Stanford CS336 · Lecture 5

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: Why GPUs? Compute Scaling, Throughput Architecture, and the Memory Wall

Language model quality scales predictably with compute: scaling laws show that loss drops steadily as more floating-point operations (FLOPs) are applied to training. For roughly two decades (1980–2000s), single-thread processor performance grew automatically through Dennard scaling, where shrinking transistors allowed higher clock frequencies at constant power density. When Dennard scaling ended, single-thread speed stopped climbing, forcing hardware designers to shift from making one thread faster to executing thousands of operations in parallel. Over a ten-year period, parallel GPU compute scaled by more than $1000\times$. That $> 1000\times$ gain came from four distinct multiplicative factors rather than raw clock speed:

- **Number representation** ($\sim 16\times$): moving from wide 32-bit formats down to lower-precision formats tailored to machine learning.
- **Complex matrix instructions** ($\sim 12.5\times$): replacing scalar multiply-adds with dedicated matrix-multiply hardware.
- **Process shrinks** ($\sim 2.5\times$): fitting more transistors onto the silicon die.
- **Sparsity** ($\sim 2\times$): exploiting structured zeros in matrix operands.

Notice the product of these factors: $16 \times 12.5 \times 2.5 \times 2 = 1000\times$. Throughout this lecture, our target hardware is **NVIDIA Blackwell (B200 and B300, the SM100 family)**. Earlier generations—such as Ampere and Hopper—appear only as historical baselines to show why specific architectural features were introduced. Our objective is to make GPU performance predictable by connecting algorithms directly to hardware bottlenecks.

CPU vs. GPU: Latency Optimization vs. Throughput Optimization

Understanding why a GPU executes neural networks orders of magnitude faster than a CPU starts with a die-area trade-off. Silicon area on a chip is fixed, and CPUs and GPUs allocate that area to solve opposite problems.

In a **CPU** (latency-optimized architecture), most of the die is devoted to a large control unit (for branch prediction and out-of-order execution) and large on-chip caches, leaving room for only a small number of arithmetic logic units (ALUs) connected to off-chip DRAM. The CPU's goal is to finish a few complex, branch-heavy threads as fast as possible. Consider a single thread timeline for a CPU: Thread T1 runs a long block of compute, hits a brief stall waiting on memory (kept short by the large caches), resumes computing, hits another brief memory stall, and continues. Every piece of silicon exists to minimize single-thread latency.

In a **GPU** (throughput-optimized architecture), the die layout flips completely. Instead of one huge control unit and giant caches, the die contains many small control and cache strips feeding a massive grid of compute units, backed by **High-Bandwidth Memory (HBM)** rather than standard DRAM. A single GPU thread actually runs slower and stalls much longer on memory than a CPU thread. Look at the GPU execution timeline across four groups of threads labeled W1, W2, W3, and W4: W1 computes briefly and then enters a long white stall waiting on memory; immediately, W2 steps in and computes, then stalls; W3 steps in and computes, then stalls; W4 steps in and computes. By the time W4 stalls, W1's memory request has returned and W1 computes again. The GPU makes no attempt to minimize any individual thread's stall time; instead, it runs thousands of lightweight threads in parallel and hides memory stalls by switching to ready work.

> **COMMON MISCONCEPTION**
> You might think a GPU is fast because each of its cores has a higher clock speed or lower memory latency than a CPU core. That breaks immediately when you examine the hardware: a single GPU thread suffers much longer memory stalls and has far weaker control logic than a CPU thread. If you give a GPU only a handful of serial, branch-heavy threads, it will run dramatically slower than a CPU because it has no other threads ready to cover those stalls.

GPU Anatomy and the Memory Hierarchy

To predict where a GPU program spends its time, you need a concrete physical map of the chip. On a **Blackwell GPU (B200/B300)**, the die contains **148 Streaming Multiprocessors (SMs)** arranged across the chip. Beneath all 148 SMs sits an **on-die L2 cache** shared across every SM. Sitting off the GPU die—packaged right next to it as stacked DRAM—is **HBM (Global Memory)**. Beyond the GPU package, connected over PCIe or NVLink, sits **Host CPU RAM**.

> **DEFINITION**
> A **Streaming Multiprocessor (SM)** is an independent processing unit on a GPU containing its own warp schedulers, register file, scalar and matrix compute units, and fast on-chip SRAM; a Blackwell B200/B300 GPU contains 148 SMs operating in parallel.

Zooming inside a single Blackwell SM reveals six tightly coupled components:

1. **4 Warp Schedulers**: hardware dispatchers that select ready groups of threads and issue instructions every cycle.
2. **Register File**: $65{,}536 \times 32$-bit registers per SM, providing the fastest private storage for active threads.
3. **CUDA Cores**: general-purpose arithmetic units executing FP32, integer (INT), and special-function unit (SFU) operations.
4. **Tensor Cores**: specialized matrix multiply-accumulate (MMA) engines.
5. **Tensor Memory (TMEM)**: a new dedicated on-SM memory introduced on Blackwell SM100 specifically to hold Tensor Core accumulators. We'll need this when we get to Blackwell's asynchronous execution model in Section 6.

6. **Shared Memory / L1 Cache**: fast on-chip SRAM—configurable up to **227 KB per block on B300**—paired with the **TMA (Tensor Memory Accelerator)** hardware copy engine.

Across this hierarchy, one physical rule governs everything: **closer means faster and smaller**. Registers, TMEM, and Shared Memory/L1 sit directly inside each SM and are built from on-chip **SRAM**. HBM sits off die and is built from **DRAM**. On-chip SRAM is roughly $100\times$ **more expensive per byte** of silicon than DRAM, which is why its capacity is measured in kilobytes per SM rather than gigabytes, yet it is roughly $8\times$ **faster** than off-die HBM. Historical cycle-count measurements from the Ampere architecture illustrate the order-of-magnitude latency gap across these levels:

Memory Level (Ampere reference)	Latency (cycles)
Shared memory (load / store)	23 / 19
L1 cache	33
L2 cache	200
Global memory (HBM)	290

A load from HBM takes more than $12\times$ as many cycles as a load from shared memory inside the SM.

The Memory Wall and Specialized Matmul Hardware

The gap between compute and memory is not static; over the past two decades it has widened by orders of magnitude. Plotting normalized hardware scaling on a logarithmic scale from 1996 through 2023 and beyond reveals three wildly diverging straight lines:

- **Hardware FLOPs** grew by $60{,}000\times$ (roughly $3.0\times$ every 2 years).
- **DRAM bandwidth** grew by only $100\times$ (roughly $1.6\times$ every 2 years).
- **Interconnect bandwidth** grew by only $30\times$ (roughly $1.4\times$ every 2 years).

> **DEFINITION**
> The **memory wall** is the widening divergence between hardware arithmetic throughput and memory bandwidth—over 20+ years, GPU FLOPs scaled $60{,}000\times$ while DRAM bandwidth scaled only $100\times$, making data movement rather than arithmetic the dominant bottleneck for most workloads.

A second divergence occurred *within* compute itself. Plotting peak TFLOP/s on a log scale across GPU generations—from K80 to P100, V100, A100, and H100 onward—shows a single line before Volta (K80 to P100) that suddenly splits into two steep branches starting with Volta (V100), which introduced **Tensor Cores**:

- The upper branch, **Tensor Core Matmul**, climbs steeply generation after generation.
- The lower branch, **Non-Matmul FP Ops** (scalar floating-point arithmetic on CUDA Cores), rises far more slowly.

The vertical gap between the two branches on modern GPUs exceeds $10\times$: matrix multiplications execute more than an order of magnitude faster than ordinary scalar floating-point operations.

DEFINITION

Tensor Cores are dedicated hardware units inside each SM that execute fused matrix multiply-accumulate (MMA) instructions in hardware, achieving $> 10\times$ higher floating-point throughput than scalar CUDA Core operations.

WORKED EXAMPLE

Quantifying the Memory Wall and Latency Gap

Setup: Suppose an algorithm performs 1 scalar floating-point operation per value fetched from memory. Compare (a) the historical 20-year shift in compute-to-bandwidth balance, and (b) the cycle latency ratio between fetching data from HBM versus on-SM shared memory using the Ampere reference latencies.

Steps:

1. Compute-to-bandwidth divergence:

$$\frac{\text{FLOPs scaling}}{\text{DRAM BW scaling}} = \frac{60{,}000\times}{100\times} = 600\times$$

   Relative to memory bandwidth, arithmetic hardware became $600\times$ more abundant over two decades.

2. Cycle latency ratio between global HBM and shared memory loads:

$$\frac{290 \text{ cycles}}{23 \text{ cycles}} \approx 12.6\times$$

Interpretation: If a kernel fetches an operand from HBM, uses it once, and writes it back, the compute units spend almost the entire time idle waiting on memory transfers. Fast machine learning algorithms therefore obey two rules: cast as much work as possible as dense matrix multiplications (to run on Tensor Cores), and ruthlessly minimize HBM movement and non-matmul scalar operations.

KEY IDEA

GPUs trade single-thread latency for massive parallel throughput, hiding long memory stalls by switching among thousands of threads. Because hardware FLOPs outpaced DRAM bandwidth by $600\times$ over two decades and Tensor Cores outpace scalar FP units by $> 10\times$, feeding the compute units—and minimizing HBM transfers and non-matmul ops—governs real-world performance.

With 148 SMs, thousands of compute lanes, and multiple memory tiers in front of us, a natural question arises: how does software actually express thousands of parallel threads, assign them to SMs, and determine how many threads can stay resident on an SM at the same time? That mapping is defined by

the CUDA execution model.

> **✎ Check Your Understanding**
>
> 1. **Yes or No (with justification)**: Suppose a workload consists of 100% scalar floating-point operations (such as elementwise exponentials and additions) with zero matrix multiplications. Will upgrading to a GPU with $2\times$ faster Tensor Cores double its execution speed?
>
>   **Answer:** No. Tensor Cores only accelerate matrix multiply-accumulate (MMA) operations; scalar floating-point operations execute on CUDA Cores (and SFUs), which sit on the lower non-matmul throughput branch and are over $10\times$ slower than Tensor Cores.
>
> 2. **Fill in the blank**: Over the past 20+ years, hardware FLOPs scaled by $60{,}000\times$ while DRAM bandwidth scaled by only _____, creating the phenomenon known as the _____.
>
>   **Answer:** $100\times$; memory wall.

Section 2: The CUDA Execution Model, Memory Scopes, and Occupancy

To exploit 148 SMs without rewriting code for every new GPU generation, software needs a programming hierarchy that decouples logical parallelism from physical SM count. CUDA provides a four-level execution hierarchy: **Threads**, **Warps**, **Blocks**, and **Grids**.

Grids, Blocks, Warps, and Threads

At the top sits the **Grid**, which represents a single kernel launch. A grid is divided into many **Thread Blocks**—also called **CTAs (Cooperative Thread Arrays)**—indexed as `Block (0,0)`, `Block (1,0)`, and so on. The hardware scheduler assigns blocks independently onto available SMs in any order. Because blocks are independent, the exact same compiled kernel runs seamlessly on a GPU with 108 SMs, 132 SMs, or 148 SMs simply by pulling blocks dynamically from the grid until none remain.

Inside one **Block** (resident on a single SM), all threads share one allocation of on-SM shared memory. For example, a block of 256 threads is subdivided by the hardware into 8 consecutive groups of 32 threads: Warp 0 holds threads 0–31, Warp 1 holds threads 32–63, and so on up to Warp 7 (threads 224–255).

At the hardware scheduling level, a **Warp** of 32 threads is the fundamental unit of execution. One warp scheduler issues **one instruction** to all 32 threads (T0 through T31) at once under the **SIMT (Single Instruction, Multiple Threads)** model. Each thread executes that same instruction using its own private registers and its own coordinates (`threadIdx` and `blockIdx`).

DEFINITION

A **warp** is a group of 32 consecutively numbered threads within a block that are scheduled together under **SIMT (Single Instruction, Multiple Threads)**, executing the same instruction concurrently on their own private registers.

DEFINITION

A **thread block** (or **CTA**, Cooperative Thread Array) is a group of threads placed together onto a single SM that share a private allocation of fast on-chip shared memory and can synchronize with one another; a **grid** is the full collection of independent blocks launched by a single kernel invocation.

WORKED EXAMPLE

Worked Example: 1D Vector Addition in CUDA

Setup: Add two length-n float arrays `a` and `b` into `c`, where $n = 1{,}000{,}000$, launching blocks of `block_size = 256` threads (8 warps per block).

```
__global__ void vec_add(const float *a, const float *b, float *c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

int block_size = 256;
int grid_size  = (n + block_size - 1) / block_size;   // ceiling division
vec_add<<<grid_size, block_size>>>(a, b, c, n);
```

Steps:

1. **Compute the grid size** using integer ceiling division:

$$\text{grid_size} = \left\lfloor \frac{1{,}000{,}000 + 256 - 1}{256} \right\rfloor = \left\lfloor \frac{1{,}000{,}255}{256} \right\rfloor = 3{,}907 \text{ blocks}$$

2. **Compute total launched threads**:

$$3{,}907 \times 256 = 1{,}000{,}192 \text{ threads}$$

3. **Map a specific thread to its global array index**: Thread 5 (`threadIdx.x = 5`) inside Block 3 (`blockIdx.x = 3`) with `blockDim.x = 256` computes:

$$\text{idx} = 3 \times 256 + 5 = 773$$

   It reads `a[773]` and `b[773]` and writes `c[773]` .

4. **Trace the tail block**: Block 0 covers indices `0–255`, Block 1 covers `256–511`, and the final block, Block 3906, covers indices $3{,}906 \times 256 = 999{,}936$ through $1{,}000{,}191$. The valid array ends at index $999{,}999$. Thus the final $1{,}000{,}192 - 1{,}000{,}000 = 192$ threads evaluate `idx < n` as false and skip memory access, preventing out-of-bounds corruption.
5. **Hardware execution**: On a B300 with 148 SMs, the hardware dynamically pulls blocks from the 3,907-block grid across all 148 SMs until every block completes. Notice also that consecutive threads (`idx`, `idx + 1`) access consecutive floats in memory—a property we will return to when analyzing memory coalescing in Section 4.

