issues TMA → TMA completes → FULL trips → Consumer reads/uses → Consumer arrives EMPTY → stage reusable

ABA ambiguity in circular-buffer synchronization

A circular buffer reuses the same stage indices across multiple passes. If synchronization is indexed only by stage_id $\in$ [0..Stages-1], then a consumer that observes “stage 0 is full” cannot determine whether that fullness corresponds to:

Pass 1, stage 0 (old data), or

Pass 2, stage 0 (new data after wrap-around all the 4 stages)

To disambiguate multiple uses of the same stage index, the pipeline tracks a phase bit that flips whenever the circular index wraps from Stages 1 $\rightarrow$ 0. Each participant (producer/consumer) tracks a state triple:

index: current stage slot in [0..Stages-1], phase: 1-bit epoch toggled on wrap around, count: monotonically increasing iteration counter (used for bookkeeping)

Iteration of counter and phase with each k tile

```
void operator++() {
  ++index_;
  ++count_;
  if (index_ == Stages) {
    index_ = 0;     // wrap around
    phase_ ^= 1;    // flip phase bit on wrap
  }
}
```

Reuse of barrier

Iteration	index	phase	Physical barrier
0	0	1	barrier[0] with parity=1
1	1	1	barrier[1] with parity=1
2	2	1	barrier[2] with parity=1
3	0	0	barrier[0] with parity=0 ← reuse
4	1	0	barrier[1] with parity=0 ← reuse
...	...	...	...

Producer Flow

Only one elected thread per warpgroup actually touches mbarrier and TMA. Each iteration does three things:

Acquire the stage. Wait on the empty barrier, then call expect_tx on the full barrier

Issue the TMA copies. When the DMA engine finishes writing bytes to shared memory, it automatically signals the full barrier. This is hardware-level producer consumer signaling.

Advance. Increment the pipeline state. Phase flips on wrap.

At the end of all work, `mbarrier.try_wait` waits for consumers to release every remaining stage before the warp exits. You can't exit while someone is still reading your data.

Consumer Flow

The consumer maintains two pointers into the pipeline to the stage being consumed right now and the stage that can be released back to the producer.

Because WGMMA is asynchronous, it doesn't complete immediately. You can't release the buffer it's reading from until you're sure the read is done. You can only know that reading from a buffer has provably completed (via `warpgroup_wait`)

Each iteration: wait for data (full barrier) → issue WGMMA (fence, arrive, gemm, commit) → wait for the oldest WGMMA to finish (`wgmma.wait_group<N>`) → release the oldest buffer (empty barrier) → advance both pointers.

The cross proxy fence before each WGMMA prevents the compiler from reordering accumulator reads/writes across the WGMMA boundary.

Three Phases

Prologue (Fill). The consumer issues N WGMMAs without releasing any buffers. This fills the pipeline so that by steady-state, there are always in-flight WGMMAs overlapping with TMA loads. The first WGMMA initializes accumulators to zero.

Steady-State. The throughput-optimal loop: for each k-tile, wait data → issue WGMMA → wait oldest → release oldest. TMA and WGMMA are fully overlapped. The producer stays Stages steps ahead of the consumer's release pointer.

Drain. After the last k-tile is consumed, call `warpgroup_wait<0>()` to flush ALL outstanding WGMMAs, then release the remaining N buffers so the producer can exit cleanly.

Two major types of pipelines

There are 2 main types of pipelines which we use for high performance kernels. Ping pong pipeline, cooperative pipeline.

Both cooperative and ping-pong pipelines use warp specialization: split your threads into producers (who load) and consumers (who compute), give them a circular buffer in shared memory, and let them run simultaneously. The producer stays ahead, filling buffer slots with future tiles, while the consumer works through slots that are already filled.

The difference between the two architectures is what happens after the consumer finishes multiplying during the epilogue. Ping pong overlaps the wgmma operations with epilogue, cooperative does not.

Cooperative Pipeline — The Architecture

384 threads, split into 3 warp groups:

WG0 : Producer — issues TMA loads

WG1: Consumer — runs WGMMA, then does the epilogue

WG2: Consumer — WGMMA and epilogue on same output tile.

The circular buffer has a fixed number of stages. Each stage has two barriers: a full barrier that consumer waits on, and an empty barrier that producer waits on.

The producer fills stages round-robin. The consumer reads them round-robin, offset by the pipeline depth. Phase bits disambiguate which pass through the buffer you're on, preventing stale reads.

Looping strategy

If your chosen output tile is 128x128, the warpgroups divide the geometric workload of that single tile:

Consumer 0 computes the top half (e.g., M = 0 to 63).

Consumer 1 computes the bottom half (e.g., M = 64 to 127).

Because they are working on the exact same output tile simultaneously, they consume the exact same A and B tiles from Shared Memory at the exact same time. Because we use persistent scheduling each CTA loops to grab a sequence of tiles Every warpgroup steps through the exact same sequence of tiles.

Producer, consumer0, consumer1 all loop through tile T0, T1, T2, T3 if that’s the way tiles are assigned

Loop for producers

```
__device__ void run_producer_loop(int base_tile, int num_tiles_for_cta) {
    for (int t = 0; t < num_tiles_for_cta; t += 1) {
        int target_tile = base_tile + t;
        // Inside the Producer's Outer Loop:
        for (int k = 0; k < K_TILE_COUNT; ++k) {
            // 1. Wait for empty_barrier[current_index] to be ready
            wait_barrier_phase(&empty_barrier[current_index], current_phase);
            // 2. TMA Loads...
            // 3. Advance state
            advance_pipeline_state(1);
        }
    }
}
```

Loop for consumers

```
__device__ void run_consumer_loop(int base_tile, int num_tiles_for_cta, int my_wg_idx) {
    // Both consumers start perfectly in sync.-
    // NO fast-forwarding the pipeline state!
    for (int t = 0; t < num_tiles_for_cta; t += 1) {
        int target_tile = base_tile + t;
        // Inside BOTH Consumers' Outer Loops:
        for (int k = 0; k < K_TILE_COUNT; ++k) {
            // 1. Wait for full_barrier[current_index] to be ready.
            wait_barrier_phase(&full_barrier[current_index], current_phase);
            // 2. Execute WGMMA using data in SMEM_A[current_index] and SMEM_B[current_index].
            // (Consumer 0 runs math using the top-half registers, Consumer 1 uses bottom-half)
            // 3. Signal empty_barrier[current_index].
            asm volatile("mbarrier.arrive.shared::cta.b64 _, [%0];" :: "l"(&empty_barrier[current_index]));
            // 4. Advance current_index (wrap around) and flip phase if wrapped
            advance_pipeline_state(1);
        }
        // ⟶ Execute Epilogue for target_tile (Only store my specific half)
    }
}
```

Cooperative — The Consumer Loop

The consumers maintain two pointers into the pipeline: a read pointer and a release pointer, offset by N (typically 1). The lag exists because WGMMA is asynchronous — you can't release a buffer until the WGMMA reading from it has actually finished.

