Lecture 6: Benchmarking, Profiling, and GPU Kernels

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

Section 1: The GPU Memory Hierarchy and Software-to-Hardware Mapping

Writing fast numerical code for modern language models requires solving two distinct problems: making a computation mathematically correct and making it execute near the physical limits of the processor. A central organizing principle governs every technique in this lecture: the **programming model** gives you correctness, while the **hardware model** gives you speed. You can write a perfectly correct GPU program using only high-level abstractions and still leave most of the machine's performance unused. Closing that gap requires an accurate mental model of where data lives, how fast it moves, and how parallel work is assigned to physical compute units.

The overall engineering workflow follows a four-stage progression:

1. **GPU Model:** Understand the hardware architecture—specifically the NVIDIA Blackwell (B200) memory hierarchy, streaming multiprocessors (SMs), warps, and occupancy.
2. **Measure:** Never guess where time is spent; benchmark end-to-end latency, profile individual kernel executions, and verify numerical invariance.
3. **Kernel DSLs:** Select the appropriate authoring abstraction across the modern spectrum—CUDA, Triton, Gluon, TileLang, NVIDIA cuTile, and CuTe DSL.
4. **Write Kernels:** Apply the core optimization levers—**kernel fusion**, **tiling**, and **cutting launch overhead**—to build GeLU, softmax, row sum, and tiled matrix multiplication kernels.

We begin with the first stage: the memory hierarchy of the NVIDIA Blackwell B200 GPU.

A GPU contains thousands of arithmetic units organized into **Streaming Multiprocessors (SMs)**, and every computation requires moving operands from memory to those arithmetic units and writing results back. The fundamental bottleneck in modern GPU computing is that memory capacity and memory bandwidth trade off sharply against each other. Large memories are slow; fast memories are tiny.

> **DEFINITION**
> **Streaming Multiprocessor (SM):** An independent processing unit on a GPU that executes groups of threads in parallel and contains its own private on-chip registers, tensor cores, L1 cache, and shared memory. The NVIDIA Blackwell B200 has 148 SMs.

Picture the B200 memory hierarchy as a vertical stack of tiers, moving from the smallest, fastest storage inside each SM down to the largest, slowest storage shared across the entire chip:

- **Inside each SM (148 SMs on B200):**

  - **Registers (top tier):** 256 KB per SM, delivering roughly **~447 TB/s** of aggregate bandwidth across the GPU. Registers are private to individual execution lanes and sit directly adjacent to the arithmetic units.
  - **Tensor Memory (TMEM):** A dedicated on-chip memory tier introduced on Blackwell, positioned between registers and shared memory specifically to feed the tensor cores during matrix operations.
  - **L1 Cache + Shared Memory:** 256 KB per SM, delivering roughly **~40 TB/s** of aggregate bandwidth (rough estimate). Unlike hardware-managed caches, shared memory is software-managed scratchpad memory accessible to cooperating threads on the same SM.
- **Shared across the entire GPU (outside individual SMs):**
  - **L2 Cache:** 96–126 MB shared across all 148 SMs, with roughly **~17 TB/s** of bandwidth (rough estimate).
  - **HBM (High Bandwidth Memory / Global Memory, bottom tier):** 192 GB of capacity at **8 TB/s** of bandwidth. HBM holds model weights, activations, and optimizer states, making it the largest and slowest tier on the GPU.

Note that the L1/shared-memory and L2 bandwidth numbers are rough estimates, whereas HBM bandwidth and all storage capacities are architectural specifications. Look at the gap between the top and bottom of this stack: aggregate register bandwidth (~447 TB/s) is more than 55 times higher than HBM bandwidth (8 TB/s), and shared-memory bandwidth (~40 TB/s) is roughly 5 times higher than HBM bandwidth. Consequently, fast kernels must keep data on-chip—in registers and shared memory—and minimize round-trips to HBM.

Comparing the current B200 against historical GPU generations (A100 and H100) reveals a hardware trend that directly shapes kernel design:

Specification	A100 (Historical)	H100 (Historical)	B200 (Current)
# Streaming Multiprocessors (SMs)	108	132	**148**
Register file (per SM)	256 KB	256 KB	**256 KB**
L1 cache + shared memory (per SM)	192 KB	256 KB	**256 KB**
L2 cache size	40 MB	50 MB	**96–126 MB**
HBM capacity	80 GB	80 GB	**192 GB**
Register bandwidth	~116 TB/s	~401 TB/s	**~447 TB/s**
L1 + shared memory bandwidth	~19 TB/s	~33 TB/s	**~40 TB/s**
L2 cache bandwidth	~5–8 TB/s	~12 TB/s	**~17 TB/s**
HBM bandwidth	2 TB/s	3.35 TB/s	**8 TB/s**

Look closely at the second and third rows of this table. From A100 to H100 to B200, HBM capacity more than doubled (80 GB to 192 GB) and HBM bandwidth quadrupled (2 TB/s to 8 TB/s), while the L2 cache grew from 40 MB to 96–126 MB. Yet the per-SM register file remained fixed at **256 KB**, and per-SM L1 cache plus shared memory stayed at **256 KB** from H100 to B200.

COMMON MISCONCEPTION

You might think that because B200 jumps to 192 GB of HBM and 8 TB/s of HBM bandwidth, on-chip memory constraints have relaxed and you can hold much larger working tiles per SM than on earlier GPUs. That breaks because per-SM register capacity (256 KB) and per-SM L1/shared memory (256 KB) did not grow from H100 to B200. On-chip tile budgets remain just as tight, and the ratio between register bandwidth (~447 TB/s) and HBM bandwidth (8 TB/s) still makes every unnecessary HBM trip expensive.

WORKED EXAMPLE

Suppose a kernel operates on 32-bit floating-point (`float32`) matrices, where each element occupies 4 bytes. How many elements can fit simultaneously inside one SM's entire 256 KB L1/shared memory pool?

1. Convert 256 KB to bytes:

$$256 \times 1{,}024 = 262{,}144 \text{ bytes}.$$

2. Divide by 4 bytes per `float32` element:

$$\frac{262{,}144}{4} = 65{,}536 \text{ elements}.$$

3. Suppose we need to hold two square tiles of equal size (one from matrix A and one from matrix B, as we will do in Section 7). Each tile gets at most $32{,}768$ elements. For a 128×128 tile ($16{,}384$ elements = 64 KB), two tiles take 128 KB—half of the SM's total L1 + shared memory budget. A 256×256 tile ($65{,}536$ elements = 256 KB) fills the entire SM alone, leaving no room for the second tile.

Now compare transfer time for 1 GB (10^9 bytes) of data at peak bandwidth across tiers on B200:

- At HBM (8 TB/s = 8×10^{12} B/s): $\frac{10^9}{8 \times 10^{12}} = 0.125 \text{ ms} = 125\ \mu\text{s}$.
- At L1/shared memory (~40 TB/s): $\frac{10^9}{40 \times 10^{12}} = 25\ \mu\text{s}$ (5× faster).
- At registers (~447 TB/s): $\frac{10^9}{447 \times 10^{12}} \approx 2.24\ \mu\text{s}$ (~56× faster).

Keeping intermediate values in registers or shared memory instead of round-tripping them through HBM turns a $125\ \mu$s memory bottleneck into a few microseconds of on-chip traffic.

To exploit this memory hierarchy in software, GPU programming models organize parallel work into a three-level software hierarchy that maps one-to-one onto the hardware scopes just described:

- **Thread (software) → Execution Lane (hardware):** A single thread executes kernel code on a slice of data, executing scalar instructions with its own **private registers**.
- **Thread Block / Cooperative Thread Array, or CTA (software) → Single SM (hardware):** A group of cooperating threads scheduled together onto a single SM. Because all threads in a block reside on the same SM, they share that SM's **shared memory** and can perform fast local synchronization.
- **Grid (software) → Entire GPU (hardware):** The collection of all thread blocks launched by a single kernel invocation, organized in 1D, 2D, or 3D coordinates. Blocks in a grid communicate through **HBM (global memory)**.

> **DEFINITION**
> **Thread Block (CTA):** A group of threads launched together and scheduled onto a single SM, allowing them to exchange data through that SM's shared memory and synchronize locally without going to HBM.

> **DEFINITION**
> **Grid:** The complete 1D, 2D, or 3D collection of thread blocks launched by one GPU kernel invocation across the entire GPU.

On H100 and B200, NVIDIA also supports an intermediate grouping called **thread block clusters**, which group multiple thread blocks across different SMs to enable **distributed shared memory**—allowing cooperating blocks to access each other's SM-local shared memory directly.

> **DEFINITION**
> **Thread Block Cluster:** A grouping of thread blocks spanning multiple SMs on H100 and B200 GPUs that enables direct inter-SM data exchange via distributed shared memory.

Why does the thread-block abstraction exist at all, instead of exposing only a flat grid of independent threads? Consider two classes of operations:

1. **Elementwise operations** (such as GeLU): each output $y_i = f(x_i)$ depends solely on its corresponding input x_i. Assigning one thread per element (or per small chunk of elements) requires zero communication between threads. A flat list of threads would suffice.
2. **Non-elementwise operations** (such as softmax or matrix multiplication): computing each output requires combining values across many inputs, so threads must exchange intermediate results. If threads lived independently across the GPU, they would have to exchange those intermediates through HBM at 8 TB/s. Placing collaborating threads inside the same **thread block** pins them to the same SM, so they can read and write the same SM-local shared memory at ~40 TB/s instead.

We will rely directly on this thread-block abstraction when we move to **Triton** in Section 5, where you write code at the thread-block level rather than specifying individual threads.

KEY IDEA

The B200 memory hierarchy spans nearly two orders of magnitude in bandwidth, from ~447 TB/s in per-SM registers and ~40 TB/s in SM-local shared memory down to 8 TB/s in HBM, while per-SM on-chip storage stays fixed at 256 KB. The software hierarchy (thread → thread block → grid) mirrors these hardware scopes so cooperating threads can stay on a single SM and communicate through shared memory rather than HBM.

Knowing how threads, thread blocks, and grids map onto lanes, SMs, and the full GPU is enough to reason about *where* data lives and write a *correct* kernel. However, a kernel that respects this memory hierarchy can still lose most of its speed if it violates how the SM schedules threads, allocates registers, services shared-memory accesses, fetches HBM cache lines, and distributes blocks across SMs. Those hardware execution mechanisms are what we examine next.

✎ Check Your Understanding

1. Fill in the blank: In the GPU software-to-hardware mapping, a **thread** maps to an execution lane with private registers, a **grid** maps to the entire GPU communicating via HBM, and a **thread block (CTA)** maps to a single _____ sharing its _____.
   Answer: A thread block (CTA) maps to a single **SM (Streaming Multiprocessor)** sharing its **shared memory**.
2. Does an elementwise activation like GeLU ($y_i = f(x_i)$) require threads to communicate through shared memory? Justify your answer in one to two sentences.
   Answer: No. In an elementwise operation each output y_i depends only on its own input x_i, so threads need zero inter-thread communication; shared memory is needed for non-elementwise operations like softmax and matrix multiplication where threads must exchange intermediate values.