Interpretation: The formula `blockIdx.x * blockDim.x + threadIdx.x` maps a 2-level hierarchy (blocks and threads) cleanly onto a flat 1D array, while the ceiling grid size plus `if (idx < n)` tail guard handles arbitrary array lengths not divisible by 256.

Warps: Latency Hiding and Control Divergence

Because every resident warp on an SM keeps its entire architectural state live inside the SM's physical register file, switching from one warp to another takes **zero cycles**—there is no context-save or context-restore overhead. Examine a timeline of a single warp scheduler managing three warps (Warp 0, Warp 1, Warp 2):

- Warp 0 issues a global memory load (`LD`) and immediately stalls for roughly 290 cycles waiting on HBM.
- On the very next cycle, the scheduler switches to Warp 1, which issues a fused multiply-add (`FMA`) before waiting on its own load.
- Immediately after, the scheduler switches to Warp 2, which issues back-to-back `FMA` instructions.
- By the time Warp 0's data returns from HBM, Warp 0 is scheduled again to run its `FMA`, and Warp 1 executes its store (`ST`).

Because other resident warps were ready to issue, the SM stayed busy every cycle and the 290-cycle stall was completely hidden.

However, SIMT execution introduces a distinct compute hazard inside a single warp called **control divergence**. Suppose all 32 threads in one warp reach a branch:

```
if (tid < 4) {
    A; B;
} else {
    X; Y;
}
Z;
```

Because one warp scheduler issues only one instruction per cycle to the entire warp, the hardware cannot execute `A` on threads T0–T3 while simultaneously executing `X` on threads T4–T31. Instead, the warp serializes both paths:

1. First, the warp executes `A` and `B` with threads T0–T3 active and threads T4–T31 **masked off** (idle).

2. Next, the warp executes `X` and `Y` with threads T4–T31 active and threads T0–T3 **masked off**.
3. Finally, all 32 threads reconverge and execute `Z` together.

DEFINITION

Control divergence is the serialization penalty that occurs when threads within the same 32-thread warp take different paths at a conditional branch, forcing the warp to execute every taken path sequentially while masking off inactive lanes.

COMMON MISCONCEPTION

You might think any `if` statement in a GPU kernel destroys performance, or that control divergence is a memory problem. Both are wrong. First, control divergence is a *compute* phenomenon, not a memory phenomenon. Second, branches are cheap whenever all 32 threads in a warp evaluate the condition identically—for example, in our `vec_add` kernel above, `if (idx < n)` is true for all 32 threads in every warp of Blocks 0 through 3905, and diverges in at most one warp inside the single final tail block (Block 3906).

CUDA Memory Model and Synchronization

The execution hierarchy maps directly onto a strict memory scope hierarchy, which dictates which threads can see which data and how they synchronize:

- **Per-thread scope**: Fast private **registers**. If a thread requires more registers than are available, the compiler spills excess variables to slow **local memory**, which physically resides in off-chip HBM.
- **Per-block scope**: Fast on-SM **shared memory (SMEM)**, visible to all threads within the same block (e.g., all threads in Block `(0,0)` share Block `(0,0)`'s SMEM, completely isolated from Block `(1,0)`'s SMEM).
- **Per-grid scope**: Slow off-chip **global memory (HBM)**, readable and writable by every block in the grid and accessible from the Host, alongside read-only **constant memory** per grid.

Synchronization follows these exact boundaries. **Within a block**, `__syncthreads()` provides a fast hardware barrier: every thread in the block waits until all other threads in the block reach the barrier, guaranteeing that shared memory writes are safely visible before reads begin. **Across blocks**, however, there is **no cheap global barrier**. Because blocks are scheduled independently onto SMs, two blocks in a large grid may not even be resident on the GPU at the same time—if Block 0 waited for Block 3000 to arrive at a barrier while Block 3000 was waiting for Block 0 to finish and free an SM, the GPU would deadlock. Consequently, standard cross-block data exchange goes through global HBM, typically by ending one kernel launch and starting a new kernel launch. (Later, in Section 6, we will see how Hopper and Blackwell extend this model with `mbarrier` for asynchronous within-block handoffs and **thread block clusters** for direct sharing across neighboring SMs.)

Occupancy: Hardware Budgets and When Maximum Occupancy Hurts

To hide latency via zero-cost warp switching, an SM must hold multiple active warps simultaneously.

DEFINITION

Occupancy is the ratio of active warps currently resident on an SM to the maximum number of warps that the SM hardware can hold simultaneously.

The number of resident thread blocks that can fit concurrently on a single SM is capped by the tightest of three independent per-SM resource budgets:

$$\text{blocks/SM} = \min\left(\left\lfloor \frac{\text{SMEM}_{\text{SM}}}{\text{SMEM}_{\text{block}}} \right\rfloor, \left\lfloor \frac{\text{Regs}_{\text{SM}}}{\text{Regs}_{\text{thread}} \cdot T_{\text{block}}} \right\rfloor, \left\lfloor \frac{W_{\max}}{T_{\text{block}}/32} \right\rfloor\right)$$

Let us break this equation apart term by term:

- SMEM_{SM} is the SM's shared memory capacity, and $\text{SMEM}_{\text{block}}$ is the shared memory requested by one block. Their integer floor $\lfloor \text{SMEM}_{\text{SM}}/\text{SMEM}_{\text{block}} \rfloor$ gives the maximum number of whole blocks that fit in shared memory.
- Regs_{SM} is the SM's total 32-bit register file ($65{,}536$ on B300). Inside the denominator, $\text{Regs}_{\text{thread}}$ is the register count per thread and T_{block} is the number of threads per block, so their product $\text{Regs}_{\text{thread}} \cdot T_{\text{block}}$ is the total register footprint of one block. The floor $\lfloor \text{Regs}_{\text{SM}}/(\text{Regs}_{\text{thread}} \cdot T_{\text{block}}) \rfloor$ gives the register-limited block cap.
- $W_{\max}$ is the SM's maximum resident warp capacity, and $T_{\text{block}}/32$ is the number of warps in one block (since every warp has 32 threads). The floor $\lfloor W_{\max}/(T_{\text{block}}/32) \rfloor$ gives the warp-slot cap.
- Taking the $\min(\cdot)$ across all three terms ensures every hardware constraint is satisfied simultaneously. As a whole, the equation tells you how many concurrent blocks—and therefore how many resident warps, $\text{blocks/SM} \times (T_{\text{block}}/32)$—an SM can keep alive to hide stalls.

WORKED EXAMPLE

Worked Example: Computing Resident Blocks per SM on Blackwell B300

Setup: A Blackwell B300 has 148 SMs, up to 227 KB of SMEM/block, and $65{,}536 \times 32$-bit registers per SM. We launch blocks of $T_{\text{block}} = 256$ threads ($256/32 = 8$ warps per block), where each thread uses $\text{Regs}_{\text{thread}} = 128$ registers. Compare two shared-memory allocations: (a) 96 KB per block, and (b) 160 KB per block.

Steps:

1. Compute registers required per block:

$$\text{Regs}_{\text{block}} = 128 \text{ regs/thread} \times 256 \text{ threads/block} = 32{,}768 \text{ regs/block}$$

The register budget allows:

$$\left\lfloor \frac{65{,}536}{32{,}768} \right\rfloor = 2 \text{ blocks/SM}$$

2. **Case (a): 96 KB SMEM/block**:

$$\left\lfloor \frac{227\ \text{KB}}{96\ \text{KB}} \right\rfloor = \lfloor 2.36 \rfloor = 2\ \text{blocks/SM}$$

Taking the minimum with the register limit (the warp-slot cap is not binding here):

$$\text{blocks/SM} = \min(2, 2) = 2\ \text{blocks/SM}$$

This puts $2 \times 8 = 16$ warps resident per SM, and $2 \times 148 = 296$ blocks concurrently active across the entire GPU.

3. **Case (b): 160 KB SMEM/block**:

$$\left\lfloor \frac{227\ \text{KB}}{160\ \text{KB}} \right\rfloor = \lfloor 1.41 \rfloor = 1\ \text{block/SM}$$

Now shared memory becomes the tightest constraint:

$$\text{blocks/SM} = \min(1, 2) = 1\ \text{block/SM}$$

Occupancy drops by half to $1 \times 8 = 8$ warps/SM, with $1 \times 148 = 148$ blocks active across the GPU.

Interpretation: Increasing per-block SMEM from 96 KB to 160 KB cuts resident occupancy in half. Whether that hurts or helps performance depends entirely on what kind of kernel you are running.

COMMON MISCONCEPTION

You might think that 100% occupancy is always the goal when tuning a CUDA kernel, and that Case (b) above must be slower than Case (a). That breaks down completely for Tensor Core workloads. Memory-bound elementwise kernels do want high occupancy so the scheduler always has ready warps to hide HBM latency. By contrast, large matrix multiplication (GEMM) and attention kernels intentionally run *few, large blocks* (often just 1 block per SM) so that each block claims maximum SMEM, registers, and TMEM for data reuse, hiding latency inside large asynchronous tiles rather than relying on dozens of tiny resident warps.

KEY IDEA

CUDA organizes parallel work into Grids of independent Blocks, which hardware executes as 32-thread SIMT Warps with zero-cost context switching and fast per-block `__syncthreads()` barriers. Resident occupancy is bounded by the tightest of SMEM, registers, and warp slots—and while high occupancy helps memory-bound kernels hide HBM latency, Tensor Core kernels deliberately trade occupancy for larger on-chip tiles and data reuse.

We now understand how threads, warps, and blocks execute on an SM. Yet knowing how many warps are active still does not tell us whether a given workload is limited by arithmetic throughput or by memory bandwidth, nor how to systematically speed it up. To diagnose bottlenecks quantitatively, we need the roofline model.

✎ Check Your Understanding

1. **Quick Computation**: You launch a 1D CUDA kernel on an array of $n = 1{,}000$ elements with `block_size = 128`. How many blocks are in the grid, and how many launched threads will be skipped by the `if (idx < n)` guard?

   Answer: $\text{grid_size} = \lfloor (1{,}000 + 127)/128 \rfloor = 8$ blocks, launching $8 \times 128 = 1{,}024$ threads; $1{,}024 - 1{,}000 = 24$ threads in the last block are skipped by the guard.

2. **What would happen if…**: In the B300 occupancy worked example (Case a: 256 threads/block, 96 KB SMEM/block), a compiler change increases register usage from 128 to 140 registers per thread. How many blocks/SM can now fit?

   Answer: Each block now needs $140 \times 256 = 35{,}840$ registers, so $\lfloor 65{,}536/35{,}840 \rfloor = 1$ block/SM; resident occupancy drops from 2 blocks/SM (16 warps) to 1 block/SM (8 warps).

Section 3: The Roofline Model and Data-Movement Levers

Before writing a single optimization, you need to know *why* a kernel is slow. Optimizing arithmetic in a kernel that is starved waiting on HBM yields zero speedup; conversely, cutting memory traffic in a kernel whose Tensor Cores are already saturated yields zero speedup. The **roofline model** turns this distinction into a clean, predictive boundary.

The Roofline Model

Every kernel performs some total number of floating-point operations and transfers some total number of bytes across a memory interface (such as HBM). The ratio of those two quantities defines the workload's **arithmetic intensity** (also called **operational intensity**):

$$\text{Intensity} = \frac{\text{FLOPs performed}}{\text{Bytes transferred}}$$

DEFINITION

Arithmetic intensity (or **operational intensity**) is the ratio of floating-point operations executed by a kernel to the number of bytes moved from memory (measured in FLOPs/Byte); it determines whether a workload is constrained by memory bandwidth or by peak compute throughput.

Given a hardware peak compute rate (Peak FLOP/s) and a memory bandwidth (BW, in Bytes/s), the maximum throughput a kernel can achieve is bounded by the **roofline equation**:

$$\text{FLOP/s} \leq \min\big(\text{Peak FLOP/s},\ \text{Intensity} \times \text{BW}\big)$$

Let us dissect each term:

- Peak FLOP/s is the hard ceiling imposed by the compute units (for example, Peak Tensor Core FLOP/s). No matter how high your arithmetic intensity climbs, throughput cannot exceed this horizontal ceiling.
- $\text{Intensity} \times \text{BW}$ has units of $(\text{FLOPs/Byte}) \times (\text{Bytes/s}) = \text{FLOP/s}$. It represents the maximum compute rate that the memory bus can sustain if every byte delivered by memory triggers Intensity operations.
- Taking $\min(\cdot)$ gives the achievable performance envelope: whichever limit is lower dictates the bottleneck.

Visualizing this equation on a **log-log plot** produces the classic roofline diagram. The horizontal axis is **Arithmetic / Operational Intensity (FLOPs/Byte, log scale)** and the vertical axis is **Throughput (FLOP/s, log scale)**. Because $\log(\text{Intensity} \times \text{BW}) = \log(\text{Intensity}) + \log(\text{BW})$, each memory level appears as a straight line rising at a 45° slope until it intersects the horizontal **Peak Tensor Core FLOP/s** ceiling:

- On the far left sits the steepest/leftmost sloped line for **Registers** (highest bandwidth, hitting the compute roof at very low intensity).
- To its right sits the sloped line for **Shared Memory BW**.
- Further right sits the **HBM Bandwidth Roof**. Where the HBM bandwidth slope meets the horizontal Peak Tensor Core ceiling defines a vertical dashed **ridge line**.
- Everything to the left of the HBM ridge line is **Memory-Bound**: here sit elementwise operations and small-batch LLM decode, pinned down along the HBM slope far below peak compute.
- Everything to the right of the ridge line is **Compute-Bound**: here sit large dense matrix multiplications (GEMMs), operating right near the Peak Tensor Core roof.

The central question of GPU optimization is: **how do we push a workload rightward across the ridge line so it avoids being memory-bound?** There are **five core levers**, every one of which either moves fewer bytes from HBM or increases the compute performed per byte fetched:

1. Low-precision math
2. Operator fusion
3. Recomputation
4. Memory coalescing and bank layout
5. Tiling (plus wave scheduling)

We cover Levers 1 through 3 in this section, Lever 4 in Section 4, and Lever 5 in Section 5.

Lever 1: Low Precision and Mixed-Precision Tensor Cores

The most direct way to move fewer bytes from HBM is to represent each number with fewer bits. At the same time, Tensor Core hardware doubles or quadruples peak throughput as operand bit-width shrinks.