We first wait on the read cursor, issue WGMMA for that stage, and advance the read cursor. Only after the corresponding wgmma work is complete (`wgmma.wait_group`) can we safely release the older stage and advance the release cursor. That lag between read and release is exactly what protects correctness.

After the K-loop is done, we drain outstanding wgmma ops so accumulators are fully valid in registers. Then epilogue begins: we run the post-processing on those register accumulators (like scale/bias/activation), move the data to shared memory for some operations, and finally write the completed result out to global memory.

Cooperative kernels do NOT reduce WGMMA results

This is the key insight. The "cooperation" between warpgroups is about sharing smem tiles, not about merging accumulators. Each warpgroup independently performs wgmma over all K tiles and accumulates into its own register-resident accum. No cross-WG register communication ever happens that's physically impossible anyway since warpgroup registers are private.

Each WG writes its own disjoint M-region, The epilogue uses the thread index (which encodes the warpgroup) to TMA-store only the M-rows that this WG computed, to the correct region of the global output matrix.

The benefit over pingpong is that a single larger tile can be mapped by two WGs together, enabling larger effective MMA tiles (256×128 instead of 128×128) while still fitting the register budget per WG.

Cooperative — The Gap

After the consumer finishes accumulating all k-tiles for an output tile, it has to do the epilogue: scale the result, apply bias, run activation, store to global memory. During that entire epilogue phase, the Tensor cores sit completely idle. Nobody is using them.

C0: [MMA Tile 0] [Epilogue Tile 0] [MMA Tile 1] [Epilogue Tile 1]

^^^^^^^^^^^^^^^^ WGMMA idle

For large epilogues or small k-dimensions, this idle time is a significant fraction of total runtime. The pipeline perfectly overlaps loads with compute, but it can't overlap compute with the epilogue because there's only one consumer doing both jobs.

Ping-Pong — The Fix

Add a second consumer. While one consumer runs its epilogue, the other does MMA on the next tile. They alternate, and the WGMMA units never go idle.

384 threads, split into 3 warp groups:

WG0: Producer, WG1: Consumer(C0), WG2: Consumer(C1)

The producer warp group deallocates its registers to give more register file space to the two MMA warp groups.

```
C0: [MMA T0][Epi T0][MMA T2][Epi T2]

C1:         [MMA T1][Epi T1][MMA T3][Epi T3]
```

C1's MMA overlaps C0's epilogue. C0's MMA overlaps C1's epilogue. WGMMA is always busy.

Ping pong loop

Hopper kernels usually use Persistent Scheduling. You launch a smaller grid (to fit perfectly on the SMs), and each CTA loops to grab a sequence of tiles.

Let's assume this specific CTA has been assigned a sequence of tiles: T0, T1, T2, T3, T4, T5. The contract dictates:

Producer must process all of them: step size = 1.

Consumer 0 must process the even ones: step size = 2, start = 0. Consumer 1 must process the odd ones: step size = 2, start = 1.

Inside the outer loop, you process one specific output tile. To do a GEMM for one output tile, you iterate along the K dimension. If your K dimension is 4096, and your tile block size is `K_TILE=64`, then `K_TILE_COUNT = 64`.

Loop structure of producers

```
__device__ void run_producer_loop(int base_tile, int num_tiles_for_cta) {
    // Step size is 1. Processes T0, T1, T2, T3...
    for (int t = 0; t < num_tiles_for_cta; t += 1) {
        int target_tile = base_tile + t;

        // Inside the Producer's Outer Loop:
        for (int k = 0; k < K_TILE_COUNT; ++k) {
            // 1. Wait for empty_barrier[current_index] to be ready
            wait_barrier_phase(&empty_barrier[current_index], current_phase);
            // 2. Issue TMA Load for A[target_tile_M, k] → SMEM_A[current_index]
            // 3. Issue TMA Load for B[k, target_tile_N] → SMEM_B[current_index]
            // (Note: TMA automatically signals full_barrier when data arrives)
            // 4. Advance current_index (wrap around at Stages=3) and flip phase if wrapped
            advance_pipeline_state(1);
        }
    }
}
```

Outer loop for consumers

```
__device__ void run_consumer_1_loop(int base_tile, int num_tiles_for_cta) {
    // THE CONTRACT TRICK:
    // We don't process Tile 0. We must initialize our pipeline state
    // to fast-forward past Tile 0's data, waiting for Tile 1's data.
    advance_pipeline_state(K_TILE_COUNT);
    // Step size is 2. Starts at 1. Processes T1, T3, T5...
    for (int t = 1; t < num_tiles_for_cta; t += 2) {
        int target_tile = base_tile + t;
        // ⟶ Execute Level 2: Inner K-Tile Loop (see below)
        // ⟶ Execute Epilogue for target_tile
        // Fast-forward past the tile Consumer 0 is doing next
        advance_pipeline_state(K_TILE_COUNT);
    }
}
```

Inner loop

```
for (int k = 0; k < K_TILE_COUNT; ++k) {
    // 1. Wait for full_barrier[current_index] to be ready
    wait_barrier_phase(&full_barrier[current_index], current_phase);
  ---
    // 2. Execute WGMMA using data in SMEM_A[current_index] and SMEM_B[current_index]
  ---
    // 3. Signal empty_barrier[current_index] so Producer can overwrite it later
    asm volatile("mbarrier.arrive.shared::cta.b64 _, [%0];" :: "l"(&empty_barrier[current_index]));
  ---
    // 4. Advance current_index (wrap around at Stages=3) and flip phase if wrapped
    advance_pipeline_state(1);
}
```

Ping pong pipeline

Each consumer gets exactly k_tile_count stages — where k_tile_count is the runtime value of K / TileK (how many K-tiles to reduce over for one output tile). We generally set the stages to be a fixed number (like 1, 2, 3, 4) and k_tile_count can be much larger (K=4096, TileK=64 ⟶ k_tile_count=64)

So the actual smem layout with Stages=4 and k_tile_count=64 is a circular buffer reused by both consumers:

Physical smem: [slot0][slot1][slot2][slot3] ⟵ only 4 slots exist

C0 logical: pos 0,1,2,...,63 ⟶ maps to slots 0,1,2,3,0,1,2,3,... ()

C1 logical: pos 64,65,...,127 ⟶ maps to slots 0,1,2,3,0,1,2,3,... (different phase)

Ping pong pipeline

The producer begins filling the 3-stage buffer with K-tiles for Output Tile 0. Consumer 0 immediately begins its MMA. Consumer 1 is hard-blocked by the mbarrier and sits completely idle. After chewing through all K-tiles for Output Tile 0, Consumer 0 calls arrive() on the order barrier

Consumer 0 unblocks its own Epilogue phase and starts writing Tile 0 to global memory. Simultaneously, Consumer 1 is unblocked. It calculates its starting logical index (which tells it exactly which phase of which barrier corresponds to the start of Tile 1) and begins its MMA

Once Consumer 1 finishes its MMA, it signals the barrier to unblock Consumer 0's next MMA, while Consumer 1 moves to its Epilogue

Ping pong pipeline and barriers

In a pipeline with 3 stages, the hardware does not allocate a new barrier for every single iteration. It allocates exactly 3 physical barriers in shared memory.