Section 2: Four Hardware Execution Realities That Govern Speed

The programming model presents a clean illusion: thousands of independent threads executing code, a flat global address space in HBM, a uniform scratchpad in shared memory, and an unlimited pool of thread blocks. The physical GPU does not execute that illusion literally. Five concrete hardware realities—spanning warps, registers, shared-memory banks, HBM cache lines, and SM waves—determine whether a correct kernel runs at peak hardware speed or stalls at a small fraction of it.

1. Warps, Control Divergence, and Latency Hiding

Inside an SM, threads are not scheduled one by one. Hardware groups every 32 consecutive threads within a thread block into a **warp** that executes the exact same instruction in lockstep. For example, a thread block of 64 threads is partitioned by the hardware into two warps: Warp 0 containing threads `T0`

through `T31`, and Warp 1 containing threads `T32` through `T63`.

> **DEFINITION**
> **Warp:** A hardware scheduling unit of 32 threads within a thread block that executes the same instruction in lockstep on an SM.

Lockstep execution creates a hazard whenever threads inside the same warp take different branches of a conditional statement such as `if (cond) A else B`. Because all 32 lanes of the warp must execute the same instruction at the same clock cycle, the hardware cannot execute `A` on some threads and `B` on others simultaneously. Instead, it serializes the branches across two passes:

- **Pass 1 (branch `A`):** The lanes where `cond` is true execute `A`, while all remaining lanes in the warp are **masked off** and sit idle.
- **Pass 2 (branch `B`):** The first group of lanes is now masked off and sits idle, while the lanes where `cond` is false execute `B`.

The total execution time for the warp is therefore:

$$\text{time}_{\text{warp}} = \text{time}(A) + \text{time}(B).$$

> **DEFINITION**
> **Control Divergence:** The serialization of execution inside a warp when different threads in the same 32-thread warp follow different conditional paths, forcing the hardware to run each path sequentially with non-participating lanes masked off.

Note that divergence only hurts *within* a warp. If all 32 threads of Warp 0 take branch `A` and all 32 threads of Warp 1 take branch `B`, neither warp diverges—each executes only its own branch.

Warps also provide the GPU's primary mechanism for coping with slow HBM accesses: **latency hiding**. Fetching data from HBM takes many cycles. Rather than sitting idle while one warp waits for an HBM load to return, an SM keeps many warps resident at the same time and switches to another ready warp with **zero cost**. As long as the SM holds enough ready warps to keep its arithmetic pipelines busy during memory stalls, HBM latency is hidden behind useful work.

2. Register Pressure, Warp Occupancy, and Thread Coarsening

Zero-cost warp switching only works if enough warps actually fit on an SM simultaneously. What limits how many warps can reside on an SM? On B200, each SM provides:

- **65,536** 32-bit registers (256 KB total, since $65{,}536 \times 4 \text{ bytes} = 262{,}144 \text{ bytes} = 256 \text{ KB}$).
- Hardware slots for up to **64 concurrent warps**.
- A per-thread limit of **0 to 255 registers**.

If a kernel keeps many intermediate values live in registers per thread, each thread block consumes a large slice of the 65,536-register pool. Fewer blocks can fit on the SM at once, which reduces the number of resident warps—a metric called **warp occupancy**.

DEFINITION

Warp Occupancy: The ratio of active concurrent warps resident on an SM to the hardware maximum supported by that SM (64 warps per SM on B200).

Formally, when register usage is the limiting resource for a kernel with T_{block} threads per block (forming $W_{\text{block}} = T_{\text{block}}/32$ warps per block) and R_{thread} registers per thread:

$$\text{Regs / block} = T_{\text{block}} \times R_{\text{thread}}$$

$$\text{Blocks / SM} = \left\lfloor \frac{65{,}536}{\text{Regs / block}} \right\rfloor$$

$$\text{Active warps / SM} = (\text{Blocks / SM}) \times W_{\text{block}}$$

$$\text{Occupancy} = \frac{\text{Active warps / SM}}{64}.$$

Each subexpression tracks a physical resource constraint: the first line computes the total register footprint of one thread block; the floor division in the second line counts how many whole blocks fit inside the SM's 65,536-register file (partial blocks cannot be scheduled); the third line converts resident blocks into resident warps; and the final ratio expresses that concurrency as a fraction of the SM's 64-warp ceiling.

WORKED EXAMPLE

Compute the register-limited warp occupancy on a B200 SM for a kernel with a block size of 128 threads (4 warps) that uses 160 registers per thread.

1. **Registers per block:**

$$\text{Regs / block} = 128 \times 160 = 20{,}480 \text{ registers.}$$

2. **Resident blocks per SM:**

$$\text{Blocks / SM} = \left\lfloor \frac{65{,}536}{20{,}480} \right\rfloor = \lfloor 3.2 \rfloor = 3 \text{ blocks.}$$

3. **Active warps per SM:**

$$\text{Active warps} = 3 \times 4 = 12 \text{ warps.}$$

4. **Warp occupancy:**

$$\text{Occupancy} = \frac{12}{64} = 0.1875 = \mathbf{18.75\%}.$$

Even though the SM has 64 hardware warp slots available, register pressure caps concurrency to just 3 blocks (12 warps) per SM, leaving 52 warp slots unused.

COMMON MISCONCEPTION

You might think that 100% warp occupancy is always optimal and that low occupancy like 18.75% automatically means a slow kernel. That breaks because of **thread coarsening**: if each thread uses more registers to hold more values and perform more independent work per thread, the SM can still saturate its arithmetic units and hide latency with fewer active warps while amortizing address arithmetic and loop overhead.

DEFINITION

Thread Coarsening: Assigning multiple output elements or larger chunks of work to each thread, which increases per-thread register usage and may lower occupancy, but can improve overall performance by amortizing index calculations and increasing per-thread instruction-level work.

We will see the Triton compiler apply thread coarsening automatically when we inspect generated PTX assembly in Section 5.

3. Shared Memory Bank Conflicts and Swizzling

Even when data resides in SM-local shared memory (~40 TB/s), poor access patterns can slash shared-memory throughput by up to 32×. Shared memory is physically partitioned into **32 banks**, labeled `B00` through `B31`, each **4 bytes wide**. Consecutive 4-byte words are striped round-robin across these 32 banks: word 0 lives in `B00`, word 1 in `B01`, …, word 31 in `B31`, word 32 wraps back to `B00`, and so on. Each bank can service **one address per clock cycle**.

Consider storing a 32×32 matrix M of 4-byte floats in shared memory in standard row-major order:

- **Row access (conflict-free):** Row 0 holds `M[0,0]` in `B00`, `M[0,1]` in `B01`, `M[0,2]` in `B02`, …, `M[0,31]` in `B31`. When a 32-thread warp reads row 0 simultaneously, thread t accesses bank `B`$_t$. Every thread hits a distinct bank, so all 32 banks respond in parallel in **1 cycle**.
- **Column access (32-way bank conflict):** Now look down column 0. Because each row has 32 elements, `M[0,0]` is in `B00`, `M[1,0]` is 32 words later—also in `B00` — `M[2,0]` is in `B00`, and all the way down to `M[31,0]` in `B00`. When a 32-thread warp reads column 0 simultaneously, all 32 threads request 32 different addresses from the single bank `B00`. Because `B00` can serve only one address per cycle, the hardware serializes the request into a **32-way bank conflict**, taking 32 cycles instead of 1.

DEFINITION

Shared Memory Bank Conflict: A stall that occurs when multiple threads in the same warp simultaneously access different addresses that map to the same 4-byte shared-memory bank, forcing the hardware to serialize those accesses across multiple cycles.

This conflict is unavoidable in naive matrix multiplication AB because multiplying tiles requires reading rows of A and columns of B. The standard solution is **swizzling**: permuting the column index inside shared memory as a function of the row index—specifically computing the physical column (and therefore bank) using bitwise exclusive-OR:

$$\text{bank} = \text{row} \oplus \text{col} \quad (\text{written } \texttt{row ^ col}).$$

DEFINITION

Swizzling: Permuting the storage layout of a 2D array in shared memory (for example, storing logical element (row, col) at physical column $\text{row} \oplus \text{col}$) so that both row accesses and column accesses map to distinct memory banks without conflicts.

WORKED EXAMPLE

Trace the swizzled mapping $\text{bank} = \text{row} \oplus \text{col}$ on the top-left 4×4 corner of matrix M (rows 0–3, columns 0–3, using 2-bit binary XOR):

- **Row 0** ($\text{row} = 00_2$): `M[0,0]` $\to 0 \oplus 0 = \text{B00}$, `M[0,1]` $\to 0 \oplus 1 = \text{B01}$, `M[0,2]` $\to 0 \oplus 2 = \text{B02}$, `M[0,3]` $\to 0 \oplus 3 = \text{B03}$.
- **Row 1** ($\text{row} = 01_2$): `M[1,0]` $\to 1 \oplus 0 = \text{B01}$, `M[1,1]` $\to 1 \oplus 1 = \text{B00}$, `M[1,2]` $\to 1 \oplus 2 = \text{B03}$, `M[1,3]` $\to 1 \oplus 3 = \text{B02}$.
- **Row 2** ($\text{row} = 10_2$): `M[2,0]` $\to 2 \oplus 0 = \text{B02}$, `M[2,1]` $\to 2 \oplus 1 = \text{B03}$, `M[2,2]` $\to 2 \oplus 2 = \text{B00}$, `M[2,3]` $\to 2 \oplus 3 = \text{B01}$.
- **Row 3** ($\text{row} = 11_2$): `M[3,0]` $\to 3 \oplus 0 = \text{B03}$, `M[3,1]` $\to 3 \oplus 1 = \text{B02}$, `M[3,2]` $\to 3 \oplus 2 = \text{B01}$, `M[3,3]` $\to 3 \oplus 3 = \text{B00}$.

Now check both access directions across the physical bank table:

- **Any row** (e.g., Row 1): physical banks holding `M[1,0..3]` are `{B01, B00, B03, B02}` —a permutation of all 4 banks, zero conflicts.
- **Any column** (e.g., Column 0: `M[0,0]`, `M[1,0]`, `M[2,0]`, `M[3,0]`): stored in `B00`, `B01`, `B02`, `B03` —again, all distinct banks, zero conflicts.

Extending this bitwise XOR across 5 bits (rows 0–31 and columns 0–31) guarantees that both 32-element row reads and 32-element column reads are completely conflict-free.

4. HBM Memory Coalescing

Moving down from shared memory to HBM introduces a different granularity constraint. HBM is not fetched one 4-byte word at a time; the memory controller combines a warp's global memory accesses into **128-byte cache-line transactions**.

- **Coalesced access (best case):** All 32 threads in a warp (`T0` through `T31`) read 32 consecutive 4-byte words (`M00` through `M31`). Because $32 \times 4 = 128$ bytes, all 32 requests fall inside a single 128-byte cache line (`Cache Line 0`). The memory controller issues **1 transaction**, 100% of the fetched bytes are used, and `Cache Line 1` (`M32` through `M63`) is never fetched.
- **Strided / uncoalesced access (worst case):** Suppose consecutive threads in a warp read with a stride of 32 floats (128 bytes) apart: `T0` reads `M00` (from `Line 0`), `T1` reads `M32` (from `Line 1`), `T2` reads `M64` (from `Line 2`), and so on up to 32 separate cache lines. Even though the warp only needed 128 bytes of data, the memory controller must fetch up to $32 \times 128 = 4{,}096$ bytes from HBM—wasting $31/32 = 96.875\%$ of the transferred HBM bandwidth.