> **WORKED EXAMPLE**
> **Worked Example: Effect of Precision on Elementwise ReLU Intensity**
>
> *Setup*: Compute $\mathrm{ReLU}(x) = \max(0, x)$ over an array of length n. Each element requires 1 read from HBM, 1 write back to HBM, and 1 comparison FLOP.
>
> *Steps*:
>
> 1. **In FP32 (4 bytes/element)**:
>   - Bytes transferred per element: $4\ \mathrm{B}\ (\mathrm{read}) + 4\ \mathrm{B}\ (\mathrm{write}) = 8\ \mathrm{B}$.
>   - Arithmetic intensity: $\frac{1\ \mathrm{FLOP}}{8\ \mathrm{B}} = 0.125\ \mathrm{FLOP/B}$ (equivalently, 8 B/FLOP).
> 2. **In FP16 or BF16 (2 bytes/element)**:
>   - Bytes transferred per element: $2\ \mathrm{B}\ (\mathrm{read}) + 2\ \mathrm{B}\ (\mathrm{write}) = 4\ \mathrm{B}$.
>   - Arithmetic intensity: $\frac{1\ \mathrm{FLOP}}{4\ \mathrm{B}} = 0.25\ \mathrm{FLOP/B}$ (4 B/FLOP).
>
> *Interpretation*: Simply halving the representation from 32 bits to 16 bits doubles the arithmetic intensity from 0.125 to 0.25 FLOP/B and cuts HBM traffic in half, doubling execution speed on the HBM-bound slope of the roofline.

Not every operation tolerates aggressive bit reduction, however. Precision sensitivity in machine learning follows three practical rules:

- **Low precision (FP16, BF16, FP8, FP4)**: suitable for matrix multiplications and most pointwise operations.
- **Higher precision (FP32)**: required for reductions (sums, softmax denominators, normalization) to prevent rounding drift as thousands of small terms accumulate.
- **Higher dynamic range (FP32 or BF16)**: required for transcendental functions like `exp`, `log`, `pow`, and loss functions, which easily overflow or underflow in narrow exponent formats.

Modern **Mixed-Precision Tensor Cores** reconcile fast low-bit storage with accurate accumulation directly in hardware. In the mixed-precision Tensor Core datapath, low-bit input tiles A and B enter a hardware multiplier; the resulting products are summed together with prior partial products inside an **FP32 Accumulator** (which resides in **TMEM** on Blackwell SM100) and emitted in FP32. You get the bandwidth and throughput benefits of low-bit inputs without suffering low-bit rounding collapse during the inner-product sum.

Floating-Point Bit Layouts: FP16, BF16, FP8, and MXFP8

Every IEEE-style floating-point format splits its bits into three fields: **Sign** (1 bit), **Exponent** (controls dynamic range), and **Mantissa** (controls relative precision). Comparing the bit layouts side by side reveals the exact range-versus-precision trade-off:

- **FP16 (16 bits)**: 1 sign + 5 exponent + 10 mantissa. High precision, but a narrow exponent range that overflows easily during training.
- **BF16 (16 bits)**: 1 sign + 8 exponent + 7 mantissa. Sacrifices 3 mantissa bits compared to FP16 in order to keep the exact same 8-bit exponent range as FP32.
- **FP8 E4M3 (8 bits)**: 1 sign + 4 exponent + 3 mantissa. More precision, narrower range.
- **FP8 E5M2 (8 bits)**: 1 sign + 5 exponent + 2 mantissa. Wider range, coarser precision.

When shrinking to 8 bits, a single **Per-Tensor FP8** format—where the entire tensor shares just **1 FP32 scale factor**—struggles whenever one outlier forces the whole tensor's scale up, pushing smaller values to zero. **Blackwell's MXFP8 (Microscaling FP8)** solves this by replacing one global scale with fine-grained block scales: instead of 1 scale per tensor, MXFP8 attaches **1 E8M0 scale factor (an 8-bit pure power-of-two exponent with 0 mantissa bits) to every block of 32 values along the inner reduction axis**. Because those local E8M0 block scales absorb dynamic range variations across the tensor, MXFP8 stores the actual data elements in **E4M3** (preserving the extra mantissa bit rather than falling back to E5M2).

> **DEFINITION**
> **MXFP8** is a microscaled 8-bit floating-point format supported on Blackwell that pairs blocks of 32 **E4M3** data elements along the inner reduction axis with a shared 8-bit **E8M0** power-of-two scale factor, delegating dynamic range to the block scale while keeping mantissa precision in the elements.

Fine-grained axis-aligned scaling creates a crucial **training implication: transposes are nontrivial**. In a linear layer's forward pass (**FPROP**), activations and weights are scaled rowwise along the contraction axis. In the backward pass (**DGRAD** for activation gradients and **WGRAD** for weight gradients), those same matrices must be multiplied in transposed orientation, which changes which dimension is the inner reduction axis. Because each E8M0 scale covers 32 elements along the *old* inner axis, you cannot simply swap row and column indices in memory—you must **transpose and re-quantize** along the new reduction axis. Meanwhile, norms, softmax, and master weights remain in BF16/FP32.

4-Bit Microscaling on Blackwell: MXFP4 vs. NVFP4

Pushing down to 4 bits (**FP4, E2M1**: 1 sign, 2 exponent, 1 mantissa) leaves only **16 representable values** on the entire number line:

$$\pm\{0.0,\ 0.5,\ 1.0,\ 1.5,\ 2.0,\ 3.0,\ 4.0,\ 6.0\}$$

With only 8 non-negative magnitudes spaced between 0.0 and 6.0, block scaling becomes mandatory—and Blackwell supports two distinct 4-bit microscaling formats that must not be confused:

Property	MXFP4 (OCP Standard)	NVFP4 (NVIDIA Blackwell)
Block size	32 values	**16 values**
Block scale format	E8M0 (power of 2)	**E4M3** (has mantissa bits)
Second-level tensor scale	None	**FP32 per-tensor scale**

DEFINITION

MXFP4 (the OCP standard) groups 32 FP4 (E2M1) values under a single E8M0 power-of-two scale with no tensor-level scale, whereas **NVFP4** (NVIDIA Blackwell's format) groups 16 FP4 values under an E4M3 block scale (with mantissa bits) combined with a second-level FP32 per-tensor scale.

NVFP4 preserves noticeably more accuracy at 4 bits for two reasons: cutting the block size from 32 to 16 isolates large outlier values so they corrupt half as many neighbors, and using E4M3 instead of E8M0 for the block scale avoids rounding every scale factor up to a coarse power of two (with the per-tensor FP32 scale extending overall dynamic range). Both 4-bit paradigms are already in 2026 production frontier models: **Kimi K3** trains its Mixture-of-Experts (MoE) experts with **MXFP4 quantization-aware training (QAT)**, and **DeepSeek-V4.1-Flash** stores its global KV cache in **FP4** (requiring just 890 bytes per token).

COMMON MISCONCEPTION

You might think MXFP4 and NVFP4 are interchangeable names for the same 4-bit Blackwell format. That breaks down on every structural parameter: 16-element blocks with E4M3 block scales and an FP32 tensor scale specifically define **NVFP4**, whereas 32-element blocks with E8M0 power-of-two scales define **MXFP4**.

WORKED EXAMPLE

Worked Example: Why E4M3 Scales Beat Power-of-Two E8M0 Scales in 4-Bit Blocks

Setup: Suppose a group of 16 values has a maximum magnitude of 9.0, and we want to map the block into the FP4 range $[-6.0, +6.0]$ so the largest value lands near 6.0 (ideal scale factor $s_{\text{ideal}} = 9.0/6.0 = 1.5$). Compare how E8M0 (MXFP4) and E4M3 (NVFP4) encode this block scale.

Steps:

1. **E8M0 scale (MXFP4)**: E8M0 can only represent exact powers of two ($\ldots, 0.5, 1.0, 2.0, 4.0, \ldots$). To avoid clipping 9.0, it must round 1.5 up to $s_{\text{E8M0}} = 2.0$. After dividing by 2.0, the maximum value maps to $9.0/2.0 = 4.5$, leaving the entire $[4.5, 6.0]$ upper range unused—throwing away a quarter of the already tiny 16-value grid.
2. **E4M3 scale (NVFP4)**: Because E4M3 includes 3 mantissa bits, it represents $1.5 = 1.5 \times 2^0$ *exactly* ($s_{\text{E4M3}} = 1.5$). Dividing by 1.5 maps 9.0 directly to 6.0, using the full dynamic range of the 4-bit FP4 values.

> *Interpretation*: At 4 bits, losing even one or two representable bins to coarse power-of-two scale rounding visibly degrades accuracy; E4M3 block scales plus 16-element granularity explain why NVFP4 retains tighter numerical fidelity.

Lever 2: Operator Fusion and Kernel Launch Overhead

Low precision shrinks the size of each transfer, but naive execution still makes far too many trips to HBM. Consider computing the elementwise expression $\sin^2 x + \cos^2 x$ on an array x.

In a **naive (unfused) implementation**, the framework executes 5 separate CUDA kernels:

1. `sin(x)` reads x from HBM, computes $\sin(x)$, and writes it back to HBM.
2. `pow2` reads $\sin(x)$ from HBM, squares it, and writes $\sin^2(x)$ to HBM.
3. `cos(x)` reads x from HBM, computes $\cos(x)$, and writes it back to HBM.
4. `pow2` reads $\cos(x)$ from HBM, squares it, and writes $\cos^2(x)$ to HBM.
5. `add` reads both $\sin^2(x)$ and $\cos^2(x)$ from HBM and writes the sum to HBM.

This incurs **5 kernel launches and 5 full HBM round-trips**—shipping intermediate tensors back and forth between the slow HBM warehouse and the fast SM factory just to perform trivial arithmetic.

In a **fused implementation**, a single CUDA kernel reads x from HBM once into registers, computes `sin(x)` $\rightarrow$ `pow2`, computes `cos(x)` $\rightarrow$ `pow2`, adds the two register values, and writes the final output to HBM once: **1 launch, 1 read, and 1 write**.

DEFINITION

Operator fusion combines a sequence of consecutive operations into a single GPU kernel so that intermediate results stay in fast on-SM registers or shared memory instead of being written to and re-read from HBM between separate kernel launches.

WORKED EXAMPLE

Worked Example: HBM Bandwidth Savings from Fusing $\sin^2 x + \cos^2 x$

Setup: Evaluate $y = \sin^2(x) + \cos^2(x)$ on an FP32 vector of length $n = 10{,}000{,}000$ (40 MB per vector).

Steps:

1. **Unfused (5 kernels)**:
   - `sin`: 1 read (x) + 1 write (t_1) = 2 vector transfers (80 MB)
   - `pow2`: 1 read (t_1) + 1 write (t_2) = 2 transfers (80 MB)
   - `cos`: 1 read (x) + 1 write (t_3) = 2 transfers (80 MB)
   - `pow2`: 1 read (t_3) + 1 write (t_4) = 2 transfers (80 MB)
   - `add`: 2 reads (t_2, t_4) + 1 write (y) = 3 transfers (120 MB)
   - **Total unfused HBM traffic**: 11 vector transfers = 440 MB.

2. **Fused (1 kernel)**:
   - 1 read (x) + 1 write (y) = 2 vector transfers = $80\ \text{MB}$.
3. **Reduction factor**: $440\ \text{MB}/80\ \text{MB} = 5.5\times$ less HBM traffic—with the exact same floating-point work, raising arithmetic intensity by $5.5\times$.

Interpretation: Because pointwise chains are deeply memory-bound, fusing them removes almost all HBM movement and delivers a multi-fold speedup automatically.

Modern graph compilers like **`torch.compile` (TorchInductor)** automatically detect and fuse pointwise chains and reductions. Even when operations cannot be fused into one kernel, issuing many tiny kernels exposes **kernel launch overhead**: CPU-side launch latency leaves the GPU sitting idle in the gaps between kernels. **CUDA Graphs** address this by capturing an entire sequence of kernel launches once and replaying the whole sequence from the GPU side in a single call—acting as an architectural bridge between operator fusion and full **megakernels** (which we will examine in Section 8).

Lever 3: Recomputation (Activation Checkpointing)

Operator fusion eliminates HBM trips *within* a forward pass, yet backpropagation introduces another massive source of HBM traffic: standard automatic differentiation saves intermediate forward activations in HBM so the backward pass can read them later to compute gradients.

Consider a chain of three stacked `sigmoid` layers taking input x to produce `out`, with intermediate activations $s_1 = \text{sigmoid}(x)$ and $s_2 = \text{sigmoid}(s_1)$:

- **Standard (Save Activations)**:
  - Forward pass: reads x (1 read), writes s_1, s_2, and `out` to HBM (3 writes).
  - Backward pass: reads s_1, s_2, and $d\text{out}$ from HBM (3 reads), computes dx, and writes dx to HBM (1 write).
  - **Total: 8 HBM accesses**.
- **Recompute in Backward**:
  - Forward pass: reads x (1 read), computes all three sigmoids in registers, discards s_1 and s_2, and writes only `out` (1 write).
  - Backward pass: reads only x and $d\text{out}$ from HBM (2 reads), recomputes s_1 and s_2 on-chip in registers alongside the backward derivatives, and writes dx (1 write).
  - **Total: 5 HBM accesses**—cutting HBM traffic to $5/8$ of the standard pass, while also shrinking peak memory footprint.

DEFINITION

Recomputation (or **activation checkpointing**) deliberately discards selected intermediate activations during the forward pass and recomputes them on-chip during the backward pass, trading abundant compute FLOPs to save scarce HBM bandwidth and memory capacity.

COMMON MISCONCEPTION

You might think doing *more* floating-point operations during training must always make training slower. That breaks whenever the workload sits in the memory-bound region of the roofline: because compute units are already idle waiting on HBM, performing extra on-chip FLOPs to avoid writing and re-reading intermediates from HBM makes the overall step *faster*, not slower. We will see this exact paradox drive FlashAttention's backward pass in Section 7.

KEY IDEA

The roofline bound $\text{FLOP/s} \leq \min(\text{Peak FLOP/s},\ \text{Intensity} \times \text{BW})$ classifies every kernel as memory-bound or compute-bound. Low precision (including Blackwell's MXFP8, MXFP4, and NVFP4), operator fusion (plus CUDA Graphs), and backward recomputation all push kernels rightward by slashing HBM bytes moved per FLOP.

