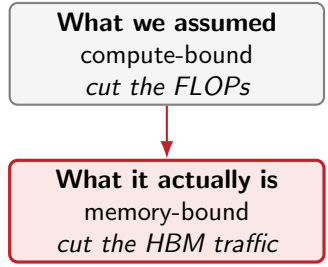
Attention got the FLOPs right and the memory wrong

Self-attention is $O(N^2)$ **in both time and memory**. As context N grows, the $N \times N$ attention matrix dominates.

The surprise: profiling shows the bottleneck is **HBM memory bandwidth**, *not* arithmetic. The matmuls are cheap; you wait on memory traffic.

FlashAttention makes attention **exact** and much **faster** by attacking the memory, not the math – tile the computation so the $N \times N$ matrix **never touches slow memory**.

Today: why attention is memory-bound, the **tiling + online-softmax** trick, **recompute-in-backward**, and the payoff (up to ~3x end-to-end, up to 20x less memory).



exact result, a fraction of the wall-clock

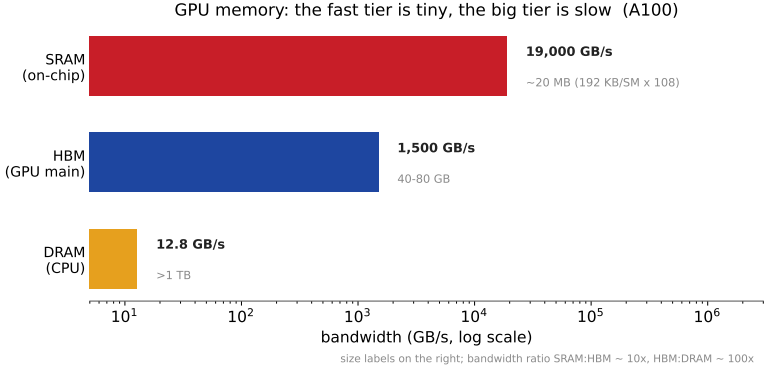
Outline

Attention is memory-bound, not compute-bound

The fast memory is tiny; the big memory is slow. That gap is the whole story.

The GPU memory hierarchy: fast is small, big is slow

A GPU has a few tiers of memory. Smaller means faster – by orders of magnitude.



On an A100, on-chip **SRAM** is ∼10x the bandwidth of **HBM** but ∼2000x smaller. If an operation reads/writes a lot of HBM relative to its arithmetic, HBM is what you **wait on**. The trick is to keep work in SRAM.

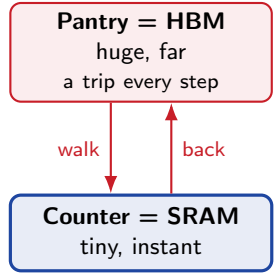
Why "memory-bound"? A kitchen analogy

Think of computing as **cooking**:

- **SRAM** = your **countertop** – tiny, but everything on it is instantly in reach.
- **HBM** = the **pantry down the hall** – huge, but every trip costs time.

Standard attention writes the giant $N \times N$ tray to the pantry and walks back for it at *every* step. The chopping (FLOPs) is quick; the **walking** (HBM traffic) is the wall-clock.

FlashAttention brings one **block** to the counter, finishes all the work on it there, and carries only the finished dish (O) back.

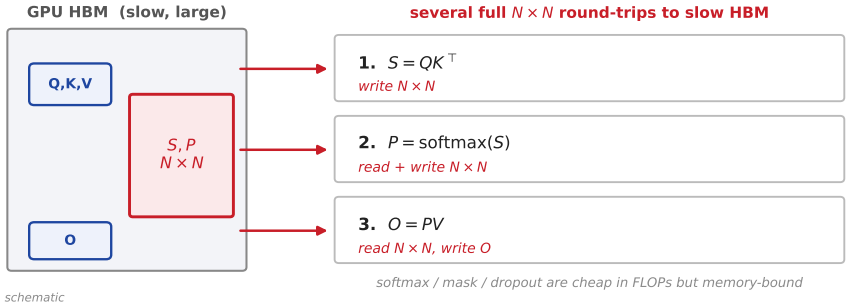


fewer trips = faster, even if you chop a bit more

The lever: minimize **trips to the pantry** (HBM traffic), not the amount of **chopping** (FLOPs). That is the whole design.