DEFINITION

Memory Coalescing: The hardware mechanism by which the GPU memory controller merges simultaneous HBM accesses from threads in the same warp into as few 128-byte cache-line transactions as possible.

To maintain full coalescing, inner loops must stride along the **contiguous dimension in memory** (stride 1), so adjacent threads access adjacent memory addresses. We will see how row-major strides satisfy this directly when we write Triton kernels in Sections 5–7.

5. Block Occupancy and Wave Quantization

The final hardware reality occurs at the grid level, when the GPU schedules thread blocks onto its 148 SMs on B200. Blocks are dispatched in **waves**: if each SM can run one block of a kernel at a time, the GPU launches up to 148 blocks in parallel as Wave 1, waits for SMs to become free, and then dispatches the remaining blocks in Wave 2.

Suppose a kernel launches **160 thread blocks** on a B200 (148 SMs), with 1 resident block per SM:

- **Wave 1:** Runs 148 blocks across all 148 SMs (`SM 0` through `SM 147`), achieving **100% SM utilization**.
- **Wave 2 (the tail wave):** Only $160 - 148 = 12$ blocks remain. Twelve SMs execute these final blocks while the other **136 SMs sit completely idle** waiting for the tail wave to finish.

If every block takes time T, Wave 1 takes T and Wave 2 also takes T, so total kernel runtime is $2T$. Adding just 12 blocks beyond 148 (an 8.1% increase in work) nearly **doubles** total execution time!

DEFINITION

Wave Quantization: The step-function jump in kernel execution time that occurs when the total number of thread blocks slightly exceeds a multiple of the GPU's concurrent block capacity across all SMs, creating a sparse tail wave that leaves most SMs idle.

WORKED EXAMPLE

Suppose a kernel runs 1 block per SM on a B200 (148 SMs), and each block takes 10 μs. Compare launching 148 blocks versus 160 blocks versus 296 blocks:

- **148 blocks:** $\lceil 148/148 \rceil = 1$ wave $\rightarrow 10$ μs total runtime (100% average SM utilization).
- **160 blocks:** $\lceil 160/148 \rceil = 2$ waves (148 blocks in Wave 1, 12 blocks in Wave 2) $\rightarrow 20$ μs total runtime. Average SM utilization drops to $\frac{160}{2 \times 148} = 54.1\%$.
- **296 blocks:** $\lceil 296/148 \rceil = 2$ full waves $\rightarrow$ still 20 μs total runtime, delivering 100% SM utilization for nearly twice the work of 160 blocks.

There are two primary mitigations for wave quantization: choose tile sizes so the total number of launched blocks is close to a multiple of the number of SMs (148 on B200), or use **persistent kernels** that launch a fixed number of blocks matching the SM count and loop internally over work tiles (which

we will also encounter in Section 3).

Summary: Programming Model vs. Hardware Reality

Every layer of the GPU programming abstraction pairs directly with a hardware performance mechanism:

Level	Programming Abstraction (Correctness)	Hardware Mechanism (Performance)
Grid	Independent thread blocks	Wave quantization across 148 SMs (B200)
HBM	Flat global address space	128-byte coalesced cache-line transactions
Thread Block	Shared scratchpad + barrier sync	32 shared-memory banks (4 B wide) & swizzling
Warp / Thread	SIMT threads with private state	32-thread lockstep, divergence, register occupancy

> **KEY IDEA**
> Correctness depends on the programming abstractions (grid, flat HBM, shared scratchpad, threads), whereas speed depends on aligning with five hardware realities: avoiding intra-warp control divergence, managing register occupancy (or compensating with thread coarsening), swizzling shared memory to eliminate 32-bank conflicts, coalescing HBM reads into 128-byte cache lines, and avoiding sparse tail waves across the 148 SMs.

Having a hardware mental model tells us *what* can go wrong inside a GPU kernel, but it cannot tell us which bottleneck actually dominates a real program. Optimizing based on intuition alone routinely wastes effort on the wrong layer. We need disciplined methods to measure wall-clock latency, separate CPU launch overhead from GPU work, and inspect exactly which kernels execute on the hardware.

✎ Check Your Understanding

1. A kernel uses 128 threads (4 warps) per block and 64 registers per thread on a B200 SM (65,536 registers, max 64 warps). Assuming registers are the only limiting resource, how many blocks fit per SM and what is the resulting warp occupancy?
   Answer: Each block uses $128 \times 64 = 8{,}192$ registers, so $\lfloor 65{,}536 / 8{,}192 \rfloor = 8$ blocks fit per SM. That gives $8 \times 4 = 32$ active warps, for a warp occupancy of $32/64 = 50\%$.
2. What would happen to HBM memory traffic if a 32-thread warp switched from reading 4-byte floats at consecutive indices (`0, 1, 2, ..., 31`) to reading with stride 32 (`0, 32, 64, ..., 992`)?
   Answer: Instead of coalescing into a single 128-byte cache-line transaction (100% utilized), each thread hits a different 128-byte cache line, triggering up to 32 separate transactions and wasting nearly all of the fetched HBM bandwidth.

Section 3: The Performance Engineering Loop — Benchmarking, Launch Overhead, and Profiling

When a model or operator runs slowly, guessing the bottleneck almost always leads to optimizing the wrong thing. Performance engineering follows a closed four-step loop:

1. **Benchmark:** Measure end-to-end wall-clock latency—*how long does it take?* Benchmarking lets you compare implementations objectively and study how latency scales with problem size.
2. **Profile:** Break total time down by individual CUDA kernel—*where is time spent?* Profiling exposes which exact kernels ran and which one dominates runtime.
3. **Change:** Apply a targeted optimization—fuse operations, tile data, or tune launch parameters.
4. **Verify:** Check numerical accuracy and batch invariance (covered in Section 4) before trusting any speedup.

After Step 4, the loop feeds back to Step 1: **re-benchmark and re-profile after every change**.

Benchmarking GPU Kernels Correctly

Timing a GPU operation looks deceptively simple until you encounter how the CPU and GPU actually interact. CUDA kernel launches are **asynchronous**: when Python executes `y = a @ b`, the CPU merely enqueues a launch command onto the GPU command stream and returns control to Python immediately—often microseconds later, before the GPU has finished (or even started) the math.

COMMON MISCONCEPTION

You might think you can benchmark a PyTorch GPU function by recording `time.perf_counter()` before and after calling `run()`. That breaks because CUDA launches are asynchronous: without synchronizing, the CPU timer returns as soon as the launch command is enqueued, measuring only CPU dispatch latency rather than actual GPU execution time.

Two additional artifacts distort naive measurements:

- **Cold-start overhead:** The first invocation of a kernel triggers one-time JIT compilation, library initialization, and memory allocator setup. Without **warmup** iterations, you measure setup cost rather than steady-state execution.
- **CPU-side variance:** Measuring with CPU wall-clock timers includes Python interpreter overhead and host-device synchronization latency. Using **CUDA events** (`torch.cuda.Event(enable_timing=True)`) stamps timestamps directly on the GPU timeline, isolating pure GPU execution time.

The standard benchmarking harness combines warmup, CUDA events, explicit synchronization, and multiple trials:

```
def benchmark(run, num_warmups=1, num_trials=3):
    for _ in range(num_warmups):
        run()                                   # compile / cache warmup
    torch.cuda.synchronize()                    # wait for warmup to finish

    times = []
    for _ in range(num_trials):
        start = torch.cuda.Event(enable_timing=True)
        end   = torch.cuda.Event(enable_timing=True)
        start.record()
        run()
        end.record()
        torch.cuda.synchronize()                # wait for GPU work!
        times.append(start.elapsed_time(end))   # milliseconds on GPU
    return sum(times) / len(times)
```

Notice the two `torch.cuda.synchronize()` calls: the first ensures warmup work completes before timing begins; the second ensures `end` has actually been reached on the GPU before `start.elapsed_time(end)` queries the elapsed time in milliseconds. Running multiple trials (or using `torch.utils.benchmark`) captures steady-state variance across runs.

WORKED EXAMPLE

Suppose a matrix multiply takes 0.015 ms for the CPU to enqueue, 4.20 ms to compile/allocate on its first call, and 0.85 ms on the GPU in steady state.

- **Naive timing (no warmup, no `synchronize`):** Returns ≈ 0.015 ms—understating GPU time by more than $50\times$ because it only timed CPU dispatch.

> - **Synchronized timing without warmup (1 trial):** Returns ≈ 4.20 ms—overstating steady-state runtime by $\sim 5\times$ because it included one-time compilation and allocation.
> - **`benchmark(run, num_warmups=1, num_trials=3)`:** The warmup absorbs the 4.20 ms setup, `torch.cuda.synchronize()` drains the warmup, and the three GPU-event trials record e.g. `[0.85, 0.86, 0.84]` ms, returning the true steady-state GPU latency of 0.85 ms.

How Latency Scales with Problem Size

Once you can benchmark accurately, sweeping matrix dimension N across $N \in \{256, 512, 1024, 2048, 4096, 8192\}$ for an $N \times N$ square matrix multiplication exposes two completely different performance regimes.

Imagine plotting wall-clock latency on the vertical axis against matrix dimension N (at powers of 2 from 256 to 8192) on the horizontal axis:

- **Left regime — Overhead-bound ($N = 256, 512, 1024$):** The latency curve is **roughly flat**. Even though going from $N = 256$ to $N = 1024$ increases the arithmetic work by $4^3 = 64\times$, wall-clock time barely moves. Here the GPU's 148 SMs are under-filled and fixed kernel launch overhead dominates total runtime.
- **Right regime — Compute-bound ($N = 2048, 4096, 8192$):** Once N is large enough to saturate the GPU's SMs and tensor cores, the curve bends steeply upward and follows the theoretical **cubic scaling** $O(N^3)$ of matrix multiplication (each doubling of N multiplies FLOPs—and runtime—by roughly $8\times$).

This two-regime curve is not an academic curiosity: **small-batch autoregressive decode** in language models lives squarely in the left, overhead-bound regime, where matrix-vector multiplies (`GEMV`) and small-batch operations finish so quickly that launch overhead dominates.

Small Batch: Launch Overhead, CUDA Graphs, and Persistent Megakernels

To see why small-batch decode suffers in standard eager execution, examine the GPU execution timeline of a single transformer decode layer across three progressively tighter execution strategies:

1. **Eager PyTorch (many small kernels):** The GPU runs `RMSNorm`, then hits an idle **gap**, then runs `QKV GEMV`, then an idle **gap**, then `RoPE`, then a **gap**, then `Attention`, then a **gap**, then `MLP / MoE`. Because each small kernel completes rapidly, the CPU dispatch gaps between kernels consume a large fraction of total latency.
2. **CUDA Graph:** Instead of having the CPU dispatch each kernel one by one on every decode step, a **CUDA Graph** captures the entire static sequence of kernels (`RMSNorm`, `QKV GEMV`, `RoPE`, `Attention`, `MLP / MoE`) once and replays the whole graph in a **single CPU call**. This removes CPU dispatch overhead and shrinks the gaps dramatically, compressing the timeline—though distinct inter-kernel boundaries (where one kernel drains and writes results to HBM before the next kernel starts) still remain.
3. **Persistent Megakernel:** Goes one step further by fusing the entire decode step (`RMSNorm` $\rightarrow$ `QKV` $\rightarrow$ `RoPE` $\rightarrow$ `Attn` $\rightarrow$ `MLP`) into **one single persistent kernel launch**. Because execution never leaves

the kernel between stages, there are no launch gaps, no inter-kernel drains, and intermediate activations can be handed off on-chip.