Reducing the logical number of bytes a kernel requests is only half the battle against the memory wall. If the physical addresses threads request do not match how DRAM bursts and shared-memory banks operate, hardware silently multiplies the actual traffic and serializes accesses by up to $32\times$. We examine those physical layout rules next.

✎ Check Your Understanding

1. **Quick Computation**: A fused kernel reads 1 BF16 input vector and writes 1 BF16 output vector (2 bytes/element each) while executing 12 FLOPs per element. What is its arithmetic intensity in FLOPs/Byte?

   Answer: Each element transfers $2 + 2 = 4$ bytes and performs 12 FLOPs, so its arithmetic intensity is $12/4 = 3.0\ \text{FLOPs/Byte}$.

2. **Yes or No (with justification)**: When training in MXFP8, can you reuse the exact same quantized activation tensor (with its existing E8M0 scales) directly as a transposed input during the backward pass without re-quantizing?

   Answer: No. MXFP8 attaches one E8M0 scale factor per 32 values along the inner reduction axis; transposing the matrix for DGRAD or WGRAD changes the reduction axis, so the tensor must be transposed and re-quantized along the new reduction axis.

Section 4: Memory Access Efficiency: Coalescing and Shared Memory Bank Conflicts

Even if you pick an optimal low-precision format and fuse your operations into a single kernel, poor memory access patterns can collapse effective bandwidth to a tiny fraction of peak hardware speed. Two distinct physical mechanisms govern access efficiency at the two most important levels of the GPU

memory hierarchy: **memory coalescing** for off-chip DRAM (HBM), and **bank conflicts** for on-chip shared memory (SRAM).

Lever 4A: Memory Coalescing and DRAM Burst Mode

DRAM does not fetch isolated 4-byte scalars efficiently. Under **DRAM burst mode**, every row-buffer read delivers an entire contiguous chunk of bytes called a **burst section**—in real GPUs, each burst section is at least **128 bytes** wide.

Suppose a memory array is partitioned into contiguous burst sections: Burst section 0 holds elements `0, 1, 2, 3`, Burst section 1 holds `4, 5, 6, 7`, and Burst section 2 holds `8, 9, 10, 11`. Now compare how a warp of threads accesses this memory:

- **Coalesced loads**: Threads T0, T1, T2, and T3 request consecutive elements `0, 1, 2, 3`, which all reside inside Burst section 0. The memory controller combines all of those thread requests into **a single DRAM burst transaction**. Every byte transferred across the HBM pins is used by a thread.
- **Uncoalesced (strided) loads**: Suppose a 2D matrix M is stored in **row-major order** (elements along a row are adjacent in memory), and threads T0, T1, T2, and T3 load `M[0,0]`, `M[1,0]`, `M[2,0]`, and `M[3,0]`—down a column. Because consecutive rows are separated by the matrix width, each thread lands in a *different* burst section. Hardware must issue **4 separate DRAM burst requests**, discarding almost all of the bytes fetched in each burst.

> **DEFINITION**
> **Memory coalescing** is the hardware merging of global memory accesses issued by the 32 threads of a warp into the minimum number of contiguous DRAM burst transactions (typically ≥ 128 bytes each), achieved when consecutively numbered threads access adjacent memory addresses.

Consider a visual comparison of two ways to assign threads to a row-major matrix:

- **Not coalesced**: Thread T0 marches left-to-right across row 0 while Thread T1 marches left-to-right across row 1. At every single step, the memory address accessed by T0 and the memory address accessed by T1 are separated by `stride = matrix width`, forcing separate burst transactions for every thread.
- **Coalesced**: Thread T0 owns column 0 and Thread T1 owns column 1, and both march downward together from row 0 to row 1 to row 2. At every single step, T0 and T1 read adjacent elements in the same row, landing cleanly inside the same DRAM burst.

> **WORKED EXAMPLE**
> **Worked Example: Bandwidth Cost of Uncoalesced Matrix Loads**
>
> *Setup*: A warp of 32 threads reads 32 FP32 values (4 bytes each, so $32 \times 4 = 128$ bytes of useful data) from a 1024×1024 row-major matrix, where each DRAM burst section is 128 bytes. Compare (a) reading across a row (coalesced) versus (b) reading down a column (strided).
>
> *Steps*:

1. **Row read (coalesced)**: Thread $t \in \{0, \ldots, 31\}$ accesses `M[r, t]`. These 32 floats occupy $32 \times 4 = 128$ contiguous bytes. Assuming alignment, hardware serves the entire warp with **1 DRAM burst transaction (128 bytes transferred)**. Utilization = $128/128 = 100\%$.
2. **Column read (strided)**: Thread t accesses `M[t, c]`. Adjacent threads are separated by $1024 \times 4 = 4{,}096$ bytes, so every thread hits a different 128-byte burst section. Hardware issues **32 DRAM burst transactions** ($32 \times 128 = 4{,}096$ bytes transferred) just to deliver 128 useful bytes. Utilization = $128/4{,}096 = 3.125\%$—a $32\times$ **waste of HBM bandwidth**.

Interpretation: On a memory-bound kernel, changing loop indexing so consecutive threads in a warp map to the contiguous (innermost) memory dimension can yield up to an order-of-magnitude speedup without changing a single arithmetic operation.

Lever 4B: Shared Memory Bank Conflicts and Swizzling

Once data is staged inside on-chip **shared memory (SMEM)**, DRAM burst mode no longer applies, yet an analogous hardware constraint governs SMEM throughput: **memory banks**.

To serve all 32 threads of a warp in a single cycle without building a prohibitively expensive 32-port monolithic SRAM, GPU shared memory is physically divided into **32 independent banks**, each **4 bytes wide**. Successive 4-byte words map round-robin across the banks:

$$\text{bank}(\text{word_index}) = \text{word_index} \bmod 32$$

Word 0 sits in Bank 0, Word 1 in Bank 1, …, Word 31 in Bank 31, Word 32 wraps back to Bank 0, and so on.

- **Conflict-free access (1 cycle)**: In a stride-1 row read of 4-byte floats, T0 reads Bank 0, T1 reads Bank 1, …, T31 reads Bank 31. All 32 banks respond in parallel in a single cycle.
- **32-way bank conflict**: Suppose a tile in SMEM has 32 columns of 4-byte floats (`tile[32][32]`), and a warp reads down Column 0 (`tile[0][0]`, `tile[1][0]`, …, `tile[31][0]`). Because row stride is 32 words, the word indices are $0, 32, 64, \ldots, 992$. Every single one has $\text{word_index} \bmod 32 = 0$—all 32 threads hit **Bank 0** at *different* addresses! Because a bank can only serve one address per cycle, hardware serializes the warp's request into **32 sequential accesses**, cutting SMEM bandwidth by $32\times$.

DEFINITION

A **shared memory bank conflict** occurs when multiple threads in the same warp simultaneously access *different* addresses that map to the same 4-byte-wide shared memory bank, forcing the hardware to serialize those accesses across multiple cycles.

COMMON MISCONCEPTION

You might think coalescing and bank conflicts are the same phenomenon, or that reading the exact same shared memory address from multiple threads causes a bank conflict. Both are false. Coalescing governs off-chip HBM DRAM bursts (where contiguous addresses merge into one burst), while bank conflicts govern on-chip SMEM's 32 parallel SRAM banks. Moreover, a bank conflict occurs only when threads in a warp access *different* addresses in the same bank—every thread reading the same word does not require separate bank lookups.

Two standard layout fixes eliminate column bank conflicts in 2D shared memory tiles:

1. **Fix 1 — Padding**: Allocate **33 columns** instead of 32 (`tile[32][33]`). Now moving down one row advances the word index by 33, which shifts the bank index by $33 \bmod 32 = +1$ per row. Column 0 maps to Banks $0, 1, 2, \ldots, 31$—completely conflict-free, while rows remain stride-1 and conflict-free. The downside is that the extra padding column wastes SMEM capacity and breaks power-of-two alignments.
2. **Fix 2 — Swizzling**: Instead of adding dummy padding bytes, permute the column placement within each row using a bitwise XOR:

$$\text{physical_col} = \text{col} \oplus \text{row}$$

Look at a 4-bank illustration of `bank = col XOR row`:

- **Row 0**: Columns 0, 1, 2, 3 map to `B0, B1, B2, B3`.
- **Row 1**: Columns 0, 1, 2, 3 map to `B1, B0, B3, B2`.
- **Row 2**: Columns 0, 1, 2, 3 map to `B2, B3, B0, B1`.

Check any horizontal row: all banks are distinct (no row conflict). Check any vertical column—for instance Column 0 (`B0, B1, B2, ...`) or Column 1 (`B1, B0, B3, ...`): all banks are distinct (no column conflict). Best of all, swizzling wastes **zero bytes** of shared memory.

DEFINITION

Swizzling is a zero-overhead memory layout permutation (such as XORing row and column indices) that redistributes both row accesses and column accesses across distinct shared memory banks without wasting capacity on padding.

WORKED EXAMPLE

Worked Example: Verifying Conflict-Free Access Under Padding vs. XOR Swizzling

Setup: Consider a 4×4 tile of 4-byte floats mapped across 4 banks (Banks 0–3) so we can trace every entry explicitly. Check the bank accessed by 4 threads reading Column 1 (`tile[0][1]`, `tile[1][1]`, `tile[2][1]`, `tile[3][1]`) under (a) naive 4×4 layout, (b) padded 4×5 layout, and (c) XOR-swizzled 4×4 layout.

Steps:

1. **Naive 4×4 (`index = 4*r + c`)**:

- Word indices for Column 1 ($r = 0, 1, 2, 3$): $1, 5, 9, 13$.
- Banks ($\bmod 4$): $1, 1, 1, 1 \rightarrow$ **4-way bank conflict** (4 cycles).

2. **Padded 4×5 (`index = 5*r + c`)**:
   - Word indices for Column 1: $1, 6, 11, 16$.
   - Banks ($\bmod 4$): $1, 2, 3, 0 \rightarrow$ **Conflict-free** (1 cycle), at the cost of 25% extra SMEM (20 floats stored instead of 16).
3. **XOR-swizzled 4×4 (`physical_c = c XOR r`, `index = 4*r + (c XOR r)`)**:
   - Row 0: `1 XOR 0 = 1` $\rightarrow$ Bank 1
   - Row 1: `1 XOR 1 = 0` $\rightarrow$ Bank 0
   - Row 2: `1 XOR 2 = 3` $\rightarrow$ Bank 3
   - Row 3: `1 XOR 3 = 2` $\rightarrow$ Bank 2
   - Banks: $1, 0, 3, 2 \rightarrow$ **Conflict-free** (1 cycle), using only 16 floats (**0 wasted bytes**).

Interpretation: XOR swizzling gives the exact same conflict-free 1-cycle access for both rows and columns as padding, while keeping memory allocations compact and power-of-two aligned. We'll need this when we get to Section 6, because Hopper and Blackwell's hardware **TMA swizzle modes** apply this exact permutation automatically during tile loads.

KEY IDEA

Physical memory layout governs realized bandwidth at two distinct levels: consecutive threads in a warp must access contiguous HBM addresses to **coalesce** into ≥ 128-byte DRAM bursts, and warp accesses to on-chip SMEM must spread across its 32 4-byte banks via **padding** or zero-overhead **XOR swizzling** to avoid serialized bank conflicts.

Now that we can transfer data cleanly from HBM into conflict-free shared memory, we can deploy the single highest-leverage optimization for matrix multiplication: **tiling**, which exploits shared memory to reuse every fetched operand dozens of times—and exposes subtle quantization cliffs when tile grids meet finite matrix sizes and SM counts.

✎ Check Your Understanding

1. **What would happen if…**: In a row-major matrix of floats, you transpose the thread assignment so that consecutive threads in a warp (`threadIdx.x = 0, 1, ..., 31`) read consecutive elements down a column instead of across a row. What happens to DRAM burst efficiency?

   Answer: The loads become uncoalesced (strided by the row width), so each thread lands in a different DRAM burst section; instead of 1 burst transaction per warp, hardware must issue up to 32 separate bursts, wasting almost all transferred HBM bandwidth.

2. **Fill in the blank**: Shared memory is divided into _____ banks, each _____ bytes wide, and _____ avoids column bank conflicts without wasting any SMEM capacity by XORing row and column bits.

   Answer: 32; 4; swizzling.

Section 5: Tiling Matrix Multiplication, Quantization Cliffs, and Wave Scheduling

Dense matrix multiplication $C = A \times B$ (for $A, B, C \in \mathbb{R}^{N \times N}$) is the computational backbone of every language model. In a naive, **non-tiled matmul**, each thread computes one output entry $C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$ directly from global HBM. That single output entry requires reading an entire length-N row of A and a length-N column of B from HBM. Across all N^2 outputs, **every element of A and B is read N times from global memory** (and the row reads of A are uncoalesced). Despite performing $2N^3$ FLOPs, a non-tiled matmul suffers terrible HBM traffic and stalls far below Peak Tensor Core throughput.

Lever 5: Tiling Matrix Multiplication

Tiling transforms matrix multiplication from memory-bound to compute-bound by exploiting on-chip shared memory for data reuse. Partition A, B, and C into grids of $T \times T$ square submatrices (tiles). Each CUDA thread block is assigned to compute one $T \times T$ output tile of C, keeping that output tile's running accumulators on chip. To compute its output tile, the block marches across N/T phases along the inner reduction dimension K:

1. **Load A and B tiles $\rightarrow$ SMEM**: All threads in the block cooperate to load the current $T \times T$ tile of A and $T \times T$ tile of B from HBM into SMEM (arranging thread indices so both tile loads achieve full DRAM coalescing).
2. **Sync**: Execute `__syncthreads()` so every thread sees the completed SMEM tiles.
3. **Accumulate partial sums**: Multiply the $T \times T$ tile of A by the $T \times T$ tile of B using fast SMEM reads, accumulating into the block's output tile of C.
4. **Next tile**: Synchronize and advance to the next $T \times T$ pair along K until all N/T phases finish, then write the final $T \times T$ tile of C to HBM once.