Consumer0 always waits on whichever generation of the barrier corresponds to its output tile 0's data. Consumer1 waits on the generation that corresponds to output tile 1's data.

Consumer0 and Consumer1 are always k_tile_count slots apart. They physically reuse the same `full_barrier_[i]` across iterations, but with different phase bits. Because Consumer 0 and Consumer 1 are working on entirely different output tiles, they never fight for the same pipeline barrier at the same time.

Tile sizes

The output tile dictates how large of a chunk of the C matrix a single CTA computes

The Goal: Make `tile_m` and `tile_n` as large as possible. Why? Because every time you load a tile of A and a tile of B, you perform `2×tile_m×tile_n×tile_k` math operations. A larger Output Tile gives you a higher Arithmetic Intensity (more math per byte loaded from global memory).

Small/Medium Tiles (e.g., 128 x 128 or 64 x 128): A single Consumer WG (128 threads) has enough registers to hold the accumulators. We use Base or Ping-Pong

Massive Tiles (e.g., 256 x 128 or 128 x 256): A single Consumer WG physically runs out of registers to hold a 256 x 128 output matrix in FP32. we generally use Cooperative in this case

The Inner Tile (`tile_K`) and Pipeline Stages

`tile_k` is how "deep" you traverse the K dimension per inner loop iteration. We size it to keep the Tensor Cores fed without blowing up your Shared Memory.

For epilogue overlap to be efficient, your K dimension must be large enough. If the K-dimension is too small, Consumer 1 will finish its MMA before Consumer 0 finishes its Epilogue. If that happens, Consumer 1 will stall again, and you lose the throughput benefits of the ping-pong design. The compute time must be greater than or equal to the memory write time.

The Constraints:

WGMMA instructions natively consume K=16 (for FP16/BF16) or K=16 (for TF32) natively.

You need multiple pipeline Stages (usually 3 or 4) to hide TMA load latency.

Tracking phase for ping pong warp specialized pipeline

`count_` (logical)	`index_` (physical slot)	`phase_`
0	0	0
1	1	0
2	2	0
3	3	0
4	**0**	**1** (first wrap → flip)
5	1	1
6	2	1
7	3	1
8	**0**	**0** (second wrap → flip back)
...	...	...
63	3	1

Roles inside a warpgroup

Producer warp group (128 threads, 4 warps) — each warp has a distinct role:

- Mainloop DMA warp: TMA loads of A and B tiles into shared memory
- Epilogue DMA warp: TMA loads of C tiles into shared memory (for residual add)
- Scheduler warp: Fetches next tile coordinates
- Auxiliary warp: Optional extra loads

Consumer warp group 0 (128 threads) — math + epilogue for even tiles

Consumer warp group 1 (128 threads) — math + epilogue for odd tiles

Epilogue handoff ping pong

The core objective of this handoff mechanism is secure, efficient sequencing ensuring that heavy Math computation and Epilogue flushes occur seamlessly between the alternating consumers without data corruption or pipeline stalls.

A dedicated Producer Warp strictly manages the epi_load_pipeline, issuing non-blocking TMA loads to bring elements like Tensor C into Shared Memory for operations like D= A X B+ C.

Meanwhile, Consumer 0 and Consumer 1 share the epilogue pipeline, creating a shared-resource bottleneck that requires precise sequencing to avoid race conditions in Shared Memory.

These Consumer warps are responsible for consuming MMA accumulators, applying the epilogue math (e.g., ReLU or scaling), and commanding the TMA unit to safely store Tensor D into global memory.

Barriers with epilogue

The most magical part of the ping pong design is that Consumer 0's Epilogue runs at the exact same time as Consumer 1's MMA. To do this safely without corrupting shared memory or issuing conflicting TMA stores we use 2x2 grid of mbarrier objects in shared memory.

Rows (Stages): 0 = MMA Phase, 1 = Epilogue Phase (phase is not)

Columns (Groups): 0 = Consumer 0, 1 = Consumer 1

Each consumer has a group_id (0 or 1). When a consumer calls arrive(), it signals the other consumer's barrier at the current depth stage. When it calls wait(), it waits on its own barrier at the current depth stage.

The depth dimension cycles: MMA phase → Epilogue phase → MMA phase → ....

The 2x2

Each consumer does two ordered operations per tile iteration: first MMA, then epilogue. We need to enforce:

1. Only one consumer does MMA at a time
2. Only one consumer does epilogue at a time
3. Each consumer does MMA before epilogue for the same tile

barrier_[depth][length]

	Consumer 0	Consumer 1	
depth 0:	bar[0][0]	bar[0][1]	← MMA handoffs
depth 1:	bar[1][0]	bar[1][1]	← Epilogue handoffs

Each consumer has a group_id. The protocol for each operation:

How does it work

You have two workers (Consumer 0 and 1) and two zones (MMA and Epilogue)

If both workers enter the same zone at the same time, shared memory corrupts. If one worker waits for the other to finish completely, the Tensor Cores sit idle and you waste cycle time. The 2x2 grid solves this by decoupling the phases.

Row 0 (MMA): Consumer 0 takes the lock, computes the math, and signals Consumer 1. Consumer 1 is now cleared to start math.

Row 1 (Epilogue): Consumer 0 immediately steps down into the Epilogue row to write its results to global memory.