> **DEFINITION**
> **CUDA Graph:** A mechanism that records a static sequence of GPU kernel launches and replays the entire sequence from a single CPU call, removing per-kernel CPU dispatch overhead while preserving inter-kernel boundaries.

> **DEFINITION**
> **Persistent Megakernel:** A single long-lived GPU kernel launch that executes an entire multi-stage computation (such as a full autoregressive decode step) without returning to the host, eliminating both launch gaps and inter-kernel drains and enabling on-chip handoff.

Profiling: Seeing What Actually Executes

Benchmarking tells you *total* time, but not *which kernels* consumed that time. Profiling opens the black box. In PyTorch, `torch.profiler` records every CUDA kernel launched during a region of code:

```
def profile(run, num_warmups=1):
    for _ in range(num_warmups):
        run()
    torch.cuda.synchronize()

    with torch.profiler.profile(activities=[ProfilerActivity.CUDA]) as prof:
        run()
        torch.cuda.synchronize()

    return prof.key_averages().table(sort_by="cuda_time_total", row_limit=10)
```

Passing `activities=[ProfilerActivity.CUDA]` and sorting by `"cuda_time_total"` lists the exact CUDA kernels invoked under the hood and how long each spent executing on the GPU. Running this profiler reveals an important property of PyTorch: **kernel dispatch depends on tensor shape**. Operations such as `add(2048)`, `matmul(2048)`, and `matmul(128)` do not invoke the same underlying GPU binary; PyTorch selects different kernels, tile shapes, and algorithms depending on the dimensions.

When you need hardware-level performance counters beyond kernel durations—such as **SM utilization**, **achieved HBM bandwidth**, and **warp stall reasons**—you step down to **NVIDIA Nsight**.

Even in the PyTorch Profiler table, the kernel names themselves carry dense architectural information. Consider an actual profiled matrix-multiplication kernel name on B200:

```
cutlass3x_sm100_simt_sgemm_f32_f32_f32_f32_f32_64x64x16_1x1x1_3_nnn_align1_bi…
```

Breaking apart its key fields tells you exactly how the library configured the GPU:

- **`cutlass3x`:** NVIDIA's CUDA linear algebra template library (CUTLASS 3.x) generated this kernel.

- `sm100`: Targeted specifically for the **NVIDIA Blackwell architecture (B200)**.
- `f32`: Uses 32-bit floating-point (`float32`) operands and accumulator.
- `64x64x16`: The thread-block tile shape ($M \times N \times K = 64 \times 64 \times 16$) selected for this specific matrix size.

Seeing `64x64x16` for one matrix dimension and a different tile shape for another confirms directly in the profile how runtime dispatch adapts tiling to tensor shapes.

> **KEY IDEA**
> Benchmarking requires warmup, CUDA events, and explicit `torch.cuda.synchronize()` to measure true GPU time rather than asynchronous CPU dispatch; profiling with PyTorch Profiler and NVIDIA Nsight exposes the exact kernels, tile shapes, and hardware bottlenecks behind that latency. In the small-batch overhead-bound regime, CUDA Graphs eliminate CPU dispatch gaps and persistent megakernels eliminate inter-kernel boundaries altogether.

With benchmarking and profiling in hand, we can now test a concrete case study—the GeLU activation function—to see what profiling reveals about naive PyTorch code, why **kernel fusion** delivers massive speedups, and what subtle correctness hazard we must guard against before shipping any fast kernel.

> **✎ Check Your Understanding**
>
> 1. If you time a GPU matrix multiplication using a Python wall-clock timer without calling `torch.cuda.synchronize()`, will the measured time be too high or too low? Explain why.
>   **Answer:** It will be far too low. Because CUDA kernel launches are asynchronous, the CPU returns as soon as it enqueues the launch command, so the timer measures only CPU dispatch overhead rather than the GPU's execution time.
> 2. What would happen to the gaps between `RMSNorm`, `QKV GEMV`, `RoPE`, `Attention`, and `MLP / MoE` during small-batch decode if you switch from eager PyTorch to a CUDA Graph, and what overhead still remains?
>   **Answer:** Switching to a CUDA Graph replays the entire static sequence in a single CPU call, removing per-kernel CPU dispatch overhead and shrinking the gaps; however, inter-kernel boundaries and drains between separate kernels still remain (unlike in a persistent megakernel).

Section 4: Kernel Fusion and Correctness Verification (Accuracy & Batch Invariance)

To see the performance engineering loop in action, consider the **GeLU** (Gaussian Error Linear Unit) activation function used throughout neural networks. Its standard tanh approximation is:

$$\mathrm{GeLU}(x) = 0.5\,x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715\,x^3\right)\right)\right).$$

Let us decompose this nested expression from the inside out:

- **Innermost cubic polynomial:** $x + 0.044715\,x^3$ applies a small cubic perturbation to the input scalar x.
- **Scaling constant:** Multiplying by $\sqrt{2/\pi} \approx 0.79788456$ scales the argument into the shape of the Gaussian cumulative distribution approximation.
- **Smooth gate:** $\tanh(\cdot)$ maps that scaled value smoothly into $(-1, 1)$; adding 1 and multiplying by 0.5 turns it into a smooth gating factor in $(0, 1)$.
- **Gated output:** Multiplying by x gates negative values smoothly toward 0 while passing large positive values nearly unchanged.

As a single conceptual unit, GeLU is a purely **elementwise** nonlinear function: each output entry y_i depends only on x_i.

Now compare three PyTorch implementations of this exact formula:

```python
# 1. Naive PyTorch (many separate elementwise ops)
def naive_gelu(x):
    return 0.5 * x * (1 + torch.tanh(0.79788456 * (x + 0.044715 * x * x * x)))

# 2. Built-in PyTorch (fused CUDA kernel)
def builtin_gelu(x):
    return torch.nn.functional.gelu(x, approximate="tanh")

# 3. PyTorch compiler (automatically generates a fused Triton kernel)
compiled_gelu = torch.compile(naive_gelu)
```

All three produce the same numerical output (verified via `torch.allclose`). Yet when benchmarked on a $16{,}384 \times 16{,}384$ tensor, `builtin_gelu` and `compiled_gelu` are dramatically faster than `naive_gelu`. Why?

Why Kernel Fusion Wins on GeLU

In eager PyTorch, every Python operator in `naive_gelu` launches its own separate CUDA kernel, and every kernel boundary forces a full round-trip to HBM:

1. Compute `x * x * x`: read `x` from HBM, write intermediate tensor back to HBM.
2. Multiply by `0.044715` and add `x`: read intermediates from HBM, write result to HBM.
3. Scale by `0.79788456`: read from HBM, write to HBM.
4. Apply `tanh`: read from HBM, write to HBM.
5. Add `1`: read from HBM, write to HBM.
6. Multiply by `0.5 * x`: read from HBM, write final `y` to HBM.

Picture this dataflow visually: above a horizontal bar representing HBM (at 8 TB/s), six red operator boxes (`x * x * x`, `mul / add`, `scale`, `tanh`, `+ 1`, `0.5 * x * ...`) each send an upward arrow pulling data out of HBM and a downward arrow pushing an intermediate tensor right back into HBM. Every element traverses the slow 8 TB/s HBM interface over and over again, while doing only a handful of floating-point operations per element. Elementwise math is **memory-bandwidth bound**, not compute bound.

Kernel fusion collapses all of those elementwise operations into a **single kernel**. Visually, all six red boxes collapse into one green fused Triton/CUDA kernel box with just **one upward arrow** ("read `x` once") and **one downward arrow** ("write `y` once"). Every intermediate value—x^3, the scaled argument, $\tanh(\cdot)$, and $1 + \tanh(\cdot)$—stays inside per-thread **registers** (~447 TB/s) and never touches HBM. Remarkably, `torch.compile(naive_gelu)` performs this fusion automatically by lowering `naive_gelu` into a single fused **Triton kernel**.

DEFINITION

Kernel Fusion: Combining multiple consecutive operations into a single GPU kernel so that intermediate values remain in on-chip registers or shared memory instead of being written to and re-read from HBM between separate kernel launches.

WORKED EXAMPLE

Estimate the minimum HBM transfer time on a B200 (8 TB/s HBM) for a $16{,}384 \times 16{,}384$ `float32` tensor under fused GeLU versus an unfused 6-kernel chain where each kernel reads 1 tensor and writes 1 tensor (a lower bound on `naive_gelu`, which actually reads `x` multiple times).

1. **Tensor size in bytes:**

$$16{,}384 \times 16{,}384 = 268{,}435{,}456 \text{ elements}.$$

   At 4 bytes per `float32`, one tensor occupies:

$$268{,}435{,}456 \times 4 = 1{,}073{,}741{,}824 \text{ bytes} = 1 \text{ GiB} \approx 1.074 \text{ GB}.$$

2. **Fused GeLU (1 read + 1 write = 2 tensors):**
   - Total HBM traffic: $2 \times 1.074 \text{ GB} \approx 2.15 \text{ GB}$.
   - Minimum HBM time at 8 TB/s ($8{,}000 \text{ GB/s}$):

$$\frac{2.15 \text{ GB}}{8{,}000 \text{ GB/s}} \approx 0.27 \text{ ms}.$$

3. **Unfused 6-kernel GeLU (at least 6 reads + 6 writes = 12 tensors; even more because `x` is reused):**
   - Total HBM traffic: at least $12 \times 1.074 \text{ GB} \approx 12.88 \text{ GB}$ (and higher when binary ops read two tensors).

- Minimum HBM time at 8 TB/s:

$$\frac{12.88\ \text{GB}}{8{,}000\ \text{GB/s}} \approx 1.61\ \text{ms}.$$

Just by eliminating intermediate HBM round-trips, fusion cuts memory traffic—and therefore runtime on this memory-bandwidth-bound operation—by at least **6×** (even before accounting for extra kernel launch overhead and repeated reads of `x`).

Correctness Testing: Numerical Accuracy & Batch Invariance

In Step 4 of the performance engineering loop, we must verify that a fast kernel is actually correct. The standard first check compares the kernel's output against a trusted PyTorch reference implementation using `torch.allclose`. However, passing `torch.allclose` on a fixed test shape is not enough to guarantee reliability in production.

A subtle production bug arises when **a kernel's output for an identical input changes depending on the batch size** it is processed with. Why would changing how many other rows sit alongside your input in a batch alter the numbers computed for your row?

The root cause is that **floating-point addition is not associative**:

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

In exact real arithmetic, grouping additions differently never changes a sum. In finite-precision floating point, rounding occurs after every operation, so changing the order in which partial sums are combined changes the low-order bits of the result.

