Lecture 5: GPU

Marshall Meng and Dylan Fang

Stanford CS336 - Spring 2025

1 Why GPUs Matter

The lecture introduces Graphics Processing Units (GPUs) as the key hardware enabling the scaling of large language models (LLMs).

Compute Drives Performance For LLMs, increasing compute power (measured in PetaFLOP/s-days) leads to predictable improvements in performance (lower validation loss).

The End of Old Scaling For decades, computers became faster "for free" due to **Moore's Law** and **Dennard Scaling**. Moore's Law increased transistor counts exponentially, while Dennard Scaling kept power usage per transistor constant, allowing clock speeds to rise without overheating. This era (before ∼2005) was known as the **Golden Age of Scaling**, where transistor density, frequency, and single-thread performance all improved together.

However, around the mid-2000s, Dennard Scaling broke down - transistors could no longer reduce voltage and power proportionally as they shrank. Power density rose sharply, limiting further frequency gains. The "free lunch" of automatic single-thread speedups ended, pushing the industry toward **parallelism and GPUs** for continued performance scaling.

Parallel Scaling is Key Modern performance gains come from massive parallelization, where GPUs have provided over a 1000x improvement in 10 years.

2 How GPUs work

2.1 CPU vs. GPU

- **CPUs (Central Processing Unit)**: Optimized for low latency. They use a few, powerful cores with large caches and complex control logic to execute a single thread as fast as possible.
- **GPUs (Graphic Processing Unit)**: Optimized for high throughput. They consist of many tiny compute units (ALUs) and less cache/control logic. They are designed to run thousands of threads in parallel.
- **Latency Hiding**: When a thread on a GPU is "Waiting for data" (a memory operation), the processor instantly switches to another "Ready" thread, hiding the memory latency and keeping the compute units busy.

2.2 Key Architecture Components

A GPU is composed of many **SMs (streaming multiprocessors)**. Each SM is an independent processor that can execute "blocks" of work. Each SM contains many **SPs (streaming processors also called CUDA cores)** that execute the actual "threads" in parallel.

2.2.1 The Memory Hierarchy A core challenge in GPU programming is the **memory wall**: *compute throughput (FLOPs) has scaled much faster than memory bandwidth.* This creates a bottleneck-keeping thousands of compute cores "fed" with data is often harder than performing the actual computation.

Note

Key Rule: The closer the memory is to the thread, the faster it is.

Understanding the GPU memory hierarchy is essential for performance optimization:

- **Registers - Fastest** *Location:* Inside each Streaming Multiprocessor (SM), private to each thread. *Latency:* Essentially 0-cycle access; used for temporary per-thread variables.
- **L1 Cache & Shared Memory (SRAM) - Very Fast** *Location:* Inside each SM, shared by all threads in a block. *Latency:* Very low (∼23–33 cycles). *Bandwidth:* Extremely high (e.g., 19 TB/s), roughly 8× faster than global memory. Shared memory is limited (e.g., 20 MB per SM) and expensive in area.
- **L2 Cache - Fast** *Location:* On the GPU die, shared across all SMs. *Latency:* Moderate (∼200 cycles). Acts as an intermediate buffer between SMs and global memory.
- **Global Memory (DRAM/HBM) - Slowest** *Location:* External to the GPU die, on the circuit board. *Capacity:* Large (e.g., 40 GB) but slow (∼1.5 TB/s bandwidth, ∼290 cycles latency). Serves as the main "warehouse" for all data.

Note

Because compute units operate orders of magnitude faster than global memory, efficient GPU programming focuses on **data locality**-reusing data in registers or shared memory to minimize slow global memory accesses.

2.2.2 Execution Model (CUDA) GPUs use a hierarchical execution model designed for massive parallelism. A CUDA program is organized into a grid of **Blocks**, each containing multiple **Warps** of **Threads**. The rule of thumb: *one block runs entirely on one Streaming Multiprocessor (SM)*, but each SM can execute many blocks concurrently.