Because there are two independent rows, Consumer 0's Epilogue runs at the exact same time as Consumer 1's MMA. The hardware achieves perfect overlap. The Tensor Cores never starve

The Full Timeline of One Warp Group's Iteration

1. ordered_barrier.wait() ← wait for Consumer1's prior epilogue
2. WGMMA mainloop ← K-loop: load from smem, accumulate into registers
3. ordered_barrier.arrive() ← "my MMA done, Consumer1 can start its MMA"
4. mma_tail() ← warpgroup_wait<0>, release last smem pipeline stages
5. ordered_barrier.wait() ← wait for Consumer1's epilogue to finish
6. epilogue.store() ← fuse + R→S copy + TMA store (details next slides)
7. epilogue.store_tail() ← wait for all TMA stores to land
8. advance pipeline states ← skip ahead by 2 (one for each warp group)
9. ordered_barrier.arrive() ← "my epilogue done, Consumer1 can start its epilogue"
10. fetch next tile, loop back to 1

The three independent pipelines

Pipeline	Producer	Consumer	What it carries
Mainloop pipeline	DMA warp (TMA loads A,B)	Math warp group (WGMMA)	A and B tiles
Epilogue load pipeline	Epilogue DMA warp (TMA loads C)	Math warp group (epilogue)	C tiles for residual
Epilogue store pipeline	Math warp group (epilogue)	TMA unit (async store)	D tiles to write out

Notice the math warp group switches roles: during MMA it's a consumer of the mainloop pipeline, but during epilogue it becomes the producer of the store pipeline. The same threads drive both the computation and the store.

Each consumer warp group runs this loop per tile:

```
// ── MMA PHASE ──
  order_barrier.wait()          // Wait for my turn to start MMA
  wgmma(accumulators, ...)      // Run the mainloop: K iterations of WGMMA
  order_barrier.arrive()        // Signal: the other consumer can start its MMA

  mma_tail(...)                 // Drain the mainloop pipeline for my tile
  advance mainloop state by 2×k_tile_count  // Skip over the other consumer's tiles

  // ── EPILOGUE PHASE ──
  order_barrier.wait()          // Wait for my turn to start epilogue
  epilogue.store(...)           // Load C from smem, fuse, write D to smem, TMA store to gmem
  epilogue.store_tail(...)      // CRITICAL: wait for ALL my TMA stores to complete
  order_barrier.arrive()        // Signal: the other consumer can start its epilogue

  advance epilogue states by 2×subtiles  // Skip over the other consumer's epilogue slots
```

Inside epilogue store

The CTA tile (e.g., 128×128) is too large to move to smem all at once. It's broken into epilogue subtiles (e.g., 64×64 or 128×32). The epilogue loops over these subtiles:

```
1. LOAD C (if needed):
   consumer_wait(epi_load_pipeline)    // wait for C subtile in smem
   copy(smem → registers)             // S2R: load C from smem to registers
   consumer_release(epi_load_pipeline) // free the C smem buffer

2. FUSE (register-to-register):
   for each vector fragment in the subtile:
     result[v] = visit(accumulator[v], ...)
   // visit() applies: scale, bias, activation, beta*C + alpha*acc, etc.
   // All in registers — no smem touch

3. CONVERT (register format change):
   convert accumulator type (FP32) → output type (FP16/BF16)
   // NumericArrayConverter handles vectorized conversion

4. STORE to smem (R2S):
   producer_acquire(epi_store_pipeline)  // wait for smem D buffer to be free
   synchronize()                         // named barrier — all threads aligned
   copy(registers → smem)               // R2S copy into smem D buffer

5. TMA STORE (S2G):
   fence_view_async_shared()             // make smem writes visible to TMA
   synchronize()                         // all threads confirm fence
   copy(tma_store_d, smem → gmem)        // TMA async bulk store
   producer_commit(store_pipeline)        // mark this store stage as committed
```

Epilogue load producer

While all of this is happening, the epilogue DMA warp (one of the 4 producer warps) has been independently loading C tiles into shared memory:

```
for each output tile:
  load_order_barrier.wait()   // wait for mainloop DMA to start first
  collective_epilogue.load()  // TMA load C tile from gmem → smem
  advance to next tile
```

The load_order_barrier ensures the mainloop DMA warp loads the first A/B tile before the epilogue warp starts loading C. After that, they run independently — the epilogue DMA warp fills C buffers ahead of time, and the math warp group consumes them via the epi_load_pipeline when it reaches the epilogue phase.

This means C data is likely already in smem by the time the math warp group needs it, overlapping the C load with the WGMMA compute of A*B.

Cooperative epilogue handoff

For each epilogue subtile, every thread in both consumer warpgroups executes:

Step 1: Load source matrix C (optional)

Step 2: Apply element-wise operations in registers

Step 3: Type conversion

Step 4: Register ⟶ Shared Memory (R2S copy)

Step 5: Shared Memory ⟶ Global Memory (TMA store)

Loading C and storing D

if the source matrix C and destination matrix D have the same element size, they can share the same shared memory buffers. No need to allocate separate smem regions for C loads and D stores. This is the protocol:

1. Producer loads C subtile into smem buffer at phase P
2. Consumers read C from that buffer (S2R copy)
3. Consumers write D into that same buffer (R2S copy)
4. TMA stores D from that buffer to global memory
5. Only after the TMA store commits do we release that buffer back to the producer for loading the next C subtile

R2S copy

The output tile (`CTA_M × CTA_N`) is too large to move from registers to shared memory all at once we don't have enough shared memory. So we break it into epilogue subtiles.

The output tile gets divided into a grid of smaller tiles (`EPI_TILE_M × EPI_TILE_N`) subtiles giving us a 2D grid: EPI_M subtiles in M × EPI_N subtiles in N

We then iterate over this grid in a nested loop:

```
for epi_n in 0..EPI_N:

   for epi_m in 0..EPI_M:

       process subtile (epi_m, epi_n)
```

Each iteration processes one subtile through the full registers→smem→gmem path. Both consumer warpgroups participate in every iteration their threads are partitioned across the subtile's elements, not across different subtiles.

Writing the data back

After the R2S copy, each thread has written its portion of the subtile to shared memory. But the TMA store needs the entire subtile to be present and visible. So we launch a cross proxy fence which makes sure that the thread's writes are visible and then we synchronize all threads across both consumer warpgroups using bar.sync to ensure every thread has completed both its R2S copy AND its fence before any thread proceeds.

Once all this is done the first thread of the first warp would issue the TMA store operation into the global memory. We use commit_group instruction to synchronize these operations. After issuing copy + commit_group, the thread doesn't wait for the data to reach global memory. This means we can overlap the TMA store of subtile N with the register computation of subtile N+1.

Integrating thread block clusters

Thread block clusters are a very important part of optimization in the pipelines, we generally are not using the DSMEM feature of clusters for wgmma ops, most of the cluster use is related to loading tiles for wgmma. Without clusters, adjacent thread blocks that compute neighboring tiles of C would independently issue slow HBM reads for the exact same tile of A.