DEFINITION

Tiling partitions large matrices into small $T \times T$ submatrices that fit inside fast on-SM shared memory so each loaded element is reused T times on chip, reducing global HBM reads by a factor of T and boosting arithmetic intensity by a factor of T.

Because every $T \times T$ tile of A loaded into SMEM is reused across all T columns of the B tile (and vice versa), each element of A and B is read from HBM only N/T **times** instead of N times—a **factor of T reduction** in global memory traffic.

WORKED EXAMPLE

Worked Example: HBM Traffic and Arithmetic Intensity of Non-Tiled vs. Tiled Matmul

Setup: Multiply two $N \times N$ FP16 matrices ($N = 1024$, 2 bytes/element) using (a) non-tiled matmul and (b) tiled matmul with $T = 32$. Ignore the negligible C-write term (N^2 elements) to focus on the A and B input reads.

Steps:

1. **Total FLOPs** (both methods): Each of the N^2 outputs does N multiplies and N adds:

$$\text{FLOPs} = 2N^3 = 2 \times 1024^3 \approx 2.15 \times 10^9 \text{ FLOPs}$$

2. **Non-tiled HBM input reads**: Every element of A (N^2 elements) and B (N^2 elements) is read N times:

$$\text{Bytes}_{\text{non-tiled}} = 2 \times N^3 \times 2\,\text{B} = 4N^3 \approx 4.29 \times 10^9 \text{ B}$$

$$\text{Intensity}_{\text{non-tiled}} = \frac{2N^3}{4N^3} = 0.5 \text{ FLOP/B}$$

3. **Tiled ($T = 32$) HBM input reads**: Each element of A and B is read $N/T = 1024/32 = 32$ times:

$$\text{Bytes}_{\text{tiled}} = 2 \times N^2 \times \frac{N}{T} \times 2\,\text{B} = \frac{4N^3}{T} = \frac{4.29 \times 10^9}{32} \approx 1.34 \times 10^8 \text{ B}$$

$$\text{Intensity}_{\text{tiled}} = \frac{2N^3}{4N^3/T} = 0.5 \times T = 0.5 \times 32 = 16 \text{ FLOP/B}$$

Interpretation: Increasing tile size T reduces HBM traffic by $T\times$ and multiplies arithmetic intensity by $T\times$, pushing GEMM directly across the roofline ridge line into the compute-bound regime. Larger tiles (such as 128×128 or 256×128) push intensity even higher—which is why Tensor Core kernels deliberately allocate large SMEM tiles per block.

Tile Quantization, Burst Alignment, and the Matrix Mystery

Tiling introduces a counterintuitive engineering phenomenon: **making a matrix slightly *larger* can make multiplying it dramatically *faster***. Two hardware boundary effects explain this "matrix mystery":

1. **Tile quantization**: Hardware GEMM kernels tile the output matrix using fixed tile sizes such as 128×128. Consider a 256×256 output matrix: it divides cleanly into $2 \times 2 = 4$ full tiles at 100% utilization. Now add a single column to make it 256×257. The kernel must launch $2 \times 3 = 6$ tiles—an extra column of 2 tiles where 127 out of 128 columns perform wasted padding work! You increased the math by 0.4%, yet increased the tile work by 50%.
2. **Burst alignment**: When the leading dimension (row stride) of a matrix aligns cleanly with DRAM burst widths, each tile row fetches in 1 clean burst. When the leading dimension is unaligned (such as an odd number like 257), tile rows straddle burst boundaries, spanning extra bursts, breaking wide vectorized load instructions, and forcing libraries to drop down to slower fallback kernels.

> **DEFINITION**
> **Tile quantization** is the efficiency loss that occurs when matrix dimensions are not clean multiples of the kernel's tile size, forcing boundary blocks to execute full-tile work on mostly empty or masked-out elements.

This exact effect appeared in a famous historical case study from **nanoGPT**: GPT-2's vocabulary size is $50{,}257$ (an awkward, unaligned number). Simply padding the vocabulary dimension upward from $50{,}257 \rightarrow 50{,}304$ (the nearest multiple of 64) delivered a $\sim 25\%$ **speedup** despite computing *more* matrix dimensions. The practical rule of thumb follows directly: always pad key matrix dimensions—vocabulary size, hidden size, and head count—to clean multiples of tile and burst widths.

Wave Quantization: A100 Baseline vs. Blackwell B300

Even if a matrix divides into tiles cleanly, a second quantization cliff arises when the total number of tiles is mapped onto the GPU's finite number of SMs.

Suppose each SM runs 1 block (tile) at a time. A **wave** is one full round of concurrent blocks occupying all SMs on the GPU. If you plot GEMM throughput against matrix dimension N, you do not get a smooth curve; instead, you get a **periodic sawtooth** with sharp downward cliffs every time the total tile count crosses an integer multiple of the GPU's SM count.

> **DEFINITION**
> **Wave quantization** is the throughput cliff that occurs when the total number of output tiles slightly exceeds an integer multiple of the GPU's concurrent block capacity across all SMs, spilling a tiny handful of leftover tiles into an extra execution wave during which most SMs sit idle.

> **WORKED EXAMPLE**
> **Worked Example: Wave Quantization Cliffs on A100 (108 SMs) vs. Blackwell B200/B300 (148 SMs)**

Setup: Assume a GEMM kernel uses 256×128 output tiles (1 tile per SM per wave) on square $N \times N$ matrices. Compute the number of tiles and execution waves on a historical A100 (108 SMs), H100 SXM (132 SMs), and Blackwell B200/B300 (148 SMs, compute capability 10.3) at $N = 1792, 1793, 2048,$ and 2049.

Steps:

1. $N = 1792$:

$$\text{Tiles} = \left\lceil \frac{1792}{256} \right\rceil \times \left\lceil \frac{1792}{128} \right\rceil = 7 \times 14 = 98 \text{ tiles}$$

   - A100 (108 SMs): $98 \leq 108 \implies$ **1 full wave**.
   - B200/B300 (148 SMs): $98 \leq 148 \implies$ **1 wave**.

2. $N = 1793$: Adding 1 row and 1 column bumps both ceilings by 1:

$$\text{Tiles} = \left\lceil \frac{1793}{256} \right\rceil \times \left\lceil \frac{1793}{128} \right\rceil = 8 \times 15 = 120 \text{ tiles}$$

   - A100 (108 SMs): $120 > 108 \implies$ **2 waves**! Wave 1 runs 108 tiles; Wave 2 runs just 12 tiles while **96 SMs sit completely idle**, nearly doubling runtime.
   - H100 SXM (132 SMs) and B200/B300 (148 SMs): $120 \leq 132$ and $120 \leq 148 \implies$ **still 1 wave!** The A100 cliff disappears at $N = 1793$.

3. $N = 2048$ **on B200/B300 (148 SMs)**:

$$\text{Tiles} = \left\lceil \frac{2048}{256} \right\rceil \times \left\lceil \frac{2048}{128} \right\rceil = 8 \times 16 = 128 \text{ tiles} \leq 148 \implies \mathbf{1} \text{ wave}$$

4. $N = 2049$ **on B200/B300 (148 SMs)**:

$$\text{Tiles} = \left\lceil \frac{2049}{256} \right\rceil \times \left\lceil \frac{2049}{128} \right\rceil = 9 \times 17 = 153 \text{ tiles}$$

   - On B200/B300 (148 SMs): $153 > 148 \implies$ **2 waves**! Wave 1 runs 148 tiles; Wave 2 runs only **5 tiles**, leaving **143 of 148 SMs idle**!

Interpretation: Upgrading from A100 to Blackwell does not remove wave quantization—it moves the cliff to multiples of 148 tiles. Crossing from 148 to 153 tiles creates a catastrophic tail wave where 96.6% of the GPU sits idle.

Mitigating Wave Quantization: Persistent Kernels, CLC, and Split-K / Stream-K

Modern GEMM libraries (such as **CUTLASS** and **DeepGEMM**) fight wave quantization and wave-boundary overhead using two complementary scheduling techniques:

1. **Persistent Kernels + Cluster Launch Control (CLC)**: Instead of launching one CTA per output tile and letting the block scheduler tear down and spin up CTAs between waves, a **persistent kernel** launches exactly **1 resident CTA per SM** (148 persistent CTAs on B300). Each persistent CTA loops over a **Tile Work Queue**, pulling its next output tile dynamically via Blackwell's hardware **Cluster Launch Control (CLC)** tile fetcher. Keeping the CTA alive across tiles lets it overlap the epilogue (writing out finished tile t) with the prologue (prefetching inputs for tile $t+1$), removing inter-wave drain gaps.
2. **Split-K and Stream-K**: Even a persistent kernel cannot hide a 5-tile tail wave if each output tile is locked to a single SM. **Split-K** and **Stream-K** solve the tail-wave imbalance by partitioning those leftover tail output tiles along the reduction dimension K into slices ($K_0, K_1, K_2, \ldots$). Multiple SMs (SM a, SM b, SM c) simultaneously compute partial sums for different K-slices of the same output tile, and then sum their partial results together. Spreading leftover tiles evenly across all SMs dramatically reduces wave quantization, though it does not eliminate it completely. (Flag for Section 8: changing the Split-K factor dynamically with batch size silently alters floating-point summation order, which is a primary cause of batch nondeterminism!)

DEFINITION

A **persistent kernel** launches one long-lived CTA per SM (148 on B300) that dynamically pulls successive output tiles from a work queue—accelerated on Blackwell by hardware **Cluster Launch Control (CLC)**—to overlap tile epilogues and prologues across waves.

DEFINITION

Split-K / Stream-K partitions the inner reduction dimension K of leftover tail output tiles across multiple SMs so that all SMs share the work of computing partial sums and reducing them, mitigating tail-wave underutilization.

KEY IDEA

Tiling shrinks GEMM HBM traffic by a factor of T, pushing dense matmuls into the compute-bound regime. Achieving peak real-world throughput also requires avoiding **tile quantization** and **unaligned bursts** (by padding dimensions to clean multiples) and mitigating **wave quantization** cliffs (at multiples of 148 SMs on Blackwell) via persistent CLC kernels and Split-K / Stream-K scheduling.

We have now completed all five roofline optimization levers. Yet if you implement a tiled, coalesced, swizzled, persistent GEMM using the classic synchronous CUDA pattern—load a tile, hit `__syncthreads()`, run MMA, hit `__syncthreads()`, load the next tile—your Tensor Cores will still spend a massive fraction of their time sitting idle on Blackwell. To keep 5th-generation Tensor Cores continuously saturated, the hardware itself had to abandon the synchronous CTA model.

✎ Check Your Understanding

1. **Quick Computation**: A square matrix multiplication of size $N = 1024$ is tiled into 64×64 blocks. By what factor does tiling reduce HBM reads of A and B compared to a non-tiled matmul, and how many phases along K does each block execute?

   Answer: Tiling reduces HBM reads by a factor of $T = 64$, and each block executes $N/T = 1024/64 = 16$ phases along K.

2. **What would happen if…**: You run a GEMM with 128×128 tiles on a B300 (148 SMs) and increase the output dimensions from 1024×2304 to 1024×2432 (1 tile per SM per wave). How does the wave structure change?

   Answer: At 1024×2304, there are $(1024/128) \times (2304/128) = 8 \times 18 = 144$ tiles (1 wave on 148 SMs). At 1024×2432, there are $8 \times 19 = 152$ tiles, which spills into 2 waves with only 4 tiles in Wave 2 (144 SMs idle), causing a steep wave-quantization throughput drop.

Section 6: The Asynchronous Blackwell Execution Model

In classic CUDA kernels (the Ampere era), all warps inside a thread block played identical roles in a **synchronous load-sync-compute loop**: every warp helped load Tile 0 from HBM to SMEM, waited at a CTA-wide `__syncthreads()` barrier, executed MMA on Tile 0, waited at another `__syncthreads()` barrier, and then began loading Tile 1. Look at the execution timeline of that synchronous CTA: during every `Load` block and every `sync` block, the Tensor Cores sit completely idle.

On Hopper and Blackwell, Tensor Core throughput outpaced everything else so sharply that synchronous loops starve the Tensor Cores. To eliminate those bubbles, modern NVIDIA GPUs introduced an **asynchronous, warp-specialized pipeline**. Look at the corresponding Blackwell timeline split into three concurrent lanes—`Load`, `MMA`, and `Other` (such as Softmax):

- While `Load` copies `TMA 1`, `MMA` computes `MMA 0`.
- While `Load` copies `TMA 2`, `MMA` computes `MMA 1`, and `Other` runs `Softmax 0`.
- While `Load` copies `TMA 3`, `MMA` computes `MMA 2`, and `Other` runs `Softmax 1`.

Data copy, Tensor Core matmuls, and non-matmul operations execute concurrently in steady state. Five specific hardware building blocks—introduced across Hopper (SM90) and perfected in Blackwell (SM100)—make this asynchronous pipeline work.

Building Block 1: TMA (Tensor Memory Accelerator) and `mbarrier`

Before Hopper, copying a 2D tile from HBM into SMEM required 128 or more SIMT threads to compute element addresses, check boundary conditions, handle padding, and issue individual load instructions. All of that address arithmetic and load issuing burned registers and instruction issue slots on the same threads needed for math.

TMA (Tensor Memory Accelerator), introduced on Hopper (SM90) and carried forward to Blackwell (SM100), replaces that 128-thread copy loop with a dedicated hardware engine inside each SM. A **single thread** issues **one** TMA instruction describing a multidimensional tile. The **TMA Copy Engine** then asynchronously executes the entire bulk GMEM $\leftrightarrow$ SMEM transfer in hardware, automatically handling:

- Hardware address calculation
- Bounds checking and zero-padding
- **Automatic SMEM swizzling** (applying the exact bank-conflict-free layout from Section 4 during the copy so the SMEM tile arrives conflict-free)

> **DEFINITION**
> The **Tensor Memory Accelerator (TMA)** is a dedicated hardware copy engine on Hopper and Blackwell SMs that executes asynchronous, bulk multidimensional tile transfers between global memory and shared memory from a single-thread instruction while automatically handling bounds, padding, and bank-conflict-free SMEM swizzling.

Note an important physical detail: DRAM coalescing (Section 4) still governs HBM efficiency underneath the TMA engine, even though software no longer hand-codes per-thread load addresses.