Remember from Section 3 that PyTorch and GPU kernel libraries dispatch different algorithms depending on tensor shape. Specifically, **Split-K** reductions (which split a matrix multiplication's inner reduction dimension across multiple blocks and sum their partial results) and **tile-shape** selection change with batch size. Changing the tile shape or Split-K partitioning changes the summation order, which produces slightly different floating-point outputs for the exact same input row.

DEFINITION

Batch Invariance: The correctness property that a kernel produces bit-identical (or invariant) outputs for a given input example regardless of the batch size or what other examples are co-batched alongside it.

COMMON MISCONCEPTION

You might think a tiny floating-point difference in the least significant bits between batch size 1 and batch size 32 is harmless because `torch.allclose` passes. That breaks in autoregressive LLM serving: small numerical differences in logits can flip which token has the highest probability or alter sampling boundaries, causing the **exact same prompt to produce different tokens** depending on what other users' requests happened to be co-batched with it.

WORKED EXAMPLE

To see how changing summation order alters a finite-precision floating-point result, consider a 3-digit decimal floating-point system (rounding after every addition) summing $a = 100$, $b = 0.4$, and $c = 0.4$:

- **Order 1 — $(a + b) + c$:**
  - First compute $100 + 0.4 = 100.4$, which rounds to 100 in 3 significant digits.
  - Then compute $100 + 0.4 = 100.4$, which rounds to 100.
- **Order 2 — $a + (b + c)$:**
  - First compute $0.4 + 0.4 = 0.8$ (exact in 3 significant digits).
  - Then compute $100 + 0.8 = 100.8$, which rounds to 101.

The inputs are identical, yet regrouping the sum changes the output from 100 to 101. In GPU kernels, switching tile shapes or Split-K partitions across batch sizes produces this exact regrouping effect across thousands of multiply-add terms.

Because of this failure mode, **batch invariance** must be tested directly alongside benchmarking, profiling, and `torch.allclose` accuracy checks. In production serving engines such as **vLLM**, engineers maintain dedicated batch-invariance tests for **matmul** and **fused all-reduce + RMSNorm** right inside continuous integration (CI). Fast kernels are only shippable if they pass both **accuracy** and **batch-invariance** checks.

KEY IDEA

Elementwise operations like GeLU are memory-bandwidth bound, so fusing them into one kernel eliminates intermediate HBM round-trips and keeps all intermediates in ~447 TB/s registers. Before trusting any optimized kernel, you must verify both numerical closeness (`torch.allclose`) and **batch invariance**, because shape-dependent tile dispatch and Split-K reductions change floating-point summation order and can make LLM outputs depend on co-batched requests.

For simple elementwise chains like GeLU, `torch.compile` can fuse operations automatically into Triton kernels. When you need custom fusion patterns, non-elementwise reductions, or explicit control over tiling and shared-memory reuse, you must write GPU kernels yourself. That brings us to the next decision: which kernel language or DSL to use, and how to write and inspect our first Triton kernel.

✎ Check Your Understanding

1. An unfused elementwise sequence consists of 5 consecutive kernels, each reading one N-element tensor from HBM and writing one N-element tensor back to HBM. If all 5 operations are fused into a single kernel, by what factor does total HBM traffic drop?
   Answer: It drops by a factor of **5×**, from $10N$ total element transfers (5 reads + 5 writes) down to $2N$ transfers (1 read + 1 write).
2. True or false: If a custom matrix-multiplication kernel passes `torch.allclose` against PyTorch at batch size 16, it is guaranteed to produce the same output for a given prompt at batch size 1. Explain your answer.
   Answer: False. Because floating-point addition is not associative ($(a+b)+c \neq a+(b+c)$), shape-dependent tile dispatch or Split-K reductions can change summation order across batch sizes, violating batch invariance even when `torch.allclose` passes.

Section 5: The Kernel DSL Landscape and Your First Triton Kernel

When automatic compilation (`torch.compile`) does not produce the kernel you need, you write a custom GPU kernel directly. Choosing the right language involves a trade-off between **programmer productivity** (letting a compiler manage low-level hardware details) and **peak performance** (explicitly controlling layouts, pipelines, and registers on the hardest kernels).

CUDA vs. Triton: Thread-Level vs. Block-Level Programming

The two foundational mental models sit at opposite ends of the abstraction hierarchy introduced in Section 1:

- **CUDA (NVIDIA) — Program at the thread level:** You write code specifying what each **individual thread** executes.
  - *Pros:* Fine-grained control over registers, warp intrinsics, shared-memory layouts, and asynchronous pipelines.
  - *Cons:* You must manually manage HBM memory coalescing, shared-memory bank conflicts (and swizzling), shared-memory allocation, and intra-block synchronization.
- **Triton (OpenAI) — Program at the thread-block level:** You write Python code specifying operations on whole **blocks (tiles)** of data.
  - *Mental model:* Load a tile from HBM into on-chip memory, compute on the tile (fusing operations together), and store the tile back to HBM.
  - *Pros:* The Triton compiler automatically handles memory coalescing, shared-memory allocation, and intra-block thread scheduling.

> **DEFINITION**
> **Triton:** A Python-based GPU kernel language developed by OpenAI in which the programmer writes operations at the **thread-block (tile)** level while the compiler automates coalescing, shared-memory management, and thread scheduling within each block.

The Modern Kernel DSL Landscape

Kernel authoring is no longer a binary choice between high-level Triton and raw CUDA. Modern GPU kernel development spans a five-tier continuum from highest-level compiler automation on the left to lowest-level explicit hardware control on the right:

1. **Triton (High level):** Block-level Python; serves as the `torch.compile` backend; ideal for fast prototyping and clean expression of tiled algorithms.
2. **NVIDIA cuTile (High level):** NVIDIA's own tile-based programming model for GPUs.
3. **TileLang (Middle level):** Exposes explicit tile buffers and scheduling controls; powers DeepSeek TileKernels.
4. **Gluon (Lower-level Triton):** A Triton dialect with explicit memory layouts, used to target Blackwell hardware closely.
5. **CuTe DSL (Lowest level among Python DSLs):** Python CUTLASS offering fine-grained layout and hardware control; FlashAttention-4 is written in CuTe DSL. Below CuTe DSL sit raw **CUDA / CUTLASS** and **PTX**.

The trade-off across this spectrum is consistent: higher-level DSLs accelerate development dramatically, while lower-level DSLs recover the last increment of performance on the hardest, most hardware-sensitive kernels. To learn the core block-level patterns cleanly, we use **Triton** for the four kernel implementations that follow.

The Four-Stage Triton Kernel Pattern

Every Triton kernel follows the exact same four-step pipeline:

1. **Identify Block:** Query which block instance is executing via `tl.program_id(axis)`.
2. **Load Tile + Mask:** Compute a vector (or 2D block) of memory addresses and load data safely from HBM using `tl.load(ptrs, mask)`.
3. **Compute On-Chip:** Perform elementwise math, reductions, or matrix dot products while all data stays in on-chip registers/shared memory.
4. **Store Tile:** Write the finished tile back to HBM with `tl.store(ptrs, y, mask)`.

We will apply this pattern to four kernels of increasing structural complexity:

1. **GeLU** (this section): 1D elementwise fusion.
2. **Softmax** (Section 6): row-wise reduction where an entire row fits in one block.
3. **Row sum** (Section 6): tiled reduction where a row exceeds one block.
4. **Matmul + ReLU** (Section 7): 2D tiling with shared-memory reuse and a fused epilogue.

Example 1: Elementwise GeLU in Triton

Start by partitioning a 1D contiguous tensor of length `num_elements` into fixed-size chunks of length `BLOCK_SIZE = 1024`. Picture the 1D array laid out horizontally from left to right:

- `Block 0 (pid=0)` owns elements `0 .. 1023`.
- `Block 1 (pid=1)` owns elements `1024 .. 2047`.
- `Block 2 (pid=2)` owns elements `2048 .. 3071`.
- `Block 3 (pid=3)` covers the tail: the first portion of its 1024 lanes maps to **valid** elements (`offsets < num_elements`), while the remaining tail lanes exceed `num_elements` and must be **masked** off to prevent out-of-bounds memory accesses.

The host-side Python launcher allocates the output tensor, computes the ceiling division for the grid size, and launches the kernel:

```
def triton_gelu(x: torch.Tensor):
    assert x.is_cuda and x.is_contiguous()
    y = torch.empty_like(x)
    num_elements = x.numel()
    BLOCK_SIZE = 1024
    num_blocks = triton.cdiv(num_elements, BLOCK_SIZE)  # ceil(N / BLOCK_SIZE)
    triton_gelu_kernel[(num_blocks,)](x, y, num_elements, BLOCK_SIZE=BLOCK_SIZE)
    return y
```

Look at two key host-side details:

- `triton.cdiv(num_elements, BLOCK_SIZE)` computes ceiling division $\lceil \text{num_elements}/\text{BLOCK_SIZE} \rceil$ so even a partial tail chunk gets a block.
- The launch syntax `kernel[grid](args...)` takes a grid tuple—here the 1D tuple `(num_blocks,)`—and spawns `num_blocks` independent program instances on the GPU. Notice that passing PyTorch tensors `x` and `y` automatically passes their base device pointers (`x_ptr`, `y_ptr`) to the kernel.

Now examine the Triton kernel itself:

```python
@triton.jit
def triton_gelu_kernel(x_ptr, y_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(axis=0)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < num_elements

    x = tl.load(x_ptr + offsets, mask=mask)

    # tanh(a) = (exp(2a) - 1) / (exp(2a) + 1)
    a = 0.79788456 * (x + 0.044715 * x * x * x)
    exp = tl.exp(2 * a)
    tanh = (exp - 1) / (exp + 1)
    y = 0.5 * x * (1 + tanh)

    tl.store(y_ptr + offsets, y, mask=mask)
```

Walk through every line of this kernel:

- `@triton.jit` marks the function for just-in-time compilation by Triton.
- `BLOCK_SIZE: tl.constexpr` declares `BLOCK_SIZE` as a **compile-time constant** (which Triton requires to be a power of 2 for `tl.arange`). Knowing the tile size at compile time allows the compiler to allocate fixed register layouts and unroll vectorized memory instructions.
- `pid = tl.program_id(axis=0)` retrieves this block's index along grid axis 0.
- `offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)` creates a 1D block vector of `BLOCK_SIZE` consecutive element indices. Because adjacent lanes access consecutive offsets, the compiler automatically generates coalesced 128-byte HBM transactions (Section 2).
- `mask = offsets < num_elements` creates a boolean vector of length `BLOCK_SIZE` that is `True` for in-bounds elements and `False` for tail padding.
- `tl.load(x_ptr + offsets, mask=mask)` loads the entire tile from HBM into on-chip registers in one fused read, safely skipping out-of-bounds lanes.
- The GeLU math computes $a = 0.79788456\,(x + 0.044715\,x^3)$, evaluates $\tanh(a) = \frac{e^{2a}-1}{e^{2a}+1}$ via `tl.exp(2 * a)`, and computes $y = 0.5\,x\,(1 + \tanh(a))$. Every intermediate (`a`, `exp`, `tanh`) lives exclusively in registers—zero HBM traffic.
- `tl.store(y_ptr + offsets, y, mask=mask)` writes the finished tile back to HBM in a single masked store.