We use TMA Multicast to issue exactly one read to GMEM for that shared A tile. The TMA unit then uses the high-speed inter-SM network to broadcast that tile directly into the local SMEM of every thread block in the cluster. This is the only use of clusters in the pipeline.

Blocks within a cluster are guaranteed to run concurrently on the same GPC. They can signal each other using cluster barriers residing in DSMEM.

Choosing the cluster size

Choosing the right cluster size is a crucial tuning step. Because clusters allow thread blocks to communicate directly and share data, picking the right shape can drastically improve performance by reducing global memory traffic

Shape of the Workload: Tall and Skinny (Large M, Small N): Use an M-heavy cluster

Short and Wide (Small M, Large N): Use an N-heavy cluster

Square / Large (Large M, Large N): Use balanced cluster size

Generally the 2 thread blocks inside a TPCs communicate a lot faster so its advised to use a cluster size of (1, 2) or (2, 1) to utilize that feature.

Cluster size 2

If ClusterSize = 2, the grid is divided into pairs of tiles.

1x2 Cluster (M=1, N=2): Both blocks are computing outputs on the same row of C but different columns. Therefore, they always share the A tile (A corresponds to the M row) and use different B tiles (B corresponds to the N column).

2x1 Cluster (M=2, N=1): Both blocks are on the same column of C but different rows. They always share the B tile and use different A tiles.

The persistent loop in your code changes. The step size is now based on how many cluster tiles you process. But crucially, the CTAs within a cluster are physically guaranteed by hardware to be co-scheduled and run concurrently.

RS vs SS

f16/bf16 (2 byte, equal width, no scales) -> SS always: These types have broad native SS GMMA support (including non-K-major cases), so both operands can stay in shared memory with no pre-transform step. This is the cheapest path (less register pressure)

Non 2 byte equal width (tf32/f32/fp8/int8) -> SS only for AkBk (TN), else RS: For these types, wgmma effectively wants K-major feeding. If input layout is already AkBk, SS can feed MMA directly. If not, you need operand swap and/or B transposition/repacking to make compatible ordering; that flexibility is implemented in RS (register-sourced A + transpose/swap machinery), so dispatch flips to RS.

Mixed widths -> RS: Mixed-width operands usually require conversion/dequantization before MMA. SS has no pre MMA transform stage. RS gives that stage via smem->register copy on A, where conversion can be done.

Scaled / tuple mixed input path -> RS: Scales/zero-points add extra per-element arithmetic (e.g., scale * A, maybe + zero) before wgmma. That transformation is done in registers, so RS is required

Decision Table

A type	B type	Layout	Scales?	IsInputSize2B	IsLayoutAkBk	IsABDiffWidth	HasScales	Result
f16	f16	TN	No	true	true	false	false	SS
f16	f16	NN/NT/TT	No	true	false	false	false	SS
bf16	bf16	TN	No	true	true	false	false	SS
bf16	bf16	NN/NT/TT	No	true	false	false	false	SS
f16	bf16	any	No	true	–	false	false	SS
tf32	tf32	TN	No	false	true	false	false	SS
tf32	tf32	NN/NT/TT	No	false	false	false	false	RS
f32	f32	TN	No	false	true	false	false	SS
f32	f32	NN/NT/TT	No	false	false	false	false	RS
e4m3	e4m3	TN	No	false	true	false	false	SS
e4m3	e4m3	NN/NT/TT	No	false	false	false	false	RS
e5m2	e5m2	TN	No	false	true	false	false	SS
e4m3	e5m2	TN	No	false	true	false	false	SS
e4m3	e5m2	NN/NT/TT	No	false	false	false	false	RS
int8	int8	TN	No	false	true	false	false	SS
int8	int8	NN/NT/TT	No	false	false	false	false	RS
f16	e4m3	any	No	–	–	true	false	RS
f16	int8	any	No	–	–	true	false	RS
bf16	e4m3	any	No	–	–	true	false	RS

Scheduling

Scheduling is the decision logic for who does what, when, and in what order. It is the specific policy + mechanism that maps Work Units (Tiles) onto Workers (Threads/Warps/CTAs/SMs).

Good scheduling distinguishes between all hardware busy and some SMs idle. It distinguishes between nice locality and thrashing and stalls. It ensures predictable completion times rather than suffering from tail-latency cliffs.

Scheduler distributes tile coordinates to workers. Standard Coordinates: (M_idx, N_idx, L_idx) for batched or grouped problems. Split-K Coordinates: Adds (K_idx, k_tile_count) when using Stream-K or Split-K strategies.

Why Scheduling Matters

The scheduler directly impacts three major performance vectors:

Occupancy, Utilization: Keep SMs fed with enough independent tiles to hide latency

Load Balance: Avoid the "long tail" effect where a few CTAs get huge tiles while others finish early and sit idle.

Locality & Bandwidth Efficiency: Maximize L2 cache reuse, minimize redundant global memory loads, enable better multicast opportunities (clusters).

Decoupling tile size from scheduling granularity: with persistent scheduling we have third axis of decomposition: the K dimension. In traditional tile-parallel system partition M×N output space across SMs. Each SM owns full-K tiles. This leads to different tradeoffs with waste.

Non persistent scheduling

In standard (non-persistent) kernels, you launch a grid of thread blocks where Grid Size = Total Work / Block Size. The GPU hardware scheduler assigns blocks to SMs, and when a block finishes, it retires, and the hardware schedules a new one.

There are three problems with this system. First, you have to launch the same kernel multiple times which comes with kernel launch overhead... second, you get tail effects where if you launch 133 blocks and GPU has 132 SMs then The hardware launches 132 blocks immediately wait for them to finish then launches only 1 block

Hardware schedulers are typically linear. They assign blocks in order, Block 0 might compute the top-left corner (0,0), and Block 1 might compute (0,1). By the time the hardware gets to the next row (1,0), the data needed for requested row would be evicted from the L2 cache because the grid spanned too wide across N.

Persistent Scheduling

In Persistent Scheduling, you launch a fixed number of thread blocks (usually equal to the number of SMs). These blocks stay persistent on the GPU. Instead of retiring after one tile, they enter a loop, calculate the index of the next tile of work, and process it, continuing until all work is done.

There are three schedulers which we are gonna learn in this lesson. They are:

1. Static Persistent Scheduler

2. Grouped Persistent Scheduler

3. Stream-K Scheduler

Baseline — Data-Parallel Scheduling

Each persistent CTA processes tiles in round-robin order:

CTA 0 gets tile 0, tile num_SMs, tile 2*num_SMs, ...

CTA 1 gets tile 1, tile num_SMs+1, ...

Total Work: 150 tiles, Hardware: 132 SMs

Total work = 1.136 tile-units per SM, Actual execution = 2 waves.

