FlashAttention

Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao, Daniel Y.Fu, Stefano Ermon, Rudra, Christopher Ré · Stanford & SUNY Buffalo · 2022

Transformers struggle with long sequences

$O(N^2)$

time complexity

Attention scales quadratically with sequence length N

To compute QK^T ,every token's query vector must be compared against every other token's key vector. For a sequence of N tokens, that's $N \times N = N^2$ dot products. Then multiplying the resulting $N \times N$ matrix by V is another N^2 operation. So total FLOPs scale as $O(N^2 d)$.

$O(N^2)$

memory cost

The N×N attention matrix is materialized in memory

The intermediate matrix $S = QK^T \in \mathbb{R}^{N \times N}$ must be materialized and stored in HBM. That matrix has N^2 entries. For example, at sequence length $N = 1024$ with 16 heads and batch size 64 (like GPT-2 in the paper), the attention matrix alone takes **40.3 GB** of HBM reads/writes.

Slow

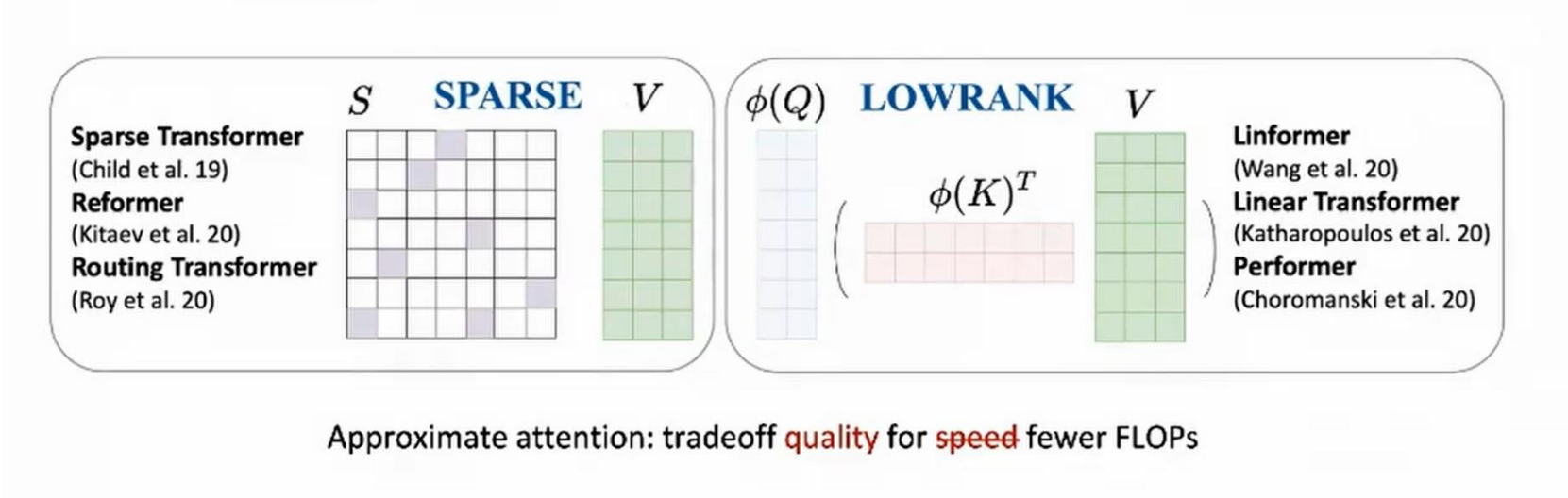
in wall-clock

Most operations are memory-bound, not compute-bound

Self-attention is the heart of every Transformer — and it is exactly the part that breaks at long context.