Because TMA runs asynchronously in the background while threads continue executing, a full-CTA `__syncthreads()` barrier is too blunt to coordinate when a tile has arrived. Instead, TMA pairs with **`mbarrier`**—an asynchronous hardware barrier stored in SMEM that tracks a **byte-completion count**. The issuing thread tells the `mbarrier` how many bytes the TMA copy will deliver; when the TMA engine finishes writing those bytes into SMEM, it decrements the barrier automatically and wakes only the consumer warps waiting on that specific tile, without stalling unrelated warps in the CTA. For example, FlashAttention-3 uses `SM90_TMA_LOAD` paired with `mbarrier` to load Q, K, V tiles.

> **DEFINITION**
> An **`mbarrier`** is an asynchronous synchronization barrier residing in shared memory that tracks arrival and TMA byte-completion counts, enabling producer and consumer warps to hand off individual pipeline stages without stalling the entire thread block with `__syncthreads()`.

Building Block 2: Asynchronous Tensor Cores — Hopper `wgmma` (History)

To overlap matrix multiplication with other work, **Hopper (SM90)** introduced **`wgmma`** (warpgroup matrix multiply-accumulate)—an asynchronous Tensor Core instruction issued cooperatively by a **warpgroup** (4 warps = 128 threads). Compared to older synchronous MMAs, `wgmma` brought two breakthroughs:

1. **Direct SMEM operands**: `wgmma` reads at least one matrix operand tile straight from shared memory, cutting register-file read traffic.
2. **Asynchronous issue**: Once the warpgroup launches `wgmma`, the Tensor Cores run in the background while the warpgroup immediately overlaps other work (such as softmax) before waiting on completion.

However, examining Hopper's `wgmma` dataflow exposes two structural bottlenecks that forced Blackwell to redesign the instruction:

- **Register-file bottleneck**: The MMA accumulators still lived in the general-purpose **register file** alongside softmax and epilogue variables, creating severe register pressure.
- **128-thread coordination overhead**: Issuing a single MMA still tied up an entire 128-thread warpgroup.

Building Block 3: Warp Specialization

Once TMA handles loads asynchronously and Tensor Cores handle MMAs asynchronously, making every warp run identical code is wasteful: a warp issuing TMA copies needs very few registers, whereas a warp performing math and softmax needs as many registers as possible.

Warp specialization divides the warps inside a single CTA into distinct permanent roles—specifically **Producer warps** and **Consumer warps**—communicating through a multi-stage **SMEM Ring Buffer** governed by pairs of `full` and `empty` `mbarrier`s:

- **Producer warp(s)**: Need very few registers (freeing hardware registers to be reallocated to consumers). They loop over tiles, wait on the `empty` `mbarrier` for the next ring slot, issue a TMA load into that slot, and let TMA signal the `full` `mbarrier` upon completion.
- **SMEM Ring Buffer (Pipeline Stages)**: Holds multiple tile slots simultaneously—for instance, in a 4-stage ring buffer: Stage 0 is *ready* for the consumer, Stage 1 is currently being *filled by TMA*, and Stages 2 and 3 are *empty* and available for the producer.
- **Consumer warp(s)**: Receive the extra reallocated registers. They wait on the `full` `mbarrier`, run Tensor Core MMAs on that stage, signal the `empty` `mbarrier` to release the buffer back to the producer, and execute softmax or the epilogue.

DEFINITION

Warp specialization assigns dedicated, distinct roles (such as TMA data producer, Tensor Core MMA consumer, and softmax/epilogue worker) to different warps within the same thread block, allowing loads and compute to run asynchronously ahead of each other through an SMEM ring buffer while reallocating registers toward compute-heavy warps.

WORKED EXAMPLE

Worked Example: Tracing a 4-Stage Warp-Specialized SMEM Ring Buffer

Setup: A single CTA runs 1 producer warp and 1 consumer group sharing a 4-stage SMEM ring buffer (Stages 0, 1, 2, 3), initialized with all 4 `empty` mbarriers signaled (ready to fill) and all 4 `full` mbarriers unsignaled. Trace how the producer prefetches ahead of the consumer across Tiles 0 through 4.

Steps:

1. **Prologue fill**: The producer loops rapidly through Tiles 0, 1, 2, 3, waiting on `empty[0..3]` (all immediately satisfied) and launching 4 asynchronous TMA loads into Stages `0, 1, 2, 3`. The

producer now blocks on `empty[0]` for Tile 4 because all 4 ring slots are occupied.

2. **Steady-state handoff**: As soon as TMA finishes writing Tile 0 into Stage 0, TMA signals `full[0]`. The consumer wakes immediately, runs MMA on Stage 0 *while TMA is still transferring Tiles 1, 2, and 3 into Stages 1–3*.
3. **Slot release and wrap-around**: When the consumer finishes reading Stage 0, it signals `empty[0]` and immediately waits on `full[1]` (which has likely already finished arriving!). Meanwhile, the producer wakes on `empty[0]` and immediately launches TMA for Tile 4 into Stage 0.

Interpretation: Because the ring buffer keeps 3 tiles in flight ahead of the consumer, the HBM latency of TMA loads is completely hidden behind Tensor Core execution. FlashAttention-3's Hopper mainloop uses this exact design with 1 producer warp and 1 to 3 MMA warpgroups.

Building Block 4: Thread Block Clusters, Distributed SMEM, and TMA Multicast

Recall from Section 2 that in standard CUDA, blocks are strictly isolated and can share data only through slow global HBM. **Thread block clusters**—introduced in Hopper and expanded in Blackwell—add a new hardware hierarchy level between a single Block (CTA) and the Grid.

A **thread block cluster** is a small group of CTAs guaranteed by the hardware scheduler to run **concurrently on neighboring SMs**. Co-residency unlocks two capabilities that take tiling one level higher:

1. **Distributed Shared Memory (DSMEM)**: CTAs inside the same cluster can read and write each other's shared memory directly across SM-to-SM links without ever touching HBM.
2. **TMA Multicast**: When neighboring CTAs compute adjacent output tiles in the same row of C—say `Tile C(0,0)` on SM 0, `Tile C(0,1)` on SM 1, and `Tile C(0,2)` on SM 2—all three CTAs require the exact same input tile of A. With **TMA hardware multicast**, HBM performs **1 read** of that shared tile and broadcasts it simultaneously into `SMEM 0`, `SMEM 1`, and `SMEM 2`.

DEFINITION

A **thread block cluster** is a group of CTAs guaranteed to execute concurrently on neighboring SMs, enabling **Distributed Shared Memory (DSMEM)** for direct SM-to-SM shared memory access and **TMA multicast** to broadcast a single HBM tile read to every CTA in the cluster.

WORKED EXAMPLE

Worked Example: HBM Read Reduction from TMA Multicast in a 3-CTA Cluster

Setup: Three CTAs on neighboring SMs compute `Tile C(0,0)`, `Tile C(0,1)`, and `Tile C(0,2)`, all requiring the same 32 KB tile of matrix A in a given K-phase. Compare total HBM reads of A without and with a 3-CTA cluster using TMA multicast.

Steps:

1. **Without cluster multicast**: Each CTA independently issues an HBM load for the 32 KB tile of A: $3 \times 32\ \text{KB} = 96\ \text{KB}$ requested from HBM (unless caught by L2).
2. **With 3-CTA TMA multicast**: One hardware multicast load reads the 32 KB tile from HBM once and distributes it directly to SMEM 0, SMEM 1, and SMEM 2: $32\ \text{KB}$ read from HBM—a $3\times$ **reduction** in redundant HBM requests for the shared operand.

Interpretation: Clusters extend the data-reuse principle of tiling across neighboring SMs, slashing HBM pressure on operands shared along rows or columns of the output matrix.

Building Block 5: Blackwell SM100 `tcgen05`, Tensor Memory (TMEM), and 2-CTA Cooperative MMA

Blackwell (SM100) resolves both remaining bottlenecks of Hopper's `wgmma`—register pressure and 128-thread issue overhead—with three tightly integrated hardware innovations:

1. **Single-thread MMA (`tcgen05`)**: Instead of coordinating a 128-thread warpgroup to launch a matrix multiply, a **single thread** on Blackwell issues asynchronous 5th-generation Tensor Core MMAs via `tcgen05`. Every other warp in the CTA stays completely free to handle TMA loads, softmax, rescaling, or epilogues.
2. **Tensor Memory (TMEM)**: Blackwell adds an explicitly allocated per-SM memory array specifically for MMA accumulators, organized as **128 lanes $\times$ 512 columns of 32-bit cells**. Moving MMA accumulators completely out of the general register file eliminates the severe register-pressure bottleneck that plagued Hopper kernels, freeing the register file for fused softmax, scaling, and epilogues. Warps explicitly allocate TMEM and access it only when moving finished tiles out to softmax or the epilogue.

DEFINITION

Tensor Memory (TMEM) is a dedicated, explicitly allocated per-SM memory array on Blackwell SM100 organized as $128\ \text{lanes} \times 512\ \text{columns} \times 32$-bit cells, designed to hold `tcgen05` Tensor Core MMA accumulators off the general register file.

WORKED EXAMPLE

Worked Example: Sizing Blackwell's Tensor Memory (TMEM)

Setup: Calculate the exact capacity of TMEM on one Blackwell SM from its hardware geometry ($128\ \text{lanes} \times 512\ \text{columns} \times 32\text{-bit cells}$), and determine how many 128×128 FP32 accumulator tiles can reside in TMEM simultaneously.

Steps:

1. **Total cells per SM**:

$$128 \times 512 = 65{,}536\ \text{32-bit cells}$$

2. **Total bytes per SM** (each 32-bit cell = 4 bytes):

$$65{,}536 \times 4 \text{ bytes} = 262{,}144 \text{ bytes} = 256 \text{ KB}$$

3. **Number of 128×128 FP32 tiles**: One 128×128 FP32 tile contains $128 \times 128 = 16{,}384$ 32-bit values (64 KB), occupying 128 lanes $\times$ 128 columns of TMEM.

$$\frac{512 \text{ columns}}{128 \text{ columns/tile}} = 4 \text{ tiles simultaneously}$$

Interpretation: A single Blackwell SM's TMEM holds four entire 128×128 FP32 accumulator matrices (such as two score tiles S_0, S_1 and two output tiles O_0, O_1 in FlashAttention-4!) without consuming a single register from the SM's general-purpose register file.

3. **Blackwell 2-CTA Cooperative MMA (`cta_group::2`)**: Taking clusters one step further, Blackwell lets **two SMs in a cluster cooperate on a single, larger `tcgen05` MMA tile**. Look at the 2-SM cluster dataflow:
   - **SM 0 (CTA 0 — Leader)**: A leader thread issues one cooperative `tcgen05` (`cta_group::2`) instruction. `SMEM 0` stages the top half of A and only the **left half of** B. `Tensor Core 0` reads operands from both `SMEM 0` and `SMEM 1` across the hardware **Cluster Link** and accumulates the top half of the larger output tile into `TMEM 0`.
   - **SM 1 (CTA 1 — Partner SM)**: `SMEM 1` stages the bottom half of A and only the **right half of** B. `Tensor Core 1` reads from both `SMEM 1` and `SMEM 0` over the Cluster Link and accumulates the bottom half of the larger output tile into `TMEM 1`.

   Because each SM stages only half of the shared B operand in its local SMEM and exchanges the other half over the cluster link, **2-CTA cooperative MMA cuts each SM's operand traffic in half**.

DEFINITION

2-CTA Cooperative MMA (`cta_group::2`) is a Blackwell SM100 mode in which two co-resident SMs in a cluster collaborate on a single larger `tcgen05` matrix multiply—each staging only half the shared operand in its SMEM and exchanging across the cluster link—halving per-SM operand traffic.

COMMON MISCONCEPTION

You might think `tcgen05` and TMEM make shared memory obsolete on Blackwell. That breaks because TMEM specifically holds Tensor Core *accumulators* (and feeds finished results to warps), while TMA still stages incoming A and B operand tiles from HBM into swizzled **SMEM** ring buffers to feed the Tensor Cores. SMEM, TMEM, and the register file each serve a distinct role in the Blackwell datapath.

KEY IDEA

Blackwell replaces synchronous load-sync-compute loops with an asynchronous, warp-specialized pipeline built on five hardware primitives: **TMA + `mbarrier`** (single-thread bulk swizzled copies), **warp specialization** (producer/consumer ring buffers), **clusters + DSMEM/multicast** (cross-SM sharing), **single-thread `tcgen05` + TMEM** (freeing registers from MMA accumulators), and **2-CTA cooperative MMAs (`cta_group::2`)** (halving per-SM operand traffic).

We now possess every hardware concept, roofline lever, and Blackwell asynchronous primitive. Next, we put all of them together in a single end-to-end case study: **FlashAttention**, tracing its evolution from its core online-softmax tiling algorithm (FA1 and FA2) through Hopper's asynchronous mainloop (FA3) to Blackwell's five-role `tcgen05`/TMEM pipeline (FA4).

✎ Check Your Understanding

1. **Fill in the blank**: On Hopper, `wgmma` required a _____ (128 threads) to issue an MMA and stored accumulators in the _____; on Blackwell, `tcgen05` is issued by _____ and stores accumulators in _____.

   Answer: warpgroup; register file; a single thread; Tensor Memory (TMEM).

2. **Yes or No (with justification)**: In a Blackwell 2-CTA cooperative MMA (`cta_group::2`), does each of the two participating SMs need to load the entire B operand tile from HBM into its own SMEM?

   Answer: No. Each SM loads only half of the B operand into its local SMEM and reads the partner SM's half across the hardware cluster link, halving each SM's operand traffic.

Section 7: FlashAttention 1–4: From Tiled Online Softmax to Blackwell Warp Specialization

Attention is the defining operation of Transformer language models. Given query, key, and value matrices $Q, K, V \in \mathbb{R}^{N \times d}$ (where N is sequence length and d is head dimension), standard attention computes three steps:

$$S = QK^T \in \mathbb{R}^{N \times N}, \qquad P = \text{softmax}(S) \in \mathbb{R}^{N \times N}, \qquad O = PV \in \mathbb{R}^{N \times d}$$

Why Standard Attention Is Memory-Bound

In a standard unfused PyTorch implementation, each sub-step launches a separate kernel that materializes the huge $N \times N$ intermediate matrix in HBM: compute $S = QK^T$ and write N^2 entries to HBM; read S to apply the causal mask and write N^2 entries back; read again to run softmax and write

P; read P to apply dropout and write it back; finally read P and V to compute $O = PV$. Every intermediate step pays $O(N^2)$ **HBM reads and writes**.