- **Blocks:** The largest organizational unit - a group of threads assigned to a single SM. All threads in a block can cooperate via fast on-chip shared memory.
- **Warps:** Each block is divided into **warps** of 32 consecutive threads. Warps are the units scheduled by the SM's *warp schedulers*. All threads in a warp execute the same instruction simultaneously on different data - the **SIMT (Single Instruction, Multiple Threads)** model.
- **Threads:** The smallest execution unit. Each thread performs computation on its own data, using private registers and executing the instructions assigned to its warp.

Example

Toy Example - Vector Addition Suppose we want to compute $C[i] = A[i] + B[i]$ for 1,024 elements.

- **Threads:** Assign one thread per element (1,024 total). Each thread performs a single addition independently.
- **Blocks:** With 256 threads per block, we need 4 blocks in total: Block 0 $\rightarrow$ Threads 0–255, Block 1 $\rightarrow$ 256–511, Block 2 $\rightarrow$ 512–767, Block 3 $\rightarrow$ 768–1023. Each block runs on one SM.
- **Warps:** Within each block, threads are grouped into 8 warps ($256 \div 32 = 8$). Warp 0 executes threads 0–31 together, Warp 1 executes 32–63, and so on. Warps execute in parallel following the SIMT model.

Summary: Grid = 4 Blocks, each Block = 256 Threads, each Warp = 32 Threads.

2.2.3 Memory Model of a GPU The GPU's memory model defines which part of the system can access each type of memory. It separates the **Host (CPU)** from the **Device (GPU)**, and organizes memory by scope: from shared and slow (global) to private and fast (registers).

Host (CPU)

The **Host** acts as the *manager* of GPU computation. Host code (e.g., a Python program using PyTorch or TensorFlow) prepares and launches GPU work.

- **Setting up:** Allocates GPU memory and copies data from CPU RAM to the GPU's **Global Memory**.
- **Defining the job:** Specifies the grid and block configuration (e.g., 4 blocks $\times$ 256 threads).
- **Launching:** Invokes the kernel on the GPU.
- **Collecting results:** Waits for execution to finish, then copies results from GPU back to CPU memory.

Device (GPU)

The **Device** executes the *kernel*-parallel function running across thousands of threads on Streaming Multiprocessors (SMs).

- Each thread uses its own **Registers** (private storage).
- Threads in the same block share fast, on-chip **Shared Memory**.
- All threads across all blocks can access large, slow **Global Memory** and read-only **Constant Memory**.

Example

Analogy - Head Chef vs. Line Cooks The **Host (CPU)** is the *Head Chef*: it writes the recipe (device code), stocks the pantry (**Global Memory**), and tells the cooks to start. The **Device (GPU)** is the team of *Line Cooks (Threads)*: each has a personal cutting board (**Registers**), shares a spice bowl with nearby cooks (**Shared Memory**), and places the final dishes on the pass (**Global Memory**) for the chef to pick up.

2.2.4 Strengths of the GPU Model GPUs are highly effective for large-scale parallel computation because of their architecture and execution model.

- **Scalability:** The GPU architecture scales simply by adding more **Streaming Multiprocessors (SMs)** to the chip-each SM contributes additional compute and memory resources.
- **SIMT Model:** GPUs follow the **SIMT (Single Instruction, Multiple Threads)** paradigm: one instruction stream operates simultaneously across many threads, each on different data. This makes parallel programming conceptually simple yet extremely powerful.
- **Latency Hiding:** GPU threads are lightweight and numerous. When one warp stalls waiting for data from slow global memory, the SM immediately switches to another ready warp. This context switching occurs with near-zero overhead, keeping the compute units busy and effectively "hiding" memory latency.

GPUs as Fast Matrix Multipliers (Historical Context)

Originally, GPUs were built for graphics rendering, not for general-purpose computation. Early researchers discovered that the same hardware used for pixel shading could be "hacked" to perform matrix multiplications.

- Early NVIDIA GPUs exposed *programmable shaders* for graphics.
- Researchers encoded matrices as pixel colors (R, G, B, Alpha channels) and used shader programs to perform math operations.
- This innovation led to the birth of **GPGPU** (General-Purpose GPU computing), paving the way for today's AI accelerators.

Matmuls Are Fast and Special

Modern GPUs have evolved from those early hacks into specialized numerical engines.