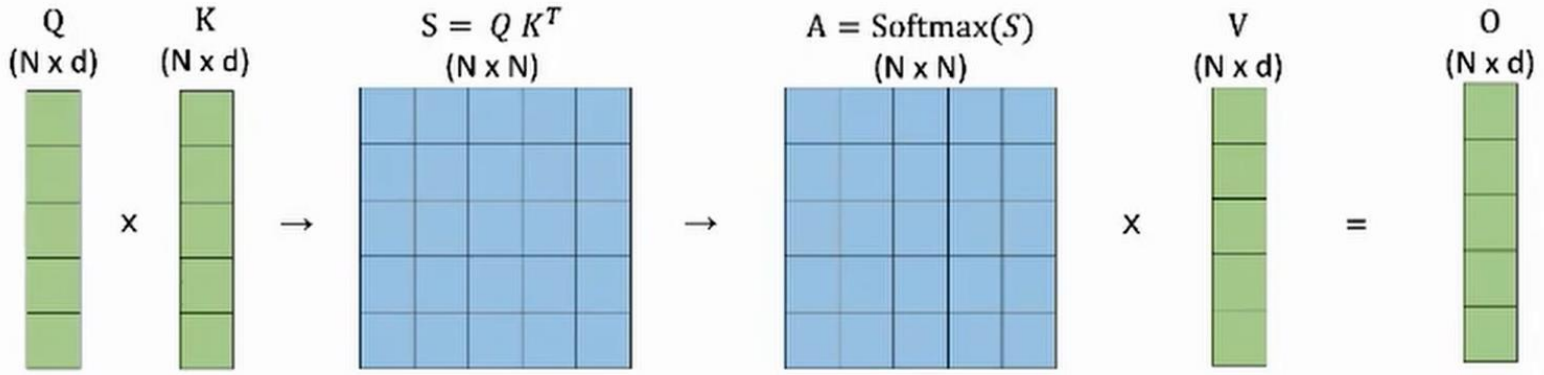
The field's response: approximate attention

Many methods tried to reduce FLOPs by approximating the N×N matrix:



But many of them don't actually run faster in wall-clock time — and so haven't been adopted.
Because you're still doing multiple passes over HBM, just with a smaller matrix.
And at the end its still approximation

Background: Attention Mechanism



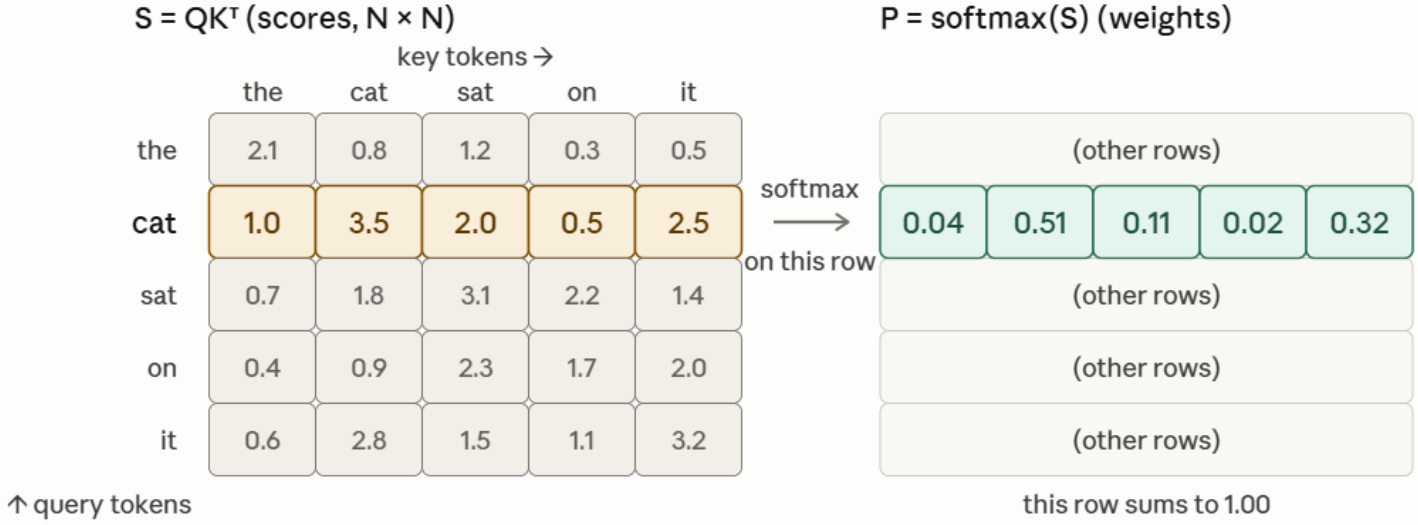
Typical sequence length N: 1K – 8K
Head dimension d: 64 – 128

$$\text{Softmax}([s_1, \cdots, s_N]) = \left[\frac{e^{s_1}}{\sum_i e^{s_i}}, \cdots, \frac{e^{s_N}}{\sum_i e^{s_i}}\right]$$