Look at the historical GPT-2 benchmark from Dao et al. (2022) comparing Standard PyTorch attention against FlashAttention:

Metric (GPT-2 Historical Baseline)	Standard PyTorch	FlashAttention
Compute (GFLOPs)	66.6	**75.2**
HBM Read/Write (GB)	40.3	**4.4**
Runtime (ms)	41.7	**7.3**

A stacked bar breakdown of PyTorch's 41.7 ms runtime makes the roofline diagnosis visually unmistakable: the two dense matrix multiplications (QK^T and PV) are thin slivers at the top and bottom of the bar, while the vast middle of the bar is consumed by **Mask (HBM)**, **Softmax (HBM)**, and **Dropout (HBM)**. By fusing everything into a single tiled SRAM kernel, FlashAttention shrinks the entire bar down to 7.3 ms—$5.7\times$ **faster**—despite executing **more GFLOPs** (75.2 vs. 66.6) because of backward recomputation! Cutting HBM traffic nearly $10\times$ (from 40.3 GB down to 4.4 GB) completely overrides the extra FLOPs.

FlashAttention-1 and 2 (2022–2023): Tiling Q, K, V in SRAM

Recall the steep gap in the GPU memory pyramid (illustrated by the historical A100 baseline: on-chip **SRAM** at 19 TB/s with 20 MB capacity, **GPU HBM** at 1.5 TB/s with 40 GB capacity, and **CPU DRAM** at 12.8 GB/s with > 1 TB capacity—the same order-of-magnitude SRAM/HBM gap holds on Blackwell). FlashAttention exploits this pyramid using the exact three Part-2 ideas we developed in Sections 3 and 5: **tiling**, **operator fusion**, and **recomputation**.

In the tiled attention dataflow, small blocks Q_i, K_j^T, and V_j are loaded from HBM into on-chip SRAM. Inside SRAM, the kernel computes (1) the score tile $S_{ij} = Q_i K_j^T$, (2) the fused online exponential weights P_{ij}, and (3) the incremental matrix product $P_{ij}V_j$ added into the running output tile O_i. Neither the $N \times N$ score matrix S nor the $N \times N$ attention matrix P is **ever written to HBM**; only the final $N \times d$ output O is written to HBM. During the **backward pass**, instead of saving S and P from the forward pass, FlashAttention stores only O and the compact per-row softmax statistics (one max and one normalizer per row), then **recomputes** S_{ij} and P_{ij} tile-by-tile inside SRAM—trading abundant Tensor Core FLOPs for massive HBM bandwidth savings.

There is one major mathematical obstacle to tiling attention across keys: **softmax normalizes across an entire row of N keys**. How can we compute exact softmax—and multiply by V—one tile at a time without seeing the whole row first?

Online Softmax and the FlashAttention Forward Pass

Recall **standard numerically safe softmax** over a length-V row vector $x = (x_1, \ldots, x_V)$:

$$y_i = \frac{e^{x_i - m_V}}{\sum_{j=1}^{V} e^{x_j - m_V}}, \qquad m_V = \max_{1 \le k \le V} x_k$$

Subtracting the row maximum m_V prevents e^{x_i} from overflowing in floating point. Naively, this requires **3 passes** over the row:

1. Pass 1: find the global max $m_V = \max_{1 \le k \le V} x_k$.
2. Pass 2: sum the shifted exponentials $d_V = \sum_{j=1}^{V} e^{x_j - m_V}$.
3. Pass 3: divide $y_i = e^{x_i - m_V} / d_V$.

Because Pass 2 cannot even begin until Pass 1 finishes reading the entire row, standard safe softmax cannot be tiled.

Online Softmax (Milakov & Gimelshein, 2018) collapses Passes 1 and 2 into a single running pass—turning softmax into a tileable algorithm—by maintaining a running maximum m_j and a running denominator d_j:

- Initialize $m_0 = -\infty$ and $d_0 = 0$.
- For $j = 1, \ldots, V$:

$$m_j = \max(m_{j-1},\ x_j)$$

$$d_j = d_{j-1} \cdot e^{m_{j-1} - m_j} + e^{x_j - m_j}$$

- Finally: $y_i = e^{x_i - m_V} / d_V$.

> **DEFINITION**
> **Online softmax** computes the numerically safe softmax maximum and normalization denominator in a single streaming pass over blocks by rescaling the prior running sum d_{j-1} by $e^{m_{j-1} - m_j}$ whenever a larger maximum m_j is encountered, forming a telescoping sum that is exact at every step.

Why is the recurrence exact? Unroll two steps to see the telescoping cancellation. After step 1, $d_1 = e^{x_1 - m_1}$. At step 2, multiplying d_1 by $e^{m_1 - m_2}$ gives:

$$d_1 \cdot e^{m_1 - m_2} = e^{x_1 - m_1} \cdot e^{m_1 - m_2} = e^{x_1 - m_2}$$

The old maximum m_1 cancels algebraically and is replaced by the new maximum m_2, while $m_{j-1} - m_j \le 0$ guarantees the rescaling factor $e^{m_{j-1} - m_j} \in (0, 1]$ never overflows!

Now generalize from single elements x_j to **blocks (tiles)** of keys and values, and observe the key algebraic property of attention: the output $O = \sum_t P^{(t)} V^{(t)}$ is **linear** in the unnormalized exponential weights! Therefore, we can rescale the running output accumulator $\tilde{O}^{(t-1)}$ by the **exact same rescaling factor** α whenever a new key tile raises the row maximum.

Trace the FlashAttention forward pass for one query tile Q_i held in SRAM across two key/value tiles $(K^{(1)}, V^{(1)})$ and $(K^{(2)}, V^{(2)})$:

- **Tile 1**:

$$S^{(1)} = Q_i (K^{(1)})^T, \quad m^{(1)} = \text{rowmax}(S^{(1)}), \quad \tilde{P}^{(1)} = e^{S^{(1)} - m^{(1)}}, \quad \ell^{(1)} = \text{rowsum}(\tilde{P}^{(1)})$$

$$\tilde{O}^{(1)} = \tilde{P}^{(1)} V^{(1)}$$

- **Tile 2**:

$$S^{(2)} = Q_i (K^{(2)})^T, \quad m^{(2)} = \max\big(m^{(1)},\ \text{rowmax}(S^{(2)})\big), \quad \alpha = e^{m^{(1)} - m^{(2)}}$$

$$\tilde{P}^{(2)} = e^{S^{(2)} - m^{(2)}}, \quad \ell^{(2)} = \alpha \ell^{(1)} + \text{rowsum}(\tilde{P}^{(2)})$$

$$\tilde{O}^{(2)} = \alpha \tilde{O}^{(1)} + \tilde{P}^{(2)} V^{(2)}$$

- **Final epilogue (after all T tiles)**:

$$O_i = \frac{\tilde{O}^{(T)}}{\ell^{(T)}}$$

Breaking down every symbol: $S^{(t)}$ is the raw score tile for key block t; $m^{(t)}$ is the running row-wise maximum seen through tile t; $\alpha = e^{m^{(t-1)} - m^{(t)}} \leq 1$ is the exact exponential correction factor that shifts all previous terms from reference $m^{(t-1)}$ to reference $m^{(t)}$; $\tilde{P}^{(t)}$ are the unnormalized local exponentials; $\ell^{(t)}$ is the running unnormalized row denominator; and $\tilde{O}^{(t)}$ is the running unnormalized output tile residing in SRAM (or TMEM on Blackwell). Dividing by $\ell^{(T)}$ once in the final epilogue yields the mathematically exact attention output O_i.

WORKED EXAMPLE

Worked Example: End-to-End 2-Tile FlashAttention Forward Calculation

Setup: Consider a single query row producing 4 attention scores split into two key/value tiles of size 2:

- Tile 1: scores $S^{(1)} = [1.0,\ 2.0]$, scalar values $V^{(1)} = [10,\ 20]^T$.
- Tile 2: scores $S^{(2)} = [4.0,\ 0.0]$, scalar values $V^{(2)} = [30,\ 40]^T$.

Compute the output O using tiled online softmax and verify that it matches full-row safe softmax.

Steps:

1. **Process Tile 1**:
   - $m^{(1)} = \max(1.0, 2.0) = 2.0$.
   - $\tilde{P}^{(1)} = [e^{1.0-2.0},\ e^{2.0-2.0}] = [e^{-1},\ 1] \approx [0.3679,\ 1.0000]$.
   - $\ell^{(1)} = 0.3679 + 1.0000 = 1.3679$.
   - $\tilde{O}^{(1)} = 0.3679(10) + 1.0000(20) = 23.6788$.

2. **Process Tile 2**:
   - $m^{(2)} = \max(m^{(1)},\ \max(4.0, 0.0)) = \max(2.0, 4.0) = 4.0$.
   - Rescaling factor: $\alpha = e^{m^{(1)} - m^{(2)}} = e^{2.0-4.0} = e^{-2} \approx 0.1353$.
   - $\tilde{P}^{(2)} = [e^{4.0-4.0},\ e^{0.0-4.0}] = [1,\ e^{-4}] \approx [1.0000,\ 0.0183]$.
   - Update denominator:

     $$\ell^{(2)} = \alpha \ell^{(1)} + (1.0000 + 0.0183) = 0.1353(1.3679) + 1.0183 = 0.1851 + 1.0183 =$$

   - Update running output:

     $$\tilde{O}^{(2)} = \alpha \tilde{O}^{(1)} + \big(1.0000(30) + 0.0183(40)\big) = 0.1353(23.6788) + 30.7326 = 3.20$$

3. **Final epilogue**:

$$O = \frac{\tilde{O}^{(2)}}{\ell^{(2)}} = \frac{33.9372}{1.2034} \approx 28.20$$

4. **Verification against full-row safe softmax**:
   - Global max $m = 4.0$. Shifted exponentials across all 4 keys: $[e^{-3}, e^{-2}, e^{0}, e^{-4}] \approx [0.0498, 0.1353, 1.0000, 0.0183]$.
   - Sum $\ell = 0.0498 + 0.1353 + 1.0000 + 0.0183 = 1.2034$.
   - Numerator $= 0.0498(10) + 0.1353(20) + 1.0000(30) + 0.0183(40) = 0.4979 + 2.7067 + 30.0000 + 0.7326 = 33.9372$.
   - Ratio $= 33.9372 / 1.2034 \approx 28.20$ — **exact match**.

Interpretation: Notice how $\alpha \cdot \tilde{O}^{(1)} = e^{-2}(e^{-1} \cdot 10 + e^{0} \cdot 20) = e^{-3} \cdot 10 + e^{-2} \cdot 20$ retroactively re-anchored Tile 1 to the global max 4.0 without ever storing or re-reading $S^{(1)}$!

Between **FlashAttention-1 (2022)** and **FlashAttention-2 (2023)**, three refinements sped up this exact math: FA2 keeps the unnormalized $\tilde{O}$ throughout the inner loop (dividing by ℓ only once at the very end, as written above, rather than rescaling by $\ell^{(t-1)}/\ell^{(t)}$ at every iteration), parallelizes across the sequence-length dimension, and minimizes non-matmul FLOPs.

COMMON MISCONCEPTION

You might think FlashAttention is an *approximate* attention algorithm (like sparse or linear attention) that trades away accuracy for speed. That is completely false: FlashAttention computes **mathematically exact** standard attention (up to floating-point rounding) solely by reordering computations through tiled online softmax and backward recomputation.

FlashAttention-3 (Hopper, 2024): Asynchronous Attention

FlashAttention-3 keeps the exact same online-softmax math as FA2, but rebuilds the kernel around Hopper's asynchronous primitives from Section 6. Look at its steady-state mainloop timeline across three warp groups:

- **1 Producer Warp**: Continuously issues `SM90_TMA_LOAD` to stream `K1, V1`, `K2, V2`, `K3, V3`, … into a multi-stage SMEM ring buffer.
- **Consumer Warpgroup 0** (4 warps, 128 threads): Executes `wgmma QK1`, then `Softmax 1`, then `wgmma P1V1`, then `wgmma QK3`, then `Softmax 3`.
- **Consumer Warpgroup 1** (part of 1 to 3 MMA warpgroups): Runs in **ping-pong** phase offset from WG 0—while WG 0 runs `Softmax 1`, WG 1 executes `wgmma QK2`; while WG 0 runs `wgmma P1V1`, WG 1 runs `Softmax 2`.

By ping-ponging consumer warpgroups, FA3 **overlaps the non-matmul softmax of one tile directly with the Tensor Core `wgmma` of the next tile**, hiding softmax latency behind matrix multiplication. FA3 also adds **FP8 support** using block quantization and incoherent processing to maintain numerical accuracy.

FlashAttention-4 (Blackwell SM100, 2025–2026): Dedicated Warp Roles and Roofline Innovations

On Blackwell SM100, FlashAttention-4 (written in **CuTe DSL**) exploits `tcgen05`, TMEM, and `cta_group::2` to push warp specialization from 2 roles to **5 fine-grained roles** inside a single forward kernel:

1. **Load Warp**: Uses TMA to load Q, K, V tiles into a swizzled SMEM ring buffer.
2. **MMA Warp (1 Thread!)**: A single thread issues `tcgen05` with `cta_group::2` (2-CTA cooperative MMA) for both QK^T and PV, writing score tiles S_0, S_1 and accumulating output tiles O_0, O_1 directly inside **TMEM**.
3. **2 Softmax Warpgroups (Ping-Pong)**: Softmax WG 0 operates on Q Tile 0 while Softmax WG 1 operates on Q Tile 1, reading finished score tiles from TMEM.
4. **Correction Warps**: Dedicated warps that rescale the running output accumulators O_0, O_1 in TMEM—and only when a threshold is crossed!
5. **Epilogue**: Normalizes and writes the final output O from TMEM out to HBM.

Because accumulators live in **TMEM** and only **1 thread** launches MMAs, the softmax and correction warps enjoy abundant general-purpose registers and minimal scheduling stalls.