Standard attention round-trips the $N \times N$ through HBM

The textbook implementation materializes the full $N \times N$ matrices S and P in HBM:



$S = QK^\top$, $P = \mathrm{softmax}(S)$, $O = PV$: several full $N \times N$ **reads and writes** to slow HBM. Softmax, masking, and dropout add almost no FLOPs – but each is another memory-bound pass over the $N \times N$ matrix.

Predict first: which is the real bottleneck?

Self-attention is $O(N^2)$ **FLOPs** *and* $O(N^2)$ **memory**. For GPT-2 sizes ($N = 1024$, $d = 64$), which one dominates the wall-clock?

(a) the arithmetic – the $QK^\top$ and PV matrix multiplies

(b) the memory traffic – moving the $N \times N$ matrix in and out of HBM

Predict first: which is the real bottleneck?

Self-attention is $O(N^2)$ **FLOPs** *and* $O(N^2)$ **memory**. For GPT-2 sizes ($N = 1024$, $d = 64$), which one dominates the wall-clock?

(a) the arithmetic – the $QK^\top$ and PV matrix multiplies

(b) the memory traffic – moving the $N \times N$ matrix in and out of HBM

(b) memory. On GPT-2 medium, standard attention runs **66.6 GFLOPs** but moves **40.3 GB** through HBM, taking **41.7 ms**. The arithmetic is a small slice; the HBM traffic is the wall-clock. Attention is a **memory-bound** operation, so cutting FLOPs barely helps – you have to cut HBM accesses.

Tile it, and never materialize the $N \times N$

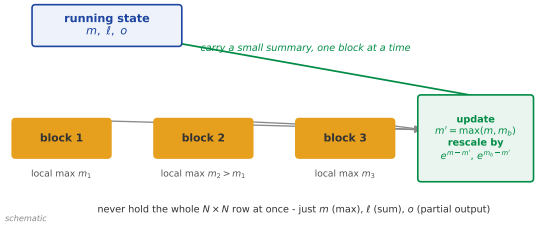
Compute the output block by block, carrying a small running summary.

Intuition: compute the output block by block

Think of a **running average or max** over a stream: you never store the whole list – you keep a small summary and update it as each value arrives.

FlashAttention does the same with attention. Split Q, K, V into **blocks** that fit in fast SRAM, and combine them while carrying a running summary (m, ℓ, o).

You **never hold the whole** $N \times N$ matrix at once – only three small vectors.



m = running max, ℓ = running normalizer (sum), o = partial output. One block at a time; the result is the same attention, computed with tiny memory.

The one hurdle: the online (running) softmax

Softmax needs the **max** and **sum** over a *whole* row – but tiling only shows us **one block at a time**. Keep running stats and **rescale** when the max shifts.

Numerically-stable softmax of a row, with $m = \max_j x_j$:

$$\mathrm{softmax}(x)_i = \frac{e^{x_i - m}}{\ell}, \qquad \ell = \sum_j e^{x_j - m}.$$

Carry state (m, ℓ, o). A new block b has local max m_b, local sum ℓ_b, and partial output $P_b V_b$:

$$m' = \max(m,\, m_b), \qquad \alpha = e^{\,m - m'}, \qquad \beta = e^{\,m_b - m'}$$

$$\ell' = \alpha\,\ell + \beta\,\ell_b, \qquad o' = \alpha\,o + \beta\,(P_b V_b), \qquad \boxed{O = o/\ell \text{ at the end}}$$

The correction factor $e^{\,m-m'}$ is the whole trick: when a later block reveals a bigger max, rescale everything so far by **one scalar**. The output is **bit-exact** softmax attention – not an approximation.