WORKED EXAMPLE

Trace `triton_gelu` on a 1D tensor with `num_elements = 3500` and `BLOCK_SIZE = 1024`:

1. **Grid size:**

$$\text{num_blocks} = \text{cdiv}(3500, 1024) = \left\lceil \frac{3500}{1024} \right\rceil = 4 \text{ blocks } (\text{pid} \in \{0, 1, 2, 3\}).$$

2. **Block 0 (`pid = 0`):** `offsets = 0 + [0..1023] = [0..1023]`. All 1024 entries satisfy `< 3500`, so `mask` is all `True`.

3. **Block 1 (`pid = 1`):** `offsets = [1024..2047]`. All `True`.
4. **Block 2 (`pid = 2`):** `offsets = [2048..3071]`. All `True`.
5. **Block 3 (`pid = 3`):** `offsets = [3072..4095]`.
   - For indices `3072..3499` (428 elements): `mask` is `True`; these elements are loaded, transformed through GeLU in registers, and stored.
   - For indices `3500..4095` (596 tail lanes): `mask` is `False`; neither `tl.load` nor `tl.store` touches memory at those out-of-bounds addresses.

Under the Hood: Triton Compiles to PTX and SASS

How does block-level Python turn into machine code running on 148 Blackwell SMs? Compilation proceeds through two lowering stages:

Triton Python (`@triton.jit`) $\longrightarrow$ PTX (Virtual GPU ISA) $\longrightarrow$ SASS (Blackwell Binar

DEFINITION

PTX (Parallel Thread Execution): NVIDIA's virtual GPU instruction set architecture that serves as an intermediate assembly representation between high-level GPU languages (like CUDA or Triton) and native hardware machine code (SASS).

DEFINITION

SASS: The native hardware-specific machine-code instruction set executed directly by a specific GPU architecture (such as `sm100` instructions on NVIDIA Blackwell B200).

Inspecting `kernel.asm["ptx"]` for `triton_gelu_kernel` reveals exactly how Triton bridges the block-level programming model back down to the thread-level hardware model from Sections 1 and 2:

- **`ld.global.*` and `st.global.*`:** The single global HBM loads and stores corresponding to `tl.load` and `tl.store`—confirming that all intermediates stay in registers.
- **`%ctaid.x` and `%tid.x`:** Hardware special registers holding the thread block index (CTA ID, i.e., `pid`) and the individual thread index inside the block.
- **`%f*` and `%r*`:** Floating-point registers and integer registers allocated per thread.
- **Automatic thread coarsening:** Rather than assigning 1 element to each of 1024 threads, Triton automatically assigns **8 elements per thread** (so 128 threads cover a `BLOCK_SIZE = 1024` tile) to amortize address calculations and instruction overhead—directly demonstrating the thread coarsening principle from Section 2.

COMMON MISCONCEPTION

You might think that setting `BLOCK_SIZE = 1024` in Triton launches 1024 physical GPU threads per thread block, one per element. That breaks because `BLOCK_SIZE` is the **data tile size**, not the hardware thread count; Triton chooses how many threads to launch and applies thread coarsening (such as 8 elements per thread in `triton_gelu`) automatically during PTX lowering.

KEY IDEA

Modern kernel authoring spans a continuum from high-level tile DSLs (Triton, cuTile) through TileLang and Gluon down to CuTe DSL and raw CUDA/PTX. Triton kernels follow a four-step block-level pattern (identify block, masked load, on-chip compute, masked store) that lowers through PTX to native `sm100` SASS with automatic coalescing and thread coarsening.

Elementwise GeLU is the simplest possible case: every lane computes its output without ever looking at any other lane. Real language models rely just as heavily on **reductions**—operations like softmax, layer normalization, and sums—where every output in a row depends on summary statistics computed across all elements of that row. Next, we extend the Triton pattern to handle both single-block reductions and multi-tile reductions.

✎ Check Your Understanding

1. Fill in the blank: Triton lowers `@triton.jit` Python kernels first into _____ (a virtual GPU ISA inspectable via `kernel.asm["ptx"]`) and then into _____ (native `sm100` Blackwell binary instructions).
   Answer: Triton lowers Python kernels first into **PTX** and then into **SASS**.
2. What would happen in `triton_gelu_kernel` if `num_elements` is not a multiple of `BLOCK_SIZE` and you omitted `mask=mask` from `tl.load` and `tl.store`?
   Answer: In the final thread block, lanes with `offsets >= num_elements` would read from and write to out-of-bounds HBM addresses past the end of the tensors, causing memory corruption or illegal memory access errors.

Section 6: Reduction Kernels in Triton — Fused Softmax and Tiled Row Sum

Unlike GeLU, where y_i depends only on x_i, a **row-wise reduction** combines values across an entire row of an $M \times N$ matrix. As we saw in Section 1, non-elementwise operations require threads to exchange intermediate values—making them prime candidates for placing an entire row inside one thread block so the exchange happens on-chip rather than through HBM.

Example 2: Fused Softmax and Counting HBM Traffic

For each row $x = (x_0, x_1, \ldots, x_{N-1})$ of an $M \times N$ matrix, the mathematically exact row-wise softmax is $e^{x_j} / \sum_k e^{x_k}$. In floating point, computing e^{x_j} directly overflows when entries are large (for example, on the row `[0, 0, 100]`, e^{100} is enormous and larger values overflow `float32` to `inf`, producing `inf / inf = NaN`).

To ensure **numerical stability**, softmax always subtracts the row maximum $m = \max_k x_k$ first:

$$\text{softmax}(x)_j = \frac{\exp(x_j - m)}{\sum_{k=0}^{N-1} \exp(x_k - m)}, \qquad m = \max_{0 \le k < N} x_k.$$

Because $\frac{e^{x_j - m}}{\sum_k e^{x_k - m}} = \frac{e^{-m} e^{x_j}}{e^{-m} \sum_k e^{x_k}} = \frac{e^{x_j}}{\sum_k e^{x_k}}$, the mathematical value is unchanged, while every exponent argument $x_j - m \le 0$, guaranteeing $\exp(x_j - m) \in (0, 1]$ with zero overflow.

Now count the exact HBM traffic of naive eager PyTorch implementing these five steps on an $M \times N$ matrix:

Naive PyTorch Step ($M \times N$ matrix)	HBM Reads	HBM Writes
`x_max = x.max(dim=1)[0]`	MN	M
`x = x - x_max[:, None]`	$MN + M$	MN
`num = torch.exp(x)`	MN	MN
`den = num.sum(dim=1)`	MN	M
`y = num / den[:, None]`	MN	MN
Naive total	$5MN + M$	$3MN + 2M$
Fused Triton kernel (ideal)	MN	MN

Look at those totals: naive PyTorch reads $5MN + M$ elements and writes $3MN + 2M$ elements—roughly $8MN$ **total HBM element transfers**. A fused Triton kernel that loads each row into on-chip memory once, performs the max, subtraction, exponential, sum, and division on-chip, and writes the final probabilities once performs only MN reads and MN writes—$2MN$ **total transfers**. Because softmax is memory-bandwidth bound, cutting HBM traffic from $\approx 8MN$ to $2MN$ yields up to a **4× memory-bandwidth speedup**.

Implementing Fused Softmax When One Row Fits in One Block

Assume a single row of length `num_cols` (N) fits inside one thread block. We launch a 1D grid of size `(M,)` —exactly **one block per row**—and choose `BLOCK_SIZE = triton.next_power_of_2(num_cols)` so the tile covers the entire row (with any lanes from `num_cols` to `BLOCK_SIZE - 1` masked off):

```python
@triton.jit
def triton_softmax_kernel(
    x_ptr, y_ptr, x_row_stride, y_row_stride,
    num_cols, BLOCK_SIZE: tl.constexpr,
):
    row_idx = tl.program_id(0)
    col_offsets = tl.arange(0, BLOCK_SIZE)

    x_ptrs = x_ptr + row_idx * x_row_stride + col_offsets
    x_row = tl.load(x_ptrs, mask=col_offsets < num_cols,
                    other=float("-inf"))

    x_row = x_row - tl.max(x_row, axis=0)
    numerator = tl.exp(x_row)
    y_row = numerator / tl.sum(numerator, axis=0)

    y_ptrs = y_ptr + row_idx * y_row_stride + col_offsets
    tl.store(y_ptrs, y_row, mask=col_offsets < num_cols)
```

Examine the structure visually: the M-row matrix is sliced horizontally so `Row 0` is owned by `pid=0`, `Row 1` by `pid=1`, `Row 2` by `pid=2`, and so on. Each row has valid columns `0 .. N-1` on the left and masked padding lanes on the right filled with `-inf`.

Three details in this kernel are essential:

1. **Row stride addressing:** `x_ptr + row_idx * x_row_stride + col_offsets` jumps to the start of `row_idx` and steps across contiguous columns with stride 1 (maintaining coalesced HBM loads).
2. **Padding value `other=float("-inf")`:** Masked lanes do not read from HBM; instead, `tl.load` fills them with $-\infty$. Because $-\infty$ is smaller than every finite number, it never alters `tl.max(x_row, axis=0)`. Then, after subtracting the max, $-\infty - m = -\infty$, and $\exp(-\infty) = 0$, so padded lanes contribute **exactly zero** to `tl.sum(numerator, axis=0)`.
3. **Automatic intra-block reduction:** `tl.max` and `tl.sum` perform the cross-thread reduction within the block automatically—you never write manual shared-memory tree reductions or synchronization barriers.

COMMON MISCONCEPTION

You might think you can safely pass `other=0.0` for masked padding lanes in `triton_softmax_kernel`, since zero seems like a neutral padding value. That breaks twice: first, if all real entries in a row are negative (e.g., `[-5, -2, -1]`), `tl.max` would mistakenly pick `0.0` from the padding; second, even when the max is positive, $\exp(0.0 - m) > 0$, so every padded lane adds a spurious positive term to the denominator `tl.sum(numerator, axis=0)`, corrupting every output probability!

WORKED EXAMPLE

Trace `triton_softmax_kernel` on the 3-element row $x = [0, 0, 100]$ (`num_cols = 3`), padded to `BLOCK_SIZE = 4`:

1. **Correct padding (`other=float("-inf")`):**
   - Loaded vector: `x_row = [0.0, 0.0, 100.0, -inf]`.
   - Row max: `tl.max(x_row, axis=0) = 100.0`.
   - Shifted vector: `x_row - 100.0 = [-100.0, -100.0, 0.0, -inf]`.
   - Exponentiated: `numerator = [e^{-100}, e^{-100}, 1.0, 0.0]` $\approx [3.72 \times 10^{-44}, 3.72 \times 10^{-44}, 1.0, 0.0]$.
   - Denominator: `tl.sum(numerator, axis=0)` ≈ 1.0.
   - Output stored for valid lanes `0..2`: `y_row` $\approx [3.72 \times 10^{-44}, 3.72 \times 10^{-44}, 1.0]$.
2. **Contrast with the pitfall (`other=0.0`) on an all-negative row $x = [-2, -2, -2]$ (`BLOCK_SIZE = 4`):**
   - Loaded vector: `[-2.0, -2.0, -2.0, 0.0]`.
   - `tl.max` picks `0.0` instead of `-2.0`.
   - `numerator = [e^{-2}, e^{-2}, e^{-2}, e^{0}] = [0.1353, 0.1353, 0.1353, 1.0]`.
   - `den = 1.406`, so each valid entry becomes $0.1353/1.406 = 0.0962$ instead of the true $1/3 = 0.3333$!