Even more revealing are the **two new roofline-driven innovations in FlashAttention-4**. Once Blackwell's 5th-generation Tensor Cores make QK^T and PV matmuls fast enough, **non-matmul operations inside softmax become the new bottleneck**:

1. **Moving `exp2` off the SFU (B200 vs. B300)**:
   - On **B200**, Tensor Cores outrun the hardware Special-Function Unit (**MUFU**), so computing `exp2` in softmax stalls the entire attention pipeline. FA4 solves this by emulating a portion of the `exp2`

evaluations using a **polynomial on ordinary FMA units** (CUDA Cores), splitting the exponential workload across both the MUFU and the FMA units.

- On **B300 (SM103)**, upgraded faster SFUs remove the MUFU bottleneck, so FA4 switches back to pure hardware `exp2`. This is a textbook example of hardware microarchitecture directly dictating algorithm implementation.

2. **Conditional (Lazy) Rescaling**:
   - Standard online softmax rescales the entire running output tile $\tilde{O}$ every single time the row maximum m increases—even if m only increased by 0.01. That triggers expensive correction work in TMEM on almost every tile.
   - FA4 lets the running reference maximum lag behind the true maximum by a **threshold**, rescaling $\tilde{O}$ **only when** m **jumps past that threshold**. As long as the new scores minus the lagging reference do not overflow, skipping the rescale keeps the final output **mathematically exact** because $\tilde{O}$ and ℓ track the exact same consistent reference exponent!

DEFINITION

Conditional (lazy) rescaling in FlashAttention-4 defers rescaling the running output accumulator $\tilde{O}$ whenever a new tile increases the row maximum by less than a chosen threshold, keeping both $\tilde{O}$ and ℓ anchored to the same lagging reference maximum so the final ratio $\tilde{O}/\ell$ remains mathematically exact while skipping most correction work.

WORKED EXAMPLE

Worked Example: Exactness of Conditional (Lazy) Rescaling

Setup: Suppose Tile 1 had row max $m_{\text{ref}} = 2.0$, giving $\tilde{P}^{(1)} = e^{S^{(1)}-2.0}$, $\ell^{(1)} = \sum \tilde{P}^{(1)}$, and $\tilde{O}^{(1)} = \tilde{P}^{(1)}V^{(1)}$. Now Tile 2 arrives with max 2.5, and our lazy-rescaling threshold is $\tau = 1.0$. Because $2.5 - 2.0 = 0.5 < 1.0$, we skip rescaling $\tilde{O}^{(1)}$! Verify that the final attention output stays exact.

Steps:

1. Keep the old reference $m_{\text{ref}} = 2.0$ instead of updating to 2.5.
2. Compute Tile 2's exponentials relative to $m_{\text{ref}} = 2.0$:

$$\tilde{P}^{(2)}_{\text{lazy}} = e^{S^{(2)}-2.0}$$

   Because the largest exponent is $2.5 - 2.0 = +0.5$ ($e^{0.5} \approx 1.65$), there is zero risk of floating-point overflow.
3. Accumulate directly with **zero rescaling** ($\alpha = 1$):

$$\ell^{(2)}_{\text{lazy}} = \ell^{(1)} + \sum e^{S^{(2)}-2.0}, \qquad \tilde{O}^{(2)}_{\text{lazy}} = \tilde{O}^{(1)} + \tilde{P}^{(2)}_{\text{lazy}} V^{(2)}$$

4. Take the final ratio:

$$O = \frac{\tilde{O}^{(2)}_{\text{lazy}}}{\ell^{(2)}_{\text{lazy}}} = \frac{e^{-2.0}\left(e^{S^{(1)}}V^{(1)} + e^{S^{(2)}}V^{(2)}\right)}{e^{-2.0}\left(\sum e^{S^{(1)}} + \sum e^{S^{(2)}}\right)} = \frac{e^{S^{(1)}}V^{(1)} + e^{S^{(2)}}V^{(2)}}{\sum e^{S^{(1)}} + \sum e^{S^{(2)}}}$$

Interpretation: The common factor $e^{-m_{\text{ref}}}$ cancels identically between numerator and denominator! Rescaling is only needed when m jumps enough to threaten overflow, allowing FA4's correction warps to stay idle on most iterations without losing exactness.

In 2026 production frontier models supporting 1M-token contexts—including **DeepSeek-V4.1-Flash**, **Kimi K3**, and **MiniMax-M3**—sparse and compressed attention architectures are paired directly with these exact Blackwell kernel primitives.

KEY IDEA

FlashAttention eliminates $O(N^2)$ HBM traffic for exact attention by fusing tiled QK^T, online softmax, and PV in on-chip SRAM and recomputing scores during backward. FlashAttention-3 (Hopper) overlaps softmax with `wgmma` via TMA and warp specialization, while FlashAttention-4 (Blackwell) deploys 5 warp roles over `tcgen05`, 2-CTA MMAs, and TMEM—solving the newly exposed non-matmul softmax bottleneck via FMA-emulated `exp2` (on B200) and conditional lazy rescaling.

Building a fast kernel like FlashAttention-4 solves peak throughput for large compute-bound operations. Yet deploying kernels inside real 2026 training and inference systems exposes three additional engineering challenges: keeping numerical outputs reproducible across varying batch sizes, eliminating kernel-launch gaps at small decode batch sizes, and choosing the right programming abstraction to build and maintain kernels. We close the lecture with those three production topics.

✎ Check Your Understanding

1. **What would happen if…**: You take FlashAttention-4 tuned for Blackwell B200 (which offloads part of `exp2` onto FMA units via polynomial emulation) and run it on a B300 (SM103) without changing that setting. Why does FA4 switch back to hardware `exp2` on B300?

   Answer: B300 upgrades its Special-Function Units (MUFUs) enough to remove the `exp2` bottleneck; continuing to emulate `exp2` on FMA units would unnecessarily spend general FMA instructions when native hardware `exp2` is already fast enough to keep up with Tensor Cores.

2. **Yes or No (with justification)**: Does FlashAttention-1 perform fewer total FLOPs than standard PyTorch attention during a combined forward and backward pass?

   Answer: No. FlashAttention performs *more* FLOPs (75.2 vs. 66.6 GFLOPs on the historical GPT-2 benchmark) because it recomputes attention scores tile-by-tile during the backward pass, yet runs dramatically faster because it cuts HBM reads/writes nearly $10\times$.

Section 8: Production Kernel Engineering: Determinism, Megakernels, and the Modern DSL Landscape

Peak TFLOP/s on an isolated benchmark is not the whole story of GPU systems engineering. Three practical realities separate research prototypes from reliable 2026 production stacks: **numerical batch invariance**, **small-batch decode launch overhead**, and **kernel developer productivity across DSLs**.

Batch-Invariant (Deterministic) Kernels

Floating-point addition is **non-associative**:

$$(a + b) + c \neq a + (b + c)$$

Because standard high-performance GPU kernels optimize aggressively for wave utilization (Section 5), they dynamically change **Split-K factors, reduction orders, or tile shapes** whenever the batch size changes. The startling consequence in production inference and training is that **the exact same prompt produces different output logits depending on what other requests happen to sit in the same batch!**

WORKED EXAMPLE

Worked Example: How Dynamic Split-K Changes a Floating-Point Sum

Setup: Suppose a dot product accumulates three terms: $a = 1.0 \times 10^8$, $b = -1.0 \times 10^8$, and $c = 1.0$, in a floating-point format with 7 decimal digits of precision (comparable to FP32's ~7.2 decimal digits, where $10^8 + 1.0$ rounds to 10^8). Compare two reduction orders caused by changing Split-K between batch size 1 and batch size 4.

Steps:

1. **Batch size 1 (Split-K = 1, left-to-right sequential sum)**:

$$(a + b) + c = (10^8 + (-10^8)) + 1.0 = 0.0 + 1.0 = \mathbf{1.0}$$

2. **Batch size 4 (different split/grouping reduces a with c first)**:

$$b + (a + c) = -10^8 + \text{fl}(10^8 + 1.0) = -10^8 + 10^8 = \mathbf{0.0}$$

Interpretation: Identical inputs yield 1.0 under one reduction tree and 0.0 under another. Across millions of multiply-adds in a Transformer layer, changing Split-K or tile reduction order with batch size perturbs logits enough to flip sampled tokens.

In September 2025, work by **Thinking Machines** traced batch nondeterminism specifically to **matmul, RMSNorm, and attention**—and notably *not* to floating-point atomics.

COMMON MISCONCEPTION

You might think batch nondeterminism on GPUs is caused by race conditions in floating-point atomic additions (`atomicAdd`). That breaks down under investigation: even completely atomic-free kernels produce batch-dependent logits whenever the runtime selects different Split-K factors, tile shapes, or reduction trees for different batch sizes.

The fix is **batch-invariant kernels**: kernels engineered to enforce a **fixed reduction tree and summation order** for every output element regardless of batch size.

DEFINITION

A **batch-invariant (deterministic) kernel** enforces an identical floating-point reduction tree and summation order for each output element regardless of the enclosing batch size, guaranteeing bit-identical outputs for a given input across different batches.

Pinning reduction order comes with a performance trade-off because the kernel can no longer freely choose arbitrary Split-K splits for small batches. Early implementations incurred tens of percent of overhead—for example, **~34% overhead** in SGLang's deterministic mode with CUDA graphs—while more recent research (**Detent**, self-reported) has narrowed the overhead down to 1.04–$1.17\times$ **latency**. In 2026, batch invariance has become mission-critical for **reinforcement learning (RL) post-training** (used in frontier models such as **DeepSeek-V4.1-Flash** and **Nemotron 3.5**), where policy optimization requires rollout log-probabilities and trainer log-probabilities to match accurately despite running at different batch sizes.

Megakernels: Removing Launch Gaps at Small Batch

Look at the timeline of a **standard LLM decode step** running at small batch size:

`RMSNorm` → *gap* → `QKV Proj` → *gap* → `Attention` → *gap* → `O Proj` → *gap* → `MLP / MoE`

Between every consecutive pair of discrete kernels sit three compounding overheads: **CPU-side launch latency**, **inter-kernel wave drains** (waiting for the slowest tail SM of one kernel to finish before launching the next), and **HBM synchronization barriers**. Even with CUDA Graphs removing CPU launch latency, discrete kernels still drain the GPU pipeline at every operator boundary.

A **megakernel** collapses an entire Transformer layer—or an entire decode step—into **one persistent kernel** driven by an **on-GPU task scheduler** (an on-GPU dependency queue). Persistent CTAs stay resident across `Norm`, `QKV Proj`, `Attention`, `O Proj`, and `MLP / MoE`, pulling ready tasks dynamically as soon as their data dependencies clear, with **zero launch gaps**.

> **DEFINITION**
> A **megakernel** collapses an entire sequence of heterogeneous neural network layers (such as a full decode step) into a single persistent GPU kernel managed by an on-GPU dependency task scheduler, eliminating inter-kernel launch gaps and wave-drain barriers.

Megakernels are constructed either automatically by specialized compilers (such as **Mirage MPK**) or hand-written per model architecture. Understanding *when* they win follows directly from the roofline and wave-occupancy analysis of earlier sections:

- **At batch 1**: Matrix multiplications are skinny and short, so launch latency and inter-kernel wave drains dominate wall-clock time—megakernels deliver their **largest speedups** here.
- **As batch size grows**: Matmuls expand to saturate all 148 SMs for long durations, making inter-kernel gaps a tiny fraction of total runtime—so megakernel gains **shrink steadily at larger batches**.

The Modern GPU Kernel DSL Landscape (2026)

Writing kernels for Blackwell spans a clean spectrum from automatic graph compilers down to explicit warp-role hardware control:

Tool / Language	Abstraction Level	Key Strengths & Examples
CUDA C++ / CUTLASS	Lowest (C++ / PTX)	Full hardware control; persistent GEMM & CLC templates
CuTe DSL	Low-level Python DSL	Explicit layouts, TMA, `tcgen05`, TMEM; used to write **FlashAttention-4**
TileLang	Middle-level tile DSL	Tile-centric programming with explicit pipelining; used by **DeepSeek's TileKernels**
Triton (+ Gluon)	High-level block DSL	Productive Python kernels; **Gluon** dialect exposes lower-level layouts & async ops
NVIDIA cuTile	Tile-based model	NVIDIA's tile-level programming model for Tensor Core GPUs
`torch.compile`	Graph compiler	Automatic pointwise & reduction fusion via TorchInductor

The practical engineering rule of thumb is disciplined escalation: **start with `torch.compile` and Triton** for rapid development and automatic fusion, and drop down to **TileLang, CuTe DSL, or CUTLASS/CUDA C++** only for the hot bottleneck kernels (like FlashAttention or core quantized GEMMs) where explicit warp specialization, TMEM placement, and `tcgen05` scheduling move end-to-end wall-clock time.

> **KEY IDEA**
> Production GPU engineering balances peak throughput against numerical reproducibility, small-batch launch efficiency, and engineering effort: **batch-invariant kernels** fix reduction trees to eliminate batch-dependent logits for RL post-training, **megakernels** eliminate launch and wave-drain gaps at small decode batch sizes, and modern **kernel DSLs** (`torch.compile`, Triton/Gluon, cuTile, TileLang, CuTe DSL, CUTLASS/CUDA) let engineers match abstraction level to the bottleneck. Across the entire lecture, three enduring principles govern Blackwell performance: **cut HBM traffic**, **keep Tensor Cores saturated**, and **move non-matmul work off the critical path**.

✎ Check Your Understanding

1. **Yes or No (with justification)**: If a serving stack replaces discrete decode kernels with a megakernel, will the relative speedup be larger at batch size 1 or at batch size 256?

   Answer: Batch size 1. At batch 1, launch overhead and inter-kernel wave drains dominate total runtime, so eliminating them yields the largest gains; at batch 256, dense matmuls already saturate all SMs and dwarf the inter-kernel gaps.

2. **What would happen if…**: You enable a standard, non-batch-invariant GEMM kernel with dynamic Split-K during RL post-training, where rollouts run at a different batch size than the trainer's logprob pass. What numerical issue occurs, and why?

   Answer: The rollout and trainer passes will produce slightly different log-probabilities for the exact same tokens, because dynamic Split-K changes the floating-point summation order across batch sizes and floating-point addition is non-associative.