Online softmax, by hand

One attention row, scores $x = [0,\ 2 \mid 3,\ 1]$, split into two blocks. Watch the running normalizer ℓ land on **exactly** the full-row denominator.

Block 1 $= [0, 2]$: $m_1 = 2$, $\ell_1 = e^{0-2} + e^{2-2} = 0.135 + 1 = 1.135$. State: $m = 2,\ \ell = 1.135$.

Block 2 $= [3, 1]$: $m_2 = 3$, so the new max is $m' = \max(2, 3) = 3$.

$$\alpha = e^{m-m'} = e^{-1} = 0.368, \qquad \ell_{\text{blk}} = e^{3-3} + e^{1-3} = 1 + 0.135 = 1.135$$

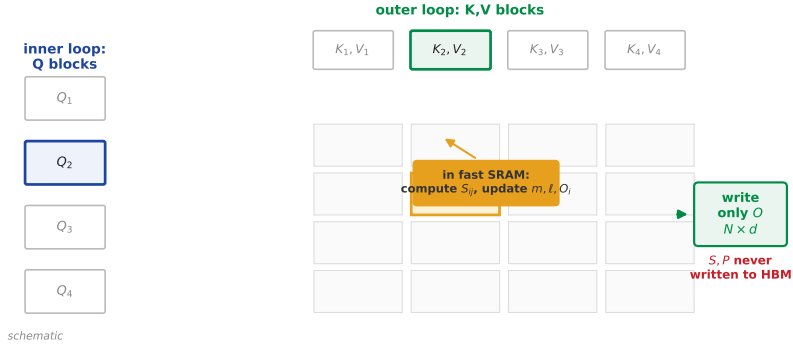
$$\ell' = \underbrace{\alpha\,\ell}_{\text{rescaled block 1}} + \ell_{\text{blk}} = 0.368(1.135) + 1.135 = \mathbf{1.553}$$

Direct check: full-row softmax denominator (max $= 3$): $e^{-3} + e^{-1} + e^{0} + e^{-2} = \mathbf{1.553}$. ✓

The single scalar $\alpha = e^{m-m'}$ retro-corrects block 1 the instant block 2 reveals a larger max. Two blocks, **exact** full-row softmax – no full row ever held at once.

The tiling algorithm (Algorithm 1)

Outer loop over K, V blocks; inner loop over Q blocks. Each tile lives in SRAM; only the final O is written back.



Load a Q_i / K_j, V_j tile into SRAM, compute the block S_{ij}, update (m, ℓ, O_i) with the rescaling, move on. **S and P are never written to HBM** – only O $(N \times d)$ is. One fused CUDA kernel does the whole thing.

Backward pass: recompute, don't store

Storing the $N \times N$ for backprop would undo everything. So don't store it.

Recompute the attention matrix instead of storing it

Backprop normally needs S and P (the $N \times N$) to form the gradients. Storing them would defeat the whole purpose.

FlashAttention instead stores **only the softmax stats** (m, ℓ) – size N – and **recomputes** the blocks of S, P on the fly in SRAM during the backward pass.

That is *more* FLOPs. But attention is memory-bound, so trading arithmetic for far fewer HBM accesses is still a **net win**.