- Matrix multiplications (**matmuls**) dominate AI workloads.
- Starting with the NVIDIA V100, GPUs introduced **Tensor Cores**-dedicated circuits optimized for dense matrix operations.
- As a result, matmuls are now over **10× faster** than other arithmetic on recent GPUs (A100, H100). This specialization explains why deep learning, built largely on matmuls, runs so efficiently on GPUs.

The Memory Wall - Compute vs. Memory Scaling

Despite exponential gains in compute power, memory performance has not kept up.

- Over the last 20 years, **Compute Throughput (FLOPs)** has increased by roughly **60,000×**, while **Memory Bandwidth (DRAM BW)** has improved only about **100×**.
- This disparity creates the **Memory Wall**-a bottleneck where compute units idle, waiting for data.
- GPUs survive this limitation through massive parallelism and **latency hiding**: when one warp waits for data, others continue executing.

Note

Summary: GPU performance relies on three key principles - parallel scalability, SIMT execution, and latency hiding. However, the growing gap between compute and memory bandwidth (the "Memory Wall") remains the main challenge for future scaling.

3 Making GPUs fast

This section covers the essential techniques for optimizing GPU performance, all of which aim to minimize slow memory accesses The **Roofline Model** helps identify whether a program is limited by **compute** or by **memory** performance.

- **Y-axis (Throughput):** Program speed - how many FLOPs are performed per second.
- **X-axis (Operational Intensity):** Ratio of FLOPs to memory bytes accessed.
  - **Low intensity:** Memory-heavy tasks (e.g., vector addition).
  - **High intensity:** Compute-heavy tasks (e.g., matrix multiplication).

The roofline has two limits:

- **Horizontal lines:** Hardware's peak compute speed - cannot be exceeded.
- **Diagonal lines:** Memory bandwidth limits - bound memory access speed.

Two Zones:

- **Memory-bound:** Low intensity; performance limited by data transfer speed.
- **Compute-bound:** High intensity; limited by peak processing speed.

Note

Key takeaway: GPU optimization means moving from the sloped (memory-bound) region to the flat "roof," by increasing data reuse and arithmetic intensity.

3.1 Problem: Control divergence

Control Divergence occurs when threads within the same **warp** (32 threads) take different control paths in code, such as an `if{else` branch. Because of the GPU's **SIMT (Single Instruction, Multiple Threads)** model, all threads in a warp must execute the same instruction at once.

- **Cause:** In SIMT, the warp scheduler issues one instruction for all 32 threads. If threads evaluate a condition differently, the warp cannot follow both branches simultaneously.
- **Effect:** The warp must **serialize** execution:
  1. Threads taking the `if` path run first, while others are paused (masked off).
  2. Then, threads taking the `else` path execute while the first group waits.
  3. Finally, the warp **reconverges** and resumes executing together.

  This wastes parallel resources - only a portion of the warp is active at a time, reducing throughput.

3.2 Low precision computation

Low precision computation is one of the key techniques that make GPUs faster and more efficient.

What It Is Instead of using standard 32-bit floating-point numbers (FP32), GPUs increasingly use lower-precision formats such as **FP16/BF16** (16-bit floats) or **INT8** (8-bit integers). Fewer bits mean less data to move and process.

- **Reduced Memory Transfer:** Smaller numbers require fewer bits to move. Since performance is often memory-bound, halving the data size (FP32 $\rightarrow$ FP16) roughly doubles efficiency.
- **Higher Arithmetic Intensity:** With smaller operands, the ratio of computation (FLOPs) to memory access increases. For example, a ReLU on FP16 data uses half the bandwidth for the same FLOP, effectively doubling intensity.

How It's Implemented (Mixed Precision) Modern GPUs use specialized **Tensor Cores** that perform computation in mixed precision:

- Multiplications are done using fast 16-bit inputs.
- Results are accumulated in higher-precision **FP32** registers.
- This keeps accuracy while benefiting from the speed and lower memory cost of FP16 math.

The Trade-off

- **Good for:** Matrix multiplications and simple operations (e.g., ReLU, convolution).
- **Bad for:** Precision-sensitive tasks such as softmax, loss computation, or operations involving many small sums - these still require FP32.