Choosing `other` to be the mathematical identity element of the reduction is mandatory for correctness.

Example 3: When a Row Does Not Fit in One Block — Tiled Row Sum

Our softmax kernel relied on a strict assumption: the entire row of length N fit into a single `BLOCK_SIZE` tile on-chip. What happens when a row is far too long to fit in one block's register and shared-memory budget?

Instead of loading the entire row at once, we keep **one thread block per row**, choose a manageable `BLOCK_SIZE` (for example, `BLOCK_SIZE = 1024` for a row of $N = 4096$ columns, giving 4 tiles; or `BLOCK_SIZE = 4` for $N = 12$, giving 3 tiles), and **loop over tiles along the row**:

1. Initialize a per-lane register vector `acc` of length `BLOCK_SIZE` to zeros (`tl.float32`).
2. Loop across the row in steps of `BLOCK_SIZE` (Tile 0, Tile 1, Tile 2, …). In each tile, the same `BLOCK_SIZE` lanes (`T0, T1, T2, T3` in the $N = 12, \mathrm{BLOCK_SIZE} = 4$ illustration) load their slice of the tile and add it into their private register slot in `acc`.
3. After all tiles have been processed, execute **one final block reduction** (`tl.sum(acc, axis=0)`) across the `BLOCK_SIZE` register slots—which Triton lowers via shared memory or warp shuffles—to produce the single scalar sum for the row.

Here is the complete tiled row-sum kernel in Triton:

```python
@triton.jit
def row_sum_kernel(x_ptr, out_ptr, N, BLOCK_SIZE: tl.constexpr):
    row = tl.program_id(0)                      # one block per row
    acc = tl.zeros([BLOCK_SIZE], dtype=tl.float32)

    for start in range(0, N, BLOCK_SIZE):       # loop over tiles along row
        cols = start + tl.arange(0, BLOCK_SIZE)
        mask = cols < N
        x = tl.load(x_ptr + row * N + cols, mask=mask, other=0.0)
        acc += x

    result = tl.sum(acc, axis=0)                # reduce BLOCK_SIZE -> scalar
    tl.store(out_ptr + row, result)
```

Notice two deliberate engineering choices in this kernel:

- **Reduction identity element:** Here the operation is summation, so masked tail lanes use `other=0.0` (the additive identity), contrasting directly with `other=float("-inf")` for softmax's max-plus-exp reduction.
- **`float32` accumulation:** `acc = tl.zeros([BLOCK_SIZE], dtype=tl.float32)` accumulates in 32-bit floating point to preserve numerical precision across long reductions, even if input values are stored in lower precision.

WORKED EXAMPLE

Trace `row_sum_kernel` on a single row of length $N = 10$ containing values `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` with `BLOCK_SIZE = 4`:

- **Initialization:** `acc = [0.0, 0.0, 0.0, 0.0]` (in registers).
- **Iteration 1 (`start = 0`):** `cols = [0, 1, 2, 3]`, `mask = [T, T, T, T]`. Loads `x = [1, 2, 3, 4]`. Updates `acc = [1.0, 2.0, 3.0, 4.0]`.
- **Iteration 2 (`start = 4`):** `cols = [4, 5, 6, 7]`, `mask = [T, T, T, T]`. Loads `x = [5, 6, 7, 8]`. Updates `acc = [6.0, 8.0, 10.0, 12.0]`.
- **Iteration 3 (`start = 8`):** `cols = [8, 9, 10, 11]`, `mask = [T, T, F, F]`. Loads `x = [9, 10, 0.0, 0.0]` (using `other=0.0`). Updates `acc = [15.0, 18.0, 10.0, 12.0]`.
- **Final block reduction:** `result = tl.sum(acc, axis=0) = 15.0 + 18.0 + 10.0 + 12.0 = 55.0`.
- **Store:** Writes scalar `55.0` to `out_ptr + row`.

Notice that cross-lane communication (`tl.sum`) happened only **once** at the very end, not inside the tile loop—minimizing synchronization while keeping the on-chip footprint fixed at `BLOCK_SIZE = 4` regardless of how large N grows.

KEY IDEA

Fusing row-wise softmax drops HBM traffic from $\approx 8MN$ to $2MN$ elements (up to 4× bandwidth speedup), provided masked lanes use the proper identity value (`-inf` for max/exp, `0.0` for sum). When a reduction dimension exceeds one block's capacity, looping over tiles into a `float32` register accumulator followed by a single final block reduction decouples on-chip memory footprint from row length.

This **tile-loop plus `float32` register accumulator** pattern is the exact bridge to the most important workload in deep learning: 2D matrix multiplication. Unlike a row sum—which reads each input element once and remains memory-bandwidth bound—matrix multiplication reuses every loaded row of A and column of B across many outputs, allowing 2D tiling to transform a memory-bound computation into a compute-bound one.

✎ Check Your Understanding

1. What would happen to the output of `row_sum_kernel` if `N` is not a multiple of `BLOCK_SIZE` and you mistakenly passed `other=float("-inf")` instead of `other=0.0`?
   Answer: On the final tile, masked tail lanes would load `-inf` into `acc`, and the final `tl.sum(acc, axis=0)` would produce `-inf` for the entire row sum instead of the true sum of the valid entries.
2. How many tile-loop iterations does `row_sum_kernel` execute per row when $N = 4{,}096$ and `BLOCK_SIZE = 1024`, and how many times is the cross-block reduction `tl.sum(acc, axis=0)` called per row?
   Answer: It executes $4{,}096/1024 = 4$ tile-loop iterations, and `tl.sum(acc, axis=0)` is called only **once** per row after the loop completes.

Section 7: 2D Tiled Matrix Multiplication and Epilogue Fusion in Triton

Matrix multiplication $C = AB$, where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$, is defined entrywise as:

$$C[m, n] = \sum_{k=0}^{K-1} A[m, k]\, B[k, n].$$

Computing all MN entries requires MNK multiplications and MNK additions—$2MNK$ total floating-point operations (FLOPs). Yet whether the GPU runs near its peak tensor-core throughput or crawls at HBM bandwidth speed depends entirely on **arithmetic intensity**: the ratio of FLOPs performed to bytes transferred from HBM.

Why Tiling Is Essential for Matrix Multiplication

Compare three ways to execute $C = AB$ on a GPU:

Approach	HBM Reads	HBM Writes	Arithmetic Intensity	Feasible?
Naive (compute 1 output $C[m, n]$ directly from HBM)	MKN pairs ($2MKN$ elements)	MN	$O(1)$	**Memory-bound**
Idealized (hold all of A, B in shared memory)	$MK + KN$	MN	$O(N)$	**Too large for 256 KB SMEM**
Tiled (load $B_M \times B_K$ and $B_K \times B_N$ tiles)	Reuse each tile across block	MN	$O(\text{tile_size})$	**Fits on-chip & compute-bound**

Why does the naive approach fail? If each thread computes a single output entry $C[m, n]$ by streaming row m of A and column n of B directly from HBM, it reads 2 elements for every 2 FLOPs (1 multiply + 1 add). Its arithmetic intensity is $O(1)$, so the 8 TB/s HBM interface bottlenecks the entire chip.

Why does the idealized approach fail? Reading each entry of A and B from HBM only once would achieve $O(N)$ arithmetic intensity, but large matrices cannot fit inside an SM's 256 KB L1/shared memory pool (Section 1).

2D tiling resolves both failures using a simple structural insight: outputs in the same row of C reuse the same row of A, and outputs in the same column of C reuse the same column of B. Therefore, we assign each thread block a $B_M \times B_N$ **output tile** of C, and march along the shared inner dimension K in steps of B_K:

1. Load one $B_M \times B_K$ tile A_k from A and one $B_K \times B_N$ tile B_k from B into on-chip memory **once**.
2. Compute the mini matrix product $A_k B_k$ (size $B_M \times B_N$) entirely on-chip and accumulate it into the block's register tile `acc`:

$$\text{acc} \leftarrow \text{acc} + A_k B_k.$$

3. Repeat for $k = 0, B_K, 2B_K, \ldots$ until $\text{acc} = \sum_k A_k B_k$ holds the complete $B_M \times B_N$ tile of C.

> **DEFINITION**
> **2D Tiling (for GEMM):** Partitioning the output matrix C into $B_M \times B_N$ tiles (one per thread block) and iterating along the inner dimension K in B_K-sized chunks so each loaded tile of A ($B_M \times B_K$) and B ($B_K \times B_N$) is reused across all outputs in the block, boosting arithmetic intensity from $O(1)$ to $O(\text{tile_size})$.

WORKED EXAMPLE

Compare HBM reads and arithmetic intensity for one 64×64 output tile ($B_M = 64, B_N = 64$) stepping with $B_K = 32$ on `float32` (4-byte) matrices:

1. **On-chip tile footprint per K-step:**
   - Tile A_k (64×32): $2{,}048$ elements $= 8$ KB.
   - Tile B_k (32×64): $2{,}048$ elements $= 8$ KB.
   - Total per step: 16 KB—easily fitting inside a B200 SM's 256 KB shared-memory pool.
2. **Compute per K-step:**

$$2 \times B_M \times B_N \times B_K = 2 \times 64 \times 64 \times 32 = 262{,}144 \text{ FLOPs}.$$

3. **HBM bytes read per K-step:**
   - **Tiled:** $8 \text{ KB} + 8 \text{ KB} = 16{,}384$ bytes.

$$\text{Arithmetic intensity (tiled)} = \frac{262{,}144 \text{ FLOPs}}{16{,}384 \text{ bytes}} = \mathbf{16} \text{ FLOPs/byte}.$$

   - **Naive** (same $64 \times 64 \times 32$ work without reuse): reads $2 \times 64 \times 64 \times 32 = 262{,}144$ floats $= 1{,}048{,}576$ bytes.

$$\text{Arithmetic intensity (naive)} = \frac{262{,}144 \text{ FLOPs}}{1{,}048{,}576 \text{ bytes}} = \mathbf{0.25} \text{ FLOPs/byte}.$$

Tiling with $64 \times 64 \times 32$ cuts HBM read traffic by **64×** (every element of A_k is reused across all $B_N = 64$ columns of C, and every element of B_k is reused across all $B_M = 64$ rows of C), lifting arithmetic intensity from 0.25 to 16 FLOPs/byte.