GPT-2 medium	standard	flash
GFLOPs	66.6	**75.2**
HBM R/W (GB)	40.3	**4.4**
runtime (ms)	41.7	**7.3**

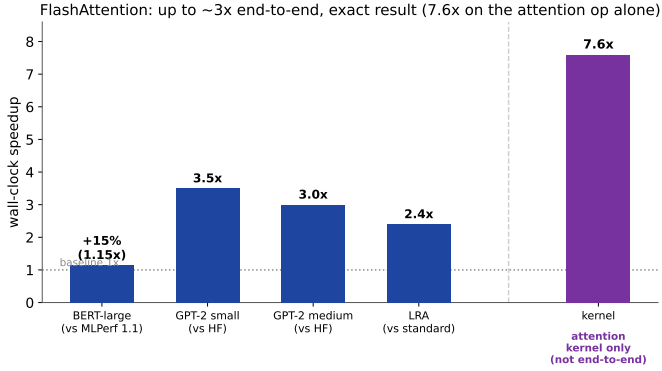
More FLOPs, ∼9x less HBM traffic, ∼5.7x faster.

Recomputation is usually a **bad** trade – here it **wins**, because we are bound by memory, not compute. (A form of selective gradient checkpointing.)

Does it work?

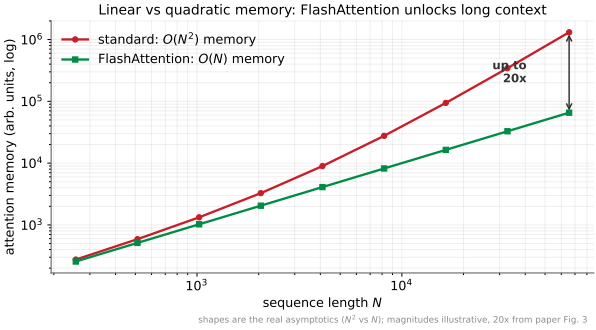
Faster training, less memory, longer context – and the same exact numbers.

Faster training, exact result



BERT-large **15%** faster than the MLPerf 1.1 record (17.4 vs 20.0 min); GPT-2 up to **3x** vs HuggingFace, ∼1.7x vs Megatron; LRA **2.4x**. The **7.6x** is the **attention kernel alone**, not end-to-end. Same perplexity: *exact* attention, not an approximation.

Linear memory unlocks long context



Standard attention's memory grows **quadratically** – the $N \times N$ matrix. Never materializing it makes FlashAttention's memory grow **linearly** in N.

- Up to **20x** less memory than exact-attention baselines.
- Memory now scales with N, not N^2.
- So you can afford sequences that used to run out of memory.

Less memory is not just cheaper – it is what makes **long-context** training possible at all.

New capability: Path-X and Path-256

Long, exact context enables the **first** Transformers to beat chance on two extreme long-range tasks (images fed one pixel at a time):

Model	Path-X (16K)	Path-256 (64K)
prior Transformers	✗	✗
FlashAttention (dense)	**61.4%**	✗
block-sparse FlashAttention	56.0%	**63.1%**

Two *distinct* results. **Dense** FlashAttention is exact and first to solve **Path-X** (16K). It cannot reach 64K – **block-sparse** FlashAttention (an *approximate* variant that skips zero blocks) is what solves **Path-256** (64K).

This is the "long context" thread the report decks (Llama3, Qwen3) build on.

Recap

- **Memory-bound, not compute-bound.** Attention's wall-clock is dominated by HBM traffic, not FLOPs – so the win is in *memory access*, not arithmetic.
- **Tiling + online softmax.** Compute the output block by block, carry (m, ℓ, o), and rescale by $e^{m-m'}$ on a new max – so the $N \times N$ matrix is **never materialized**. Exact, not approximate.
- **Recompute in backward.** Store only (m, ℓ); recompute S, P on-chip. More FLOPs, far less IO, still a net speedup.
- **The payoff.** Up to $\sim$**3x** end-to-end (**7.6x** on the attention op alone), up to **20x** less memory, linear in N.
- **You already use this.** `torch.nn.functional.scaled_dot_product_attention` dispatches to FlashAttention kernels under the hood.

Next: long context everywhere – *Llama3 (07)* at 128k, *Qwen3 (08)* at 32k. The efficiency mindset continues in *DeepSeek-V3 (17)*, where **MLA** shrinks the **KV cache** – the other half of the attention-memory story.