3.3 Operator Fusion

Operator Fusion is a compiler optimization that combines multiple sequential operations (e.g., `add`, `sin`, `cos`) into a single, *fused kernel*. This reduces the number of memory reads and writes, keeping data in fast on-chip registers instead of repeatedly accessing slow global memory.

The Problem - The Memory Wall Memory access is much slower than computation; GPUs often wait for data to arrive from memory. In a naïve (non-fused) computation such as computing $\sin^2(x) + \cos^2(x)$, each intermediate result is written back to and reloaded from memory multiple times:

$$\texttt{load} \rightarrow \sin(x) \rightarrow \texttt{store} \rightarrow \texttt{load} \rightarrow \sin^2(x) \rightarrow \texttt{store} \ldots$$

Each of these steps triggers redundant global memory traffic, wasting bandwidth and time.

The Solution - Fused Kernel **Operator Fusion** merges these operations into one kernel:

1. Load x from memory once.
2. Compute $\sin(x)$, $\cos(x)$, and their squares in fast local registers.
3. Add results directly from registers.
4. Write the final output back to memory once.

This approach avoids unnecessary data transfers, dramatically reducing latency and memory pressure.

3.4 Recomputation

Recomputation, also known as **activation checkpointing**, trades cheap compute for expensive memory access. Instead of storing all intermediate activations during the forward pass, the GPU discards them. When needed in the backward pass, those activations are simply recomputed from the original inputs.

The Problem - Naïve Method During backpropagation, intermediate activations (e.g., s_1, s_2) are required to compute gradients.

- **Forward:** Read x (1 read), compute s_1, s_2, and *out*, and write all three to global memory (3 writes).
- **Backward:** Read s_1, s_2, and *dout* (3 reads), then write the gradient dx (1 write).

Total: 8 slow memory operations (4 reads + 4 writes).

The Solution - Recomputation Recomputation avoids saving intermediate values:

- **Forward:** Read x (1 read), compute s_1, s_2, and *out*, but only write the final *out* (1 write).
- **Backward:** Read x and *dout* (2 reads), recompute s_1 and s_2 inside fast registers, compute gradients, and write dx (1 write).

Total: 5 memory operations - saving 3 expensive memory accesses by doing a bit of extra compute.

3.5 Memory Coalescing

Memory coalescing is a memory access pattern that maximizes bandwidth efficiency. An access is *coalesced* when all 32 threads in a warp read consecutive addresses that fall within the same 128-byte **burst section** of Global Memory. The memory controller always fetches an entire burst, so grouping nearby accesses together minimizes the number of transactions.

Global Memory is physically organized into fixed-size bursts (like 128-byte "shelves"). If each thread requests nearby data, all 32 values can be served in one burst. If threads access scattered locations, the GPU must fetch multiple bursts, wasting bandwidth and time.

Example 1 – Coalesced (Fast) A warp of four threads (T_0–T_3) processes the same row of a row-major matrix:

$$T_0 \to M_{0,0}, \quad T_1 \to M_{0,1}, \quad T_2 \to M_{0,2}, \quad T_3 \to M_{0,3}.$$

Because these elements are stored consecutively in memory, all addresses fall within a single burst section. The GPU issues **one** memory transaction to load all four values-this is **coalesced** and very fast.

Example 2 – Not Coalesced (Slow) Now each thread works on a different row:

$$T_0 \to M_{0,0}, \quad T_1 \to M_{1,0}, \quad T_2 \to M_{2,0}, \quad T_3 \to M_{3,0}.$$

In row-major storage these addresses are far apart, separated by full row lengths. Each access lands in a different burst section, forcing the GPU to perform **four separate memory requests**. This is **not coalesced** and much slower.

Why It Matters Coalesced access can reduce 32 individual memory operations to one, providing up to a 32× improvement in effective bandwidth. Efficient GPU kernels are carefully written so that threads in a warp access consecutive memory locations whenever possible.

3.6 Tiling (Using Shared Memory for Data Reuse)

Core Idea - "Work Smarter, Not Harder" The main performance bottleneck on GPUs is that **Global Memory** (DRAM) is very slow, while **Shared Memory** (on-chip SRAM) is extremely fast. **Tiling** is a programming strategy that minimizes slow memory accesses by reusing data in fast memory.