2D Tiled Matmul Dataflow and Epilogue Fusion

Visualize the full dataflow across matrices A ($M \times K$), B ($K \times N$), and C ($M \times N$):

- In matrix A, block `(pid_m, pid_n)` sweeps across a horizontal row strip of height B_M, sliding its active tile A_k ($B_M \times B_K$) from left to right along K in increments of `BLOCK_K`.
- In matrix B, the same block sweeps down a vertical column strip of width `BLOCK_N`, sliding its active tile B_k ($B_K \times B_N$) from top to bottom along K in increments of `BLOCK_K`.
- In matrix C, the block owns one fixed $B_M \times B_N$ yellow output tile holding `acc` on-chip, accumulating $\sum_k A_k B_k$. Inside Triton, each tile multiplication maps directly onto hardware **tensor cores** via `tl.dot(a, b)` (lowering to tensor-core MMA instructions).
- **Bonus epilogue fusion:** Once the K-loop finishes, the completed tile `acc` is already sitting in registers. Applying a pointwise activation—such as **ReLU** (`tl.maximum(acc, 0.0)`) or **GeLU**—to `acc` *before* writing C back to HBM executes with **zero extra HBM traffic**, completely eliminating what would otherwise be a separate activation kernel and full HBM round-trip.

DEFINITION

Epilogue Fusion: Performing post-processing operations (such as ReLU, GeLU, or bias addition) directly on the finished tile accumulator in registers right before storing the matrix-multiplication output to HBM, avoiding a separate kernel launch and extra HBM round-trip.

Addressing 2D Tiles: Linearized Memory and Strides

To load 2D tiles A_k and B_k in Triton, we must map logical 2D coordinates `(row, col)` onto flat 1D HBM memory addresses. HBM has no notion of rows or columns—it is simply a 1D array of addresses.

Consider a logical 2×4 matrix stored row-major:

- Row 0: `[0, 1, 2, 3]`
- Row 1: `[4, 5, 6, 7]`

In flat 1D HBM, those eight elements occupy consecutive offsets `+0, +1, +2, +3, +4, +5, +6, +7`. Moving down one row jumps forward by `stride_row = 4` elements; moving right one column jumps forward by `stride_col = 1` element. Any element `(row, col)` therefore lives at the linearized offset:

$$\text{index} = \text{row} \times \text{stride_row} + \text{col} \times \text{stride_col}.$$

Checking `(row = 1, col = 2)`: $\text{index} = 1 \times 4 + 2 \times 1 = 6$, which matches entry `6` at offset `+6` in the flat 1D layout.

DEFINITION

Stride: The number of elements (or memory units) to advance in a flat 1D buffer to move by 1 step along a given tensor dimension. Passing strides explicitly to a kernel allows the exact same kernel code to handle both row-major and transposed (column-major) inputs without copying data.

In Triton, **broadcasting** builds an entire 2D block of pointers in a single line:

```
a_ptrs = a_ptr + indices_m[:, None] * stride_am + indices_k[None, :] * stride_ak
```

Look closely at the shapes:

- `indices_m` has shape `[BLOCK_M]`; indexing with `[:, None]` turns it into a column vector of shape `[BLOCK_M, 1]`.
- `indices_k` has shape `[BLOCK_K]`; indexing with `[None, :]` turns it into a row vector of shape `[1, BLOCK_K]`.
- Adding `[BLOCK_M, 1]` and `[1, BLOCK_K]` broadcasts into a 2D pointer matrix `a_ptrs` of shape `[BLOCK_M, BLOCK_K]`, where entry `(i, j)` points directly to `A[indices_m[i], indices_k[j]]`.

Example 4: Complete Triton Matmul + ReLU Kernel

First, the host launcher sets tile sizes `BLOCK_M = 64, BLOCK_N = 64, BLOCK_K = 32`, creates a 2D grid covering all output tiles of C, and passes the pointers, dimensions, and all six strides:

```
# Host launcher: 2D grid of output tiles
M, K = a.shape
K, N = b.shape
c = torch.empty((M, N), device=a.device)
BLOCK_M, BLOCK_N, BLOCK_K = 64, 64, 32
grid = (triton.cdiv(M, BLOCK_M), triton.cdiv(N, BLOCK_N))
matmul_relu_kernel[grid](
    a, b, c, M, N, K,
    a.stride(0), a.stride(1), b.stride(0), b.stride(1), c.stride(0), c.stride(1),
    BLOCK_M, BLOCK_N, BLOCK_K,
)
```

Inside `@triton.jit matmul_relu_kernel`, each thread block reads its 2D block coordinates `(pid_m, pid_n)` and builds its initial 2D pointer tiles `a_ptrs` (shape `[BLOCK_M, BLOCK_K]`) and `b_ptrs` (shape `[BLOCK_K, BLOCK_N]`):

```
# Inside @triton.jit matmul_relu_kernel:
pid_m, pid_n = tl.program_id(0), tl.program_id(1)
indices_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
indices_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
indices_k = tl.arange(0, BLOCK_K)
a_ptrs = a_ptr + indices_m[:, None] * stride_am + indices_k[None, :] * stride_ak
b_ptrs = b_ptr + indices_k[:, None] * stride_bk + indices_n[None, :] * stride_bn
```

Next comes the K-loop, the fused ReLU epilogue, and the single masked store to C:

```
acc = tl.zeros([BLOCK_M, BLOCK_N], dtype=tl.float32)

for k in range(0, K, BLOCK_K):
    a = tl.load(a_ptrs, mask=(indices_m[:, None] < M) & (indices_k[None, :] + k < K),
        other=0.0)
    b = tl.load(b_ptrs, mask=(indices_k[:, None] + k < K) & (indices_n[None, :] < N),
        other=0.0)
    acc += tl.dot(a, b)
    a_ptrs += BLOCK_K * stride_ak      # advance along rows of A
    b_ptrs += BLOCK_K * stride_bk      # advance along cols of B

acc = tl.maximum(acc, 0.0)             # fused ReLU epilogue (zero extra HBM traffic)

c_ptrs = c_ptr + indices_m[:, None] * stride_cm + indices_n[None, :] * stride_cn
tl.store(c_ptrs, acc, mask=(indices_m[:, None] < M) & (indices_n[None, :] < N))
```

Notice how cleanly every earlier concept clicks together here:

- **2D boundary masks with `other=0.0`:** Handle arbitrary dimensions M, N, K that are not multiples of `BLOCK_M`, `BLOCK_N`, `BLOCK_K`. Because out-of-bounds entries are loaded as `0.0`, they multiply to zero inside `tl.dot(a, b)` and never corrupt the accumulated dot product.

- **Pointer advancement:** After each iteration, `a_ptrs += BLOCK_K * stride_ak` slides the A-tile right by `BLOCK_K` columns, and `b_ptrs += BLOCK_K * stride_bk` slides the B-tile down by `BLOCK_K` rows—saving recomputation of the base row/column terms on every step.
- **Tensor cores + epilogue fusion:** `tl.dot(a, b)` dispatches to tensor-core MMA instructions while accumulating in `tl.float32`, and `tl.maximum(acc, 0.0)` fuses ReLU directly in registers before the single masked HBM store.

COMMON MISCONCEPTION

You might think that advancing `a_ptrs += BLOCK_K` (without multiplying by `stride_ak`) and `b_ptrs += BLOCK_K * N` is always sufficient inside the K-loop. That breaks as soon as either input matrix is transposed or non-contiguous (where `stride_ak != 1` or `stride_bk != N`). Multiplying `BLOCK_K` by the explicit strides `stride_ak` and `stride_bk` guarantees correct advancement for any strided layout.

WORKED EXAMPLE

Trace `matmul_relu_kernel` on non-multiple dimensions $M = 100$, $N = 80$, $K = 50$ (row-major, so `stride_am = 50`, `stride_ak = 1`, `stride_bk = 80`, `stride_bn = 1`) with `BLOCK_M = 64, BLOCK_N = 64, BLOCK_K = 32`:

1. **Grid dimensions:**

$$\text{grid} = (\text{cdiv}(100, 64),\ \text{cdiv}(80, 64)) = (2, 2) \rightarrow 4 \text{ thread blocks total.}$$

2. **Focus on the bottom-right corner block `(pid_m = 1, pid_n = 1)`:**
   - `indices_m = 64 + [0..63] = [64..127]`. Valid rows (`< 100`): `64..99` (36 rows).
   - `indices_n = 64 + [0..63] = [64..127]`. Valid cols (`< 80`): `64..79` (16 cols).
   - `indices_k = [0..31]`.
3. **Iteration 1 (`k = 0`):**
   - `indices_k + 0 = [0..31] < 50` (all 32 K-positions valid).
   - `a` loads rows `64..99`, cols `0..31` (rows `100..127` padded with `0.0`).
   - `b` loads rows `0..31`, cols `64..79` (cols `80..127` padded with `0.0`).
   - `acc += tl.dot(a, b)` accumulates the first 32 terms of the inner product.
   - Pointer updates: `a_ptrs` advances by $32 \times 1 = 32$ elements; `b_ptrs` advances by $32 \times 80 = 2{,}560$ elements.
4. **Iteration 2 (`k = 32`):**
   - `indices_k + 32 = [32..63]`. Valid K-positions (`< 50`): `32..49` (18 positions); positions `50..63` are masked and filled with `0.0`.
   - `acc += tl.dot(a, b)` completes all 50 inner-product terms; the 14 padded K-columns in `a` and 14 padded K-rows in `b` contribute $0.0 \times 0.0 = 0.0$.
5. **Epilogue & Store:**

- `acc = tl.maximum(acc, 0.0)` applies ReLU in registers.
- `tl.store` writes only the valid 36×16 corner (`rows 64..99`, `cols 64..79`) to C, leaving memory outside `(M, N)` untouched.

Putting the entire lecture together: fast GPU kernels start from the B200 hardware mental model (148 SMs, 8 TB/s HBM vs. ~40 TB/s shared memory vs. ~447 TB/s registers, warps, occupancy, coalescing, bank conflicts, and wave quantization), stay grounded through disciplined measurement (warmed-up CUDA-event benchmarking, PyTorch Profiler and Nsight profiling, CUDA Graphs and persistent megakernels for small-batch overhead, and CI batch-invariance testing), select the right abstraction on the modern kernel DSL continuum, and apply the four canonical Triton patterns—elementwise fusion (GeLU), single-block reduction (softmax), tiled reduction (row sum), and 2D tiled matrix multiplication with fused epilogue (matmul + ReLU).

KEY IDEA

2D tiling transforms matrix multiplication from an $O(1)$ memory-bound operation into an $O(\text{tile_size})$ compute-bound operation by reusing each $B_M \times B_K$ tile of A and $B_K \times B_N$ tile of B across an entire $B_M \times B_N$ output block on-chip. Passing explicit strides supports arbitrary transposed layouts via 2D pointer broadcasting, 2D masking with `other=0.0` handles non-multiple shapes cleanly, and epilogue fusion applies activations like ReLU in registers at zero extra HBM cost.

✎ Check Your Understanding

1. A 4×6 matrix is stored in row-major order (`stride_row = 6`, `stride_col = 1`). Using the linearized addressing formula $\text{index} = \text{row} \times \text{stride_row} + \text{col} \times \text{stride_col}$, what flat 1D offset corresponds to element `(row = 2, col = 5)`?
   Answer: $\text{index} = 2 \times 6 + 5 \times 1 = 17$.
2. Does applying `acc = tl.maximum(acc, 0.0)` inside `matmul_relu_kernel` right before `tl.store` increase HBM memory traffic compared to computing `C = A @ B` alone? Justify your answer.
   Answer: No. Because `acc` is already resident in on-chip registers when the K-loop finishes, applying ReLU before the single `tl.store` executes entirely in registers with zero extra HBM reads or writes.