Utilization = 1.136/2 = 56.8%

The work is identical to a non-persistent kernel. The only gain is guaranteed ordering. Wave quantization remains.

Wave Quantization Elimination

With 132 SMs (H100) and 140 output tiles:

Non-persistent: ceil(140/132) = 2 waves, Wave 1: 132 SMs busy, Wave 2: 8 SMs busy, 124 idle ⟶ ~47% of last-wave capacity wasted

The remaining 8 tiles are divided evenly across all 132 SMs. Each SM computes approx 0.06 tiles worth of K-work for the tail and cross-SM reduction overhead is strictly limited to the final 8 tiles.

The fundamental insight is that wave quantization is not a hardware limitation it is a consequence of choosing tile-parallel decomposition. Persistent scheduling lets you choose a different decomposition.

Static Persistent Scheduler

This is the default, high-throughput scheduler for standard GEMM problems. It is "Static" because the mapping of work to threads is pre-calculated mathematically, not dynamically claimed via atomic counters.

It treats the output matrix as a grid of tiles. It assigns tiles to persistent thread blocks using a Rasterization (Swizzling) curve (often a Z-curve or U-curve). A persistent block calculates its first tile index, computes it, and then jumps ahead by the total number of launched blocks (Grid Stride Loop) to find the next tile.

We use this method for Standard GEMMs, Compute Bound kernels because it has the lowest overhead (pure math, no synchronization between blocks) and maximizes cache hits (swizzling).

Rasterization

After persistent scheduling, each persistent CTA maintains a linear work index. CTA 0 starts with tile 0, CTA 1 with tile 1, and so on. After finishing a tile, each CTA strides forward by the total number of persistent CTAs to grab its next assignment

This works. It's correct. And it absolutely destroys your performance.

Rasterization is the order in which CTAs are assigned to output tiles in the kernels. This order directly impacts L2 cache locality by making sure neighboring CTAs that share input data (rows of A or columns of B) execute close together in time. Get it wrong and you're reloading entire matrices from DRAM every time you step to a new tile

Why Traversal Order Matters: The L2 Cache Problem

To compute one output tile C[m, n], you need to load a strip of tiles from row m of matrix A, and a strip of tiles from column n of matrix B. You stream through the K dimension, accumulating partial results.

When tiles of A or B get loaded from HBM, they land in L2 and can be reused by subsequent tile computations BUT only if those subsequent computations actually need the same data

If you’re scanning left to right across a very wide matrix. You hold row m of A and iterate through columns 0, 1, 2, ..., tiles_n-1 of B. If tiles_n is large, by the time you finish the row and start row m+1 (where you'd reuse B's columns starting from 0), those early columns of B have been evicted from L2. Now you have to reload them from HBM all over again.

Path Strategy — AlongN vs AlongM

Rasterization uses two tile traversal strategies depending on the major (outer/slow) vs minor (inner/fast) axis:

Column-major: outer = N, inner = M. Finish a tile column top to bottom, then move right. You're sweeping down M for a fixed column n, every tile computation needs column n of B. That column stays hot in L2. You're cycling through different rows of A (which is the price you pay), but B gets maximum reuse

Row-major: outer = M, inner = N. Sweep across a tile row left-to-right, then move down. You're holding row m of A hot in L2 while cycling through columns of B. A gets maximum reuse.

Bottom line is whichever matrix you want to keep resident in cache, traverse perpendicular to its reuse dimension.

Swizzling

Even though we're reusing A perfectly (one row, held constant), we're streaming through B with zero reuse each column of B is loaded once, used once, and evicted.

When we get to the next row of M and start scanning again, we need those same B columns... but they're gone from L2. We're back to reloading B from HBM. We need locality in BOTH dimensions simultaneously so that the required tiles of both A and B fit in L2 and get reused multiple times.

We use swizzling which modifies the rasterization path so that instead of processing tiles in a thin line, the scheduler processes a "thick block" of tiles.

We tune swizzle size for thickness, if the swizzle size is 1, and you get standard thin raster. If size = 2, the swizzle size is 4, and the traversal becomes 4 tiles thick.

A few important points

You want the minor (fast/inner) axis to be the longer dimension Because a longer inner loop means more iterations before you take a "major step" and the major step is where the expensive cache context switch happens.

When you step in the major dimension, you're potentially shifting to a completely new strip of the other input matrix, which means new data needs to load into L2 and old data gets evicted.

The swizzle size is tunable. Larger swizzle = more locality = better L2 reuse, but at diminishing returns. The sweet spot depends on L2 cache size, tile size, and matrix dimensions. CUTLASS typically uses values like 1, 2, 4, or 8.

Clusters and swizzling

In plain swizzle, reuse is hoping that neighboring CTAs run close in time. Even with a perfect logical order, persistent CTAs finish at different times (edge predication, uneven math paths, split-K/reduction overhead). So physical execution order drifts away from logical order, and neighboring tiles may no longer be close in time. Good schedulers try to keep CTAs in the same locality region despite this drift.

Clusters solve this issue as CTAs inside a cluster execute together in space and time. They can synchronize and cooperate on data movement. Shared operand panels can be reused across CTAs in the cluster making reuse intentional, not accidental.

We get better L2 utilization as intra-cluster mapping defines local sharing pattern where inter-cluster rasterization defines global reuse flow. This minimizes reuse distance at both local and global levels.

Grouped Persistent Scheduling

This scheduler is designed for kernels like Grouped GEMM, where you want to compute multiple distinct GEMMs in a single kernel launch.

Instead of thinking of the workload as "Group 0, then Group 1...", the scheduler conceptually concatenates all tiles from all groups into one long linear sequence. The scheduler maintains current group ID of the problem (0, 1, 2...), the global linear index where this group begins, `total_tiles`: number of tiles in this specific group.

Groups have different sizes. So when a CTA's `linear_idx` moves forward (Progression is typically grid-stride: `linear_idx += grid_size` per tile), it may cross group boundaries. A naive scalar search is expensive. We use warp-level speculative search

warp-level speculative search

If `linear_idx` isn't in the current group, the warp scans groups in batches of 32. Each lane loads one group's shape and computes its (cluster-aligned) tile count. Using warp intrinsics (`__ballot_sync, __ffs, __shfl_sync`), it picks the lane whose range contains `linear_idx` and broadcasts that GroupInfo. If no hit, it jumps ahead by 32 groups and repeats.

After finding the owning group, compute the local offset `k = linear_idx - start_linear_idx`, swizzle k → (cluster_major, cluster_minor), convert to (M,N) per raster order (AlongM/AlongN), and add CTA-in-cluster offsets.

The scan is usually cheap because persistence amortizes it mostly triggered only at group boundaries.