1. Load a small **tile** (block) of data from slow Global Memory ("the warehouse").
2. Store it temporarily in fast Shared Memory ("the workbench").
3. Perform as many computations as possible using the data in Shared Memory.
4. Write only the final results back to Global Memory and move on to the next tile.

This approach greatly reduces the number of global memory reads and writes.

1. The Problem - Naïve Matrix Multiplication

Without tiling, computing each element of the output matrix P requires repeatedly reading entire rows of M and entire columns of N from Global Memory.

$$P_{i,j} = \sum_{k=0}^{N-1} M_{i,k} \times N_{k,j}.$$

- To compute $P_{0,0}$: read all of Row 0 (from M) and Column 0 (from N).
- To compute $P_{0,1}$: read Row 0 (again) and Column 1.
- To compute $P_{1,0}$: read Row 1 and Column 0 (again).

Issues:

1. **No Data Reuse:** The same rows and columns are read from Global Memory repeatedly.
2. **Not Coalesced:** Reading rows is coalesced, but reading columns is scattered and slow.

2. The Solution - Tiling

Tiling divides the computation into phases where each phase operates on small blocks (tiles) of M and N.

Phase 1:

- **Load:** All threads in a block cooperatively load one tile from M and one tile from N into fast Shared Memory (SHM).
- **Compute:** The threads compute the partial matrix product using only these tiles, keeping partial results in registers.

Phase 2:

- **Load:** The threads load the next pair of tiles (next segment of M and N) into SHM.
- **Compute:** Multiply the new tiles and accumulate their result into the previous partial sum.

  This process repeats until the full output matrix block is computed.

Benefits:

- **Data Reuse:** Each element is loaded from slow Global Memory once per tile and reused T times from fast Shared Memory.
- **Coalesced Access:** The tile loading step is organized so all memory reads are fully coalesced (threads read adjacent memory addresses).

3. Quantitative Benefit

Let N be the matrix dimension and T the tile size (e.g., $T = 32$).

- **Non-Tiled:** Each element of M and N is read N times from slow Global Memory.
- **Tiled:** Each element is read only $\frac{N}{T}$ times - once per tile in its row or column.

  Once loaded into Shared Memory, each tile is reused T times by the threads in the block.

$$\text{Speedup factor } \approx T.$$

Example: If $N = 1024$ and $T = 32$, each element is reused $32\times$ from Shared Memory, reducing global memory reads by a factor of 32 - a dramatic improvement in effective bandwidth.

3.7 The Matrix Mystery

The "Matrix mystery" refers to why the performance graph for a simple square matrix multiply ($N \times N$) is not a smooth, increasing line. Instead, the graph is complex, jagged, and shows multiple distinct "families" of curves.

This behavior is explained by three main factors:

- **Compute Intensity (The General Trend):** The overall performance (TF/s) increases as the matrix size (N) grows. This is because larger problems are more efficient. They can keep all the GPU's Streaming Multiprocessors (SMs) busy (a state called **high occupancy**) and the initial "startup cost" of launching the kernel becomes a smaller fraction of the total time.
- **Part 1: Tiling & Alignment (The "Jumps"):** The graph shows several distinct "families" of curves. This is explained by **memory alignment**.
  - **Fast Curves (Aligned Layout):** When the matrix size N is a "nice" multiple of the tile size (e.g., divisible by 32 or 128), the tiles are "aligned" with the memory's burst sections. This allows for fast, **coalesced** memory access ("One Nice Tile").
  - **Slow Curves (Unaligned Layout):** When N is a "bad" size, the tiles "straddle" multiple burst sections ("Two Bad Tiles"). This forces the GPU to make multiple slow, uncoalesced memory requests to get the data for a single tile, dropping performance to a lower, slower curve.
- **Part 2: Wave Quantization (The "Jagged Waves"):** This explains the periodic, sharp drops in performance. It is caused by **wasted work** when the total number of tiles doesn't fit the GPU's core count.
  - **The Cause:** A GPU has a fixed number of SMs (e.g., an A100 has 108). It tries to execute a "wave" of tiles all at once.
  - **Example (at N=1792):** Using a 256×128 tile, this matrix size divides *perfectly* into **98 tiles** (7×14). The A100 can execute all 98 tiles in **one single, efficient wave** (since 98 is less than 108).
  - **Example (at N=1793):** This tiny change means the job no longer divides evenly, requiring **120 tiles** (8×15) to cover the matrix. The A100 must run this in **two waves**: a full wave of 108 tiles, and a second, highly inefficient wave for the 12 remaining tiles. This second wave is mostly wasted work and causes the sharp performance drop.