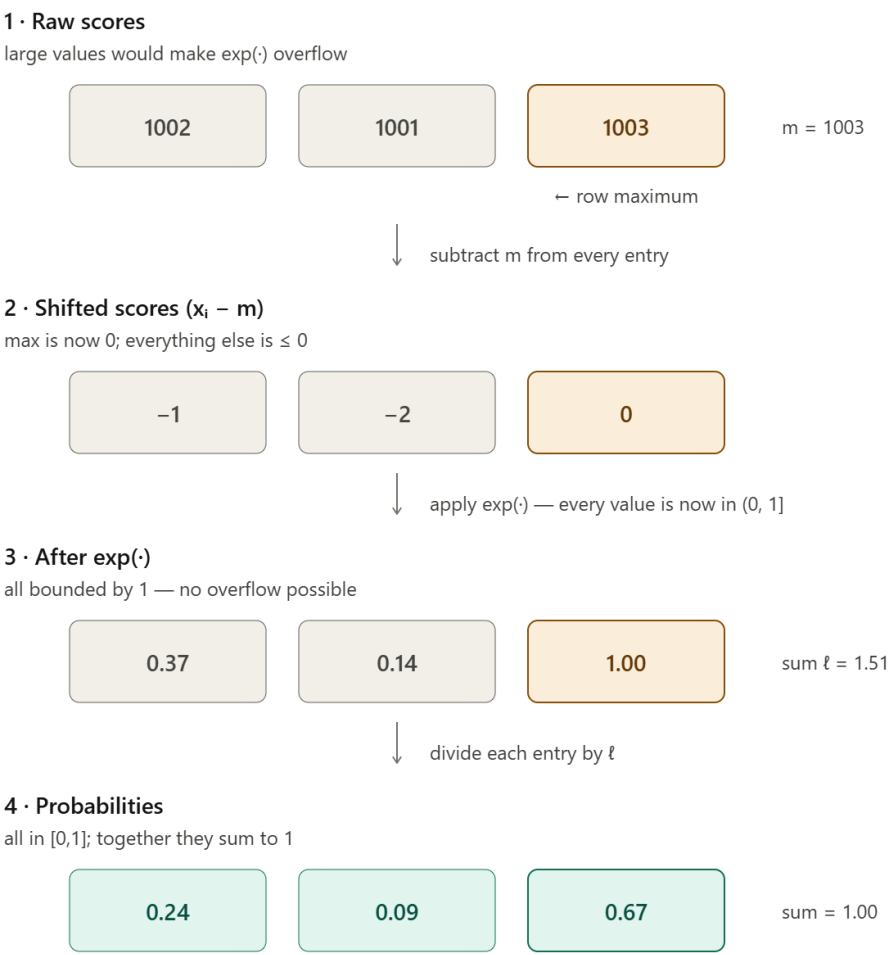
$$O = \text{Softmax}(QK^T)V$$

SOFTMAX

Each row of S asks: "how much should this query attend to each key?"
softmax is applied to that row independently — N rows, N softmaxes