Slow path

Warp lanes speculate on different groups. lane 0 checks group G, lane 1 checks G+1, ... lane 31 checks G+31

Each lane computes its candidate group's tile count (`total_tiles`), including padding for cluster/swizzle alignment.

Warp inclusive prefix sum computes cumulative tiles across those 32 candidates `shfl_sync` loop

Each lane derives its own candidate start offset (`start_linear_idx`) and checks if `linear_idx < start + total_tiles` using ballot sync

If one or more lanes match then you pick first matching lane (`__ffs`) and broadcast winner's `group_idx/start/total_tiles` to whole warp (`__shfl_sync`)

If none match you advance by 32 groups and repeat. Start offset is updated from lane 31 cumulative sum

The Problem with Data-Parallel Scheduling

The "Tail Effect": Standard schedulers assign full output tiles to Thread Blocks. If the total number of tiles isn't a perfect multiple of the available SMs, the final "wave" of work is only partially filled. During this tail wave, active SMs process their final tiles while the remaining SMs sit completely idle, waiting for the kernel to finish.

This inefficiency is a software decomposition problem, not a hardware limitation

Instead of viewing the problem as a 2D grid of output tiles, we introduce Stream-K scheduling which treats the entire matrix multiplication as a single, continuous 1D tape of math iterations

Stream-K Scheduling

Standard Scheduler schedules Output Tiles (M, N). The unit of work is "Computing one full output tile. With Stream-K Scheduler, we schedules Math Iterations where the unit of work is a specific number of MMA ops.

Stream K eliminates the idle time of 133rd tile by splitting the 133rd tile into 132 tiny pieces, so everyone finishes at the exact moment. The scheduler calculates the total math operations required for the entire problem and divides this work strictly and evenly across all available processing units.

It divides this total work strictly evenly by the number of available processing units. A Thread Block might compute a full tile, inherit a partially finished tile, or stop halfway through a tile when its assigned budget runs out

Hybrid Implementation: Targeting the Tail

Splitting tiles across the K-dimension introduces cross-block communication overhead. Applying Stream-K to the entire problem is often counterproductive.

The optimal implementation is a hybrid approach. The scheduler keeps the early waves purely data-parallel for maximum throughput.

The 50% Heuristic: Stream-K splitting is only applied to the final "tail" waves to balance the load. If the tail wave is already mostly full (e.g., >50%), the scheduler falls back to standard data-parallel execution to avoid unnecessary reduction overhead.

How does Stream K scheduling works

The scheduler calculates the total number of "math steps" in the remaining tiles.

$$\text{Total Work} = M_{tiles} \times N_{tiles} \times K_{tiles}$$

It divides this total work strictly evenly by the number of available SMs/Clusters

$$\text{Work per Cluster} = \frac{\text{Total Work}}{\text{Num Clusters}}$$

Each Cluster is assigned a specific range of this 1D tape. This results in 3 types of work. Most of the time, a Cluster's assigned range covers complete (M,N) tiles.

A Cluster might inherit a tile that the previous Cluster started but didn't finish.

A Cluster might run out of its assigned budget halfway through a tile. It computes the first half of the K-loop for this tile, saves the partial result, and stops.

The ”Fixup”: Peer-to-Peer Reduction

If Cluster A computes the first 50% of Tile X, and Cluster B computes the remaining 50% of Tile X, they need to sum their results before writing to global memory. This is called Fixup. There is a global memory buffer (scratchpad) allocated for these split tiles.

First half finishes its math. It writes its partial accumulator values into the Workspace and sets a flag (Barrier). Second half finishes its math. It checks the Workspace. If Cluster B sees Cluster A is done, Cluster B loads A's partial results, adds them to its own registers, and then writes the final sum to the destination matrix.

Backward Tile Iteration

When two Thread Blocks share a tile, the block computing the "end" of the K-dimension must wait for the block computing the "start" to finish its work before writing the final output

To minimize this idle waiting time, Stream-K workers iterate through their assigned output tiles in reverse K order

By working backward, the "ending" block computes its shared portion of the tile later in its execution sequence. By the time it is ready to combine the results, the "starting" block has already finished, drastically reducing barrier wait times.

Preserving L2 Cache Locality

Standard scheduling preserves L2 cache locality by clustering work geometrically. Stream-K's 1D tape approach naturally breaks this spatial locality, potentially causing cache thrashing.

To solve this, Stream-K workers are logically grouped together. The scheduler assigns units within the same group to process overlapping K-ranges across different output tiles. Because they are iterating through the exact same K-dimension slices simultaneously, their input reads overlap perfectly in the L2 cache.

Memory bound kernels

There are 3 major types of memory bound kernels:

- Bandwidth-bound: DRAM/L2 throughput near peak.

- Latency-bound: long scoreboard/memory wait dominates, bandwidth not saturated

- Locality-bound: poor L2 hit, cache/TLB thrash, indirection-heavy access.

Bias, activation, scaling, residual add, amax tracking all of these would be separate memory-bound kernels in a naive implementation. Instead they get folded into the GEMM epilogue in good kernels.

You never write a standalone bias+GELU kernel; you just compose kernels in the epilogue. One global memory round-trip instead of three.

The Big 3

Bandwidth-bound kernels: This is the bottleneck when you are already pushing near the maximum sustained bytes/second from HBM (or sometimes L2), so runtime is dominated by how many total bytes you move and how efficiently you move them

Latency bound kernels: Performance is limited mainly by how long a dependency takes to resolve (memory access, atomics, sync, long instruction chains), not by peak FLOPs or peak memory bandwidth.

Locality-bound kernels: This happens when you're moving plenty of bytes, but a big chunk is wasted due to poor reuse: low L2 hit rate, random/indirect accesses, cache/TLB thrash etc.

Epilogue

The epilogue is the final processing stage of a GEMM kernel. It occurs after the wgmma operations are complete in the registers, but before data is written back to Global Memory.