4 Flash Attention

Key idea: FlashAttention (Dao et al., 2023) dramatically accelerates the attention operation by **avoiding the explicit materialization of the $n \times n$ attention matrix**. Instead, it performs the computation **tile-by-tile entirely in on-chip memory** (registers and shared memory), using an **online softmax** to maintain exactness.

1. The problem with standard attention

Standard attention computes:

$$S = \frac{QK^\top}{\sqrt{d}}, \quad P = \mathrm{softmax}(S), \quad O = PV.$$

This approach is **memory-bound**, not compute-bound, because the full score matrix $S \in \mathbb{R}^{n \times n}$ must be written to and read from high-bandwidth memory (HBM) multiple times:

- Write S after computing $QK^\top$.
- Read S again for the softmax.

- Possibly read/write again for PV.

Although the number of FLOPs is acceptable, the huge number of HBM accesses dominates runtime.

2. Core idea: Never store the full S

FlashAttention eliminates the bottleneck by:

- Splitting the computation into **tiles** that fit in fast on-chip memory.
- Performing all intermediate operations (`matmul`, `softmax`, and `value mixing`) on those tiles before discarding them.

3. Tiling for KQV multiplication

To understand the benefit of FlashAttention's tiling, it's crucial to compare its workflow to the old, un-fused method. The primary bottleneck is the $O(N^2)$ attention matrix $S = QK^\top$.

Workflow of Standard Attention (The Slow Method)

The traditional implementation of attention is *memory-bound*. It is limited by the speed of reading and writing to HBM (global memory), not by the speed of computation.

- **Step 1:** Compute the entire, massive $N \times N$ matrix $S = QK^\top$.
- **Step 2 (The Bottleneck):** Write this entire S matrix from fast SRAM to slow **HBM**. This is an $O(N^2)$ memory operation.
- **Step 3 (The Bottleneck):** Read the entire S matrix back from HBM into SRAM to compute the softmax.
- **Step 4:** Compute $P = \text{softmax}(S)$ and write this $N \times N$ matrix back to HBM.
- **Step 5:** Read P and V from HBM to compute the final output $O = PV$.
- **Problem:** The process is stalled waiting for the $O(N^2)$ write and read of S (and P), which is much slower than the compute operations.

Workflow of FlashAttention (The Tiled & Fused Method)

FlashAttention re-engineers this into a single, *fused kernel* that avoids the $O(N^2)$ memory bottleneck. It processes the Q matrix in blocks. For each *query tile* Q_i, it streams through *key/value tiles* (K_j, V_j):

1. **Load Tiles:** Load the small tiles (Q_i, K_j, V_j) from slow HBM into fast on-chip SRAM.
2. **Compute Partial Scores:** Compute the scores for just this tile, S_{ij}. This calculation happens *entirely within SRAM.*

$$S_{ij} = \frac{Q_i K_j^\top}{\sqrt{d}}.$$

3. **Fuse & Fold:** This is the key trick. Instead of saving S_{ij}, the kernel *immediately* uses it. It performs the softmax math (using an "online" method to find the max, exponentiate, and sum) and multiplies the result by the V_j tile. This partial output is folded into a *running output accumulator* O_i that stays in fast registers.

4. **Discard & Loop:** The intermediate score S_{ij} is never written to HBM and is immediately discarded from SRAM. The kernel then loads the next tile (K_{j+1}, V_{j+1}) and repeats the process, accumulating the results into the same O_i block.

This method reduces HBM read/writes from $O(N^2)$ to $O(N)$ (for the final O matrix) because the giant S matrix is computed and consumed "on-the-fly" in small pieces.