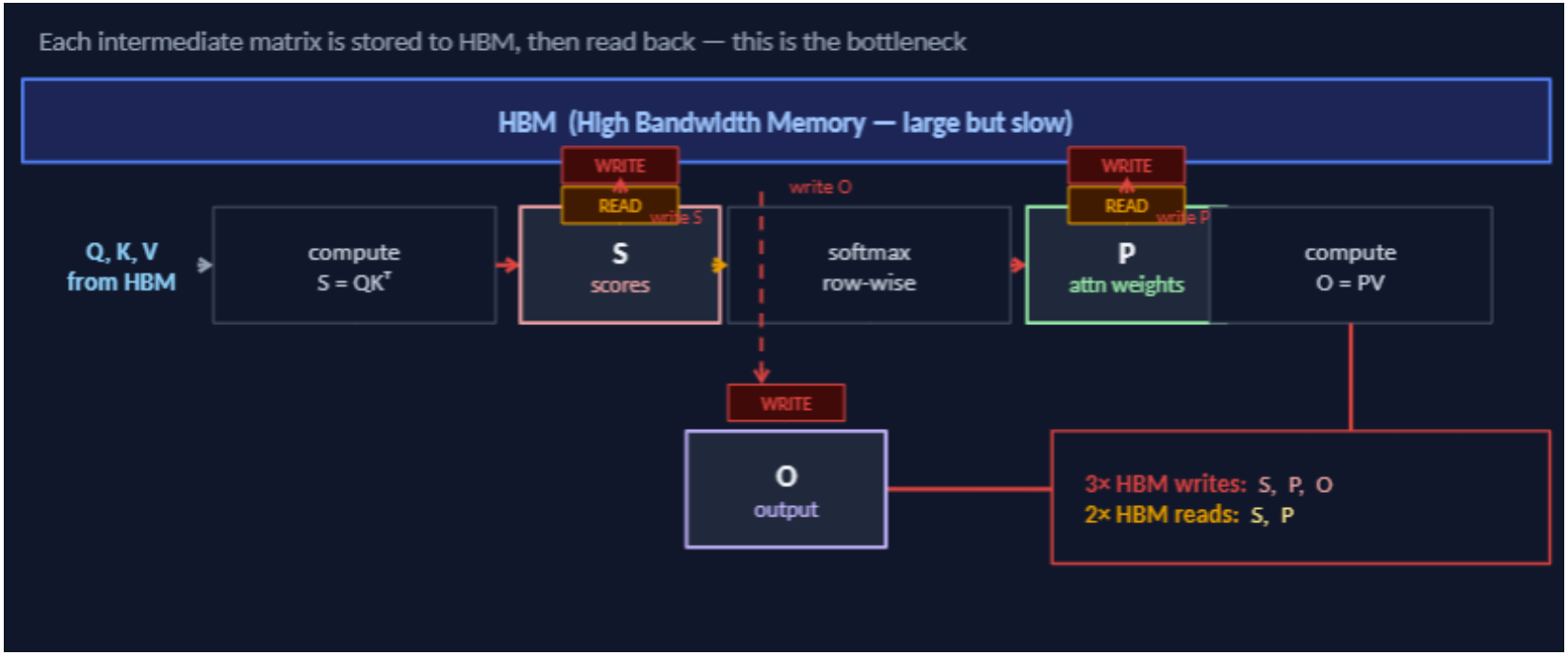
- **Softmax takes a query token's raw dot-product scores with every key.**
- **Squashes them into probabilities that sum to 1.**
- **Where each probability tells you how well that query matches with that key.**



How is softmax calculated??

Numerically stable softmax

$$\text{softmax}(x_i) = \exp(x_i - m) / \Sigma_j \exp(x_j - m), \text{ where } m = \max(x)$$



Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Why are we even writing the intermediate results S and P into the HBM memory?

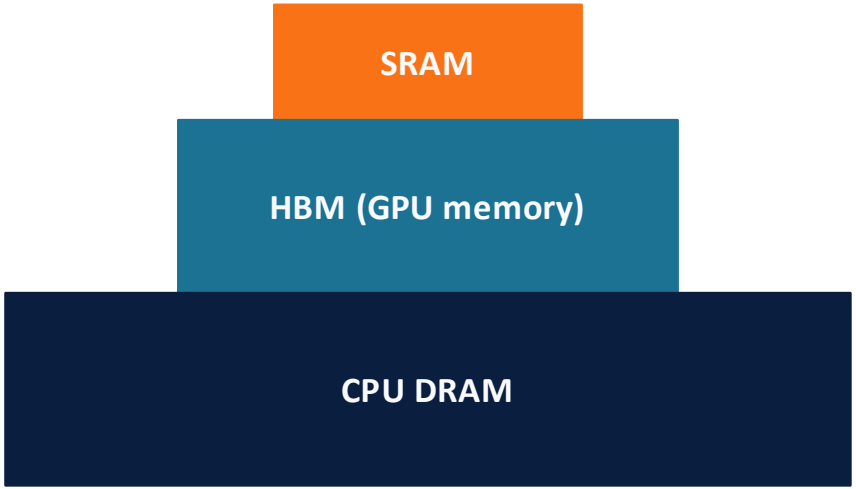
It's not the FLOPs.
It's the memory traffic.

Past methods optimized the wrong thing. They cut arithmetic operations but kept reading and writing the giant N×N matrix to slow GPU memory. The real cost is that data movement, not the math.

Make the algorithm IO-aware — count reads and writes between fast and slow GPU memory.