It transforms raw accumulation results into the final output format but we can also use this to do other memory-bound operations. The goal is to hide latency while performing necessary math (scaling, bias, activation) without spilling registers

The epilogue is logically structured as a nested loop that iterates over epilogue tiles. These tiles are subdivisions of the larger CTA output tile. Since the matrix accumulation results reside in registers after the GEMM mainloop, this outer shell manages the flow of data out of registers, through shared memory, and finally to global memory via TMA managing the granularity at which fusion operations are applied

Epilogue operations

1. alpha * Acc (scaled accumulator only), alpha * Acc + beta * C (linear combination)
2. Bias add: per-row bias per-column bias
3. Activations on top of lincomb/bias: ReLU GELU SiLU
4. Residual-style add paths (source added in residual form)
5. TopK + Softmax fusion (column-wise softmax variants)
6. Aux tensor ops: aux load (extra input tensor) aux store (extra output tensor)
7. Reductions fused in epilogue: row reduction column reduction scalar reduction
8. Absmax/amax tracking (common for FP8 paths)
9. Scaled fusion: scale factors for A/B/C/D scaling per row/column alpha-beta scaling variants
10. Block-scale-factor generation variants (for block-scaled workflows)

Sequence of operations in epilogue

1. Load all needed inputs for the tile: Acc, optional C, bias, scales, optional AuxIn.
2. Build base expression first: Z = alpha * Acc + beta * C (or just alpha * Acc if no C).
3. Add affine terms next: row/col bias, residual/additional linear terms.
4. Apply nonlinearity on Z: ReLU, GELU, SiLU, etc.
5. Apply output scaling/quantization (and collect amax if needed).
6. Materialize outputs:
   - main output D
   - optional AuxOut (pre-activation or post-activation, whichever you need)
   - optional row/col/scalar reductions
7. Final type convert + store.

Templates

1. Common forward fusion:

Z = alpha*Acc + beta*C-> Z+=bias -> Y = activation(Z) ->D=cast/scale(Y) (optional AuxOut)

2. Training-style with saved aux:

Z = alpha*Acc + beta*C + bias -> AuxOut=Z -> Y=activation(Z) -> D = Y(optional reductions)

3. Backward-style:

dY = alpha*Acc + beta*C -> dX = dActivation(dY, AuxIn) -> dBias = reduction(dX) -> D = dX

4. Softmax/TopK path (special case):

S = alpha*Acc + beta*C -> rowmax/rowsum/exp normalization -> optional TopK -> D (usually separate from standard ReLU/GELU chain)

The Orchestrator: Store phase

This phase starts when tensor-core WGMMA math is complete and the kernel transitions from compute to output processing. Physically, the epilogue owns the data path between accumulator registers and final global-memory writes.

The hardware now pivots from matrix-multiply issue slots to epilogue instructions that read accumulator registers, apply output transforms, and prepare store-ready values. This is the first moment where output precision and layout policy are enforced.

A warpgroup-level handoff occurs from “WGMMA producer” behavior to “epilogue consumer” behavior, typically guarded by internal ordering and barriers so no lane reads stale accumulator state. Once ordered, each lane can safely process its register-owned elements.

Epilogue Fusion Lifecycle

The SM90 epilogue isn't a single function call. It's a six-stage pipeline designed around one principle: separate what changes from what doesn't

begin() — One-time global initialization

begin_loop() — Per-tile setup

previsit() — Operand fetch

visit() — Arithmetic compute (the EVT traversal)

reduce(), postreduce(), tma_store() — Post-compute & store prep

end_loop(), end() — Finish & cleanup

`begin()` and `previsit()` - Staging Inputs

This phase stages non-accumulator operands (bias, scales, possibly residual data). The physical goal is to move auxiliaries from global memory into on-chip storage, then into registers right before use.

Auxiliary vectors are fetched via TMA-assisted movement into shared-memory buffers. Shared memory acts as the rendezvous point for reuse.

Lanes then load only their needed scalars/vectors from shared memory into registers. This keeps upcoming fused math register-resident and avoids redundant global reads.

A synchronization step confirms auxiliary buffers are fully populated before any lane consumes them. After this barrier, the epilogue can execute fused operations with predictable data availability.

Arithmetic Compute: `visit()`

This is the core fused-compute phase where scaling, bias application, activation, and clamping happen directly on accumulator values.

Each lane reads accumulator registers, combines them with staged auxiliary registers, and produces transformed outputs element by element. The data path remains entirely inside the register file and ALU/Tensor execution resources.

Once register results are finalized for the current subtile, lanes hand off to the store prep path rather than writing intermediates to shared memory. This preserves bandwidth for the actual commit path.

The main risk is register overuse; if fusion depth exceeds register budget, local-memory spills can erase the benefit of fusion.

Typical SM90 fusion pattern:

```
acc = alpha * acc          // Scale
acc = acc + (beta * C)     // Source addition
acc = acc + bias_vector    // Bias
acc = silu(acc)            // Activation
```

fragmentation is made tractable by explicit fragment->logical-coordinate mapping, then a staged reduction pipeline

(register -> shuffle -> smem -> gmem/atomic) depending on scope.

Post-Compute: `reduce()`

FP8 output introduces a unique requirement: you need the absolute maximum (amax) of the computed values to set the scaling factor.

Each thread computes its local amax across its fragment, Threads contribute to a shared memory reduction buffer, A single thread computes the final tile-wide amax, that amax is written to a global scaling factor pointer

Intra-warp reductions use warp shuffles so lanes exchange register values directly through the warp network without shared-memory traffic. This is the lowest-latency path for warp-scope collectives.

When reduction scope exceeds one warp, partials are staged in shared memory and combined by designated lanes or warps. The result is then broadcast back so all producers can apply consistent scaling or metadata.

Store Preparation: `postreduce()`

Register to Shared Memory (R→S)

Threads write computed fragments into a shared memory buffer using stmatrix instruction. This is more than a transfer it is a transformation:

Swizzling: Data is stored using a specific layout (like SmemLayoutD) to ensure the TMA sees contiguous, aligned blocks, regardless of the logical matrix orientation.

Precision Casting: This is the final opportunity to convert high-precision float accumulators into fp16 storage formats.

The Handoff Boundary

Shared memory acts as the definitive handoff point. Once the data is staged, the CUDA cores are released from the pipeline, and the TMA hardware independently manages the asynchronous movement to global memory.

TMA store

The consumer warps hand off to the TMA engine.

The sequence:

`fence.proxy.async` — cross proxy fence to make sure TMA can view the data.

`producer_commit` — Signals the StorePipeline that a stage is ready

One leader thread issues `cp.async.bulk.tensor`

TMA takes over: Smem ⟶ Global Memory, asynchronously

After this we cleanup per-tile state, Advance pipeline barriers for the next tile iteration, If using persistent kernels, loop back to begin_loop() with the next tile coordinates

Full pipeline

`begin()` → Load alpha, beta, resolve predicates. Once.

`begin_loop()` → Rebind to tile (m,n,l). Per tile.

`previsit()` → Fetch operands. Hide latency.

`visit()` → EVT math in registers. The hot path.

`reduce()` → FP8 amax reduction. Conditional.

`postreduce()` → Stage results: registers → smem.

`tma_store()` → Hardware moves smem → gmem. Async.

`end_loop()` → Advance pipeline. Next tile.

`end()` → Final flush and cleanup.

Code and illustrations