4. Online (incremental) softmax (with the "previous sum" trick)

Goal. Compute the exact, numerically-stable softmax for a sequence $\{x_1, x_2, \ldots, x_N\}$ in a *single pass*, without storing the entire vector. We maintain two running accumulators:

$$m^{\text{old}} \text{ (running max)}, \qquad D^{\text{old}} \text{ (running denominator)}.$$

When a new element x_j arrives, we update in four steps:

1. **Update the max (for stability):**

$$m^{\text{new}} = \max(m^{\text{old}}, x_j).$$

2. **Compute the exponential of the new element (referenced to the new max):**

$$p'_j = e^{x_j - m^{\text{new}}}.$$

3. **The Previous Sum (The "Trick").** We must express the *old* denominator, which was computed using m^{old}, in the coordinates of the new maximum m^{new}.

   Notation:

$$D^{\text{old}} = \sum_{i=1}^{j-1} e^{x_i - m^{\text{old}}} \quad \text{(old denominator; old max } m^{\text{old}}\text{)}.$$

   We want to compute

$$D^{\text{old}\to\text{new}} = \sum_{i=1}^{j-1} e^{x_i - m^{\text{new}}} \quad \text{(the same sum but referenced to the new max)}.$$

   Rescaling identity:

$$e^{x_i - m^{\text{new}}} = e^{x_i - m^{\text{old}} + m^{\text{old}} - m^{\text{new}}} = \left(e^{x_i - m^{\text{old}}}\right) \cdot e^{m^{\text{old}} - m^{\text{new}}}.$$

   Therefore,

$$D^{\text{old}\to\text{new}} = \sum e^{x_i - m^{\text{new}}} = \sum e^{x_i - m^{\text{old}} + m^{\text{old}} - m^{\text{new}}} = \left(e^{m^{\text{old}} - m^{\text{new}}}\right) \sum e^{x_i - m^{\text{old}}} = D^{\text{old}} \cdot e^{m^{\text{old}} - m^{\text{new}}}.$$

   Thus the update rule becomes:

$$D^{\text{new}} = \underbrace{D^{\text{old}} \cdot e^{m^{\text{old}} - m^{\text{new}}}}_{\text{previous sum, rescaled}} + \underbrace{e^{x_j - m^{\text{new}}}}_{\text{current term}}.$$

4. **Commit the new state:** $m^{\text{old}} \leftarrow m^{\text{new}}, \quad D^{\text{old}} \leftarrow D^{\text{new}}$.

Finish. After the final element x_N is processed, the exact softmax is computed as:

$$\boxed{\text{softmax}(x)_j = \frac{e^{x_j - m^{\text{new}}}}{D^{\text{new}}}, \quad j = 1, \ldots, N.}$$

Two sanity cases:

- If the max does not change ($m^{\text{new}} = m^{\text{old}}$), the factor $e^0 = 1$ and the update reduces to a simple addition: $D^{\text{new}} = D^{\text{old}} + e^{x_j - m_j}$.
- If a larger max is found ($m^{\text{new}} > m^{\text{old}}$), the factor $e^{m^{\text{old}} - m^{\text{new}}} < 1$ correctly rescales previous contributions so that all terms are expressed relative to the new max.

5. Forward-pass fusion

FlashAttention fuses the following inside one kernel:

- Tile-wise computation of inner products ($QK^\top$).
- Immediate application of scaling and exponentiation.
- Online softmax accumulation (the "telescoping" sum trick).
- Accumulation of PV contributions.

All of these steps occur entirely within on-chip memory, avoiding intermediate global writes.

6. Backward pass

During backpropagation, FlashAttention does not store intermediate P or S matrices. Instead, it reuses the stored per-row statistics (m_i, ℓ_i) and **recomputes tiles on-the-fly**. This keeps memory usage at $O(nd)$ rather than $O(n^2)$.

7. Why it's fast

- **HBM traffic drops** from quadratic to linear in sequence length.
- **Higher compute intensity:** more FLOPs per byte fetched.
- **Kernel fusion:** fewer kernel launches and no intermediate writes.
- **Alignment effects:** performance steps up when tile sizes align with Tensor Core/warp shapes ("wave quantization").

8. Summary

- FlashAttention is **exact**, not approximate.
- It leverages **tiling, online softmax, and recomputation** to achieve **sub-quadratic memory access**.
- The main gain is **I/O-awareness**: it optimizes data movement, not just math.