A GPU has a memory hierarchy

FASTER



A100 GPU specs

SRAM
~20 MB · 19 TB/s

HBM
40 GB · 1.5 TB/s

DRAM
>1 TB · 12.8 GB/s

SLOWER

SRAM is ~13× faster than HBM, but ~2,000× smaller. Working on SRAM is great if you can fit there.

Most attention operations are memory-bound

Compute-bound

- Lots of math per byte read
- GPU's compute units are the bottleneck
- Examples: large matrix multiplies

Modern GPUs have enormous compute power — these run fast.

Memory-bound

- Few operations per byte read
- GPU sits waiting on memory
- Examples: softmax, dropout, masking, norm

Loading and storing data dominates the runtime.

How will you reduce these trips to main memory(or HBM)?

FlashAttention: keep the work inside SRAM

The whole approach in one sentence:

Load small blocks of Q, K, V into fast SRAM, do all the math there, and write only the final output back — never store the intermediate N×N matrix like S= QK^T and P=softmax(S) in HBM.

01

Tiling

Split inputs into blocks small enough to fit in SRAM. Process one tile at a time and combine results incrementally.

02

Recomputation

Don't store the attention matrix ***P=softmax(S)*** for the backward pass. Save tiny softmax statistics and regenerate the matrix on-the-fly.

Tiling sounds easy — softmax makes it hard

Why you can't just split the attention matrix into independent blocks:

Softmax of a row x = [x_1, x_2, … , x_N]:

$$\text{softmax}(x_i) = \exp(x_i - m) / \Sigma_j \exp(x_j - m) \quad \text{where } m = \max(x)$$

The denominator sums over ALL columns in a row. And we need the max of ALL columns first (for numerical stability). So a single output entry depends on the entire row — we cannot compute it from one block alone.

Online softmax (Milakov & Gimelshein, 2018)

Carry two extra numbers per row: the running max m and the running denominator ℓ. After each new block, update m and ℓ in a way that's algebraically equivalent to having seen all data. The final answer is **exact — not an approximation.**

Softmax of Two Concatenated Vectors

Block 1

$\vec{x}^{\{1\}}$

$m_1 = \max(\vec{x}^{\{1\}})$

$l_1 = \sum_j e^{x_j - m_1}$

$\vec{f}_1 = e^{x^{\{1\}} - m_1}$

Block 2

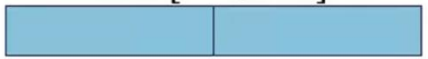
$\vec{x}^{\{2\}}$

$m_2 = \max(\vec{x}^{\{2\}})$

$l_2 = \sum_j e^{x_j - m_2}$

$\vec{f}_2 = e^{x^{\{2\}} - m_2}$

$\vec{x} = [\vec{x}^{\{1\}}, \vec{x}^{\{2\}}]$



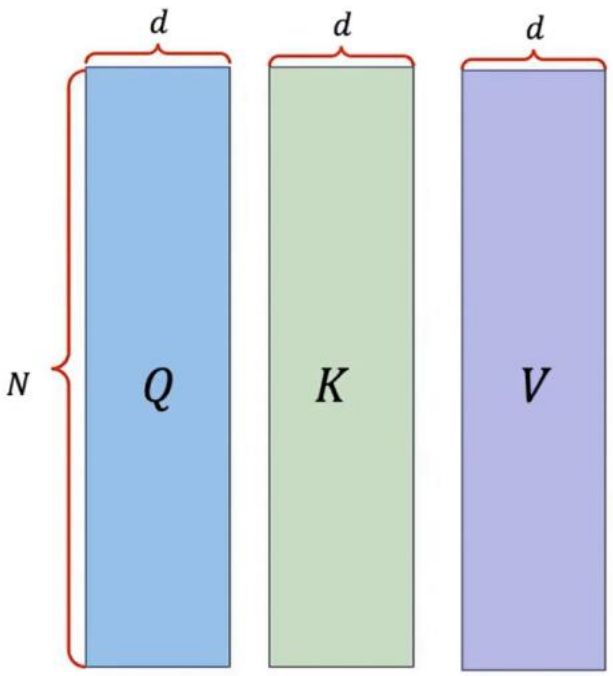
Overall max $m(x) = \max(m_1, m_2)$

Adjusted $\vec{f}_1$ and $\vec{f}_2$ $\vec{f}\,(x) = [e^{m_1 - m(x)} f_1, e^{m_2 - m(x)} f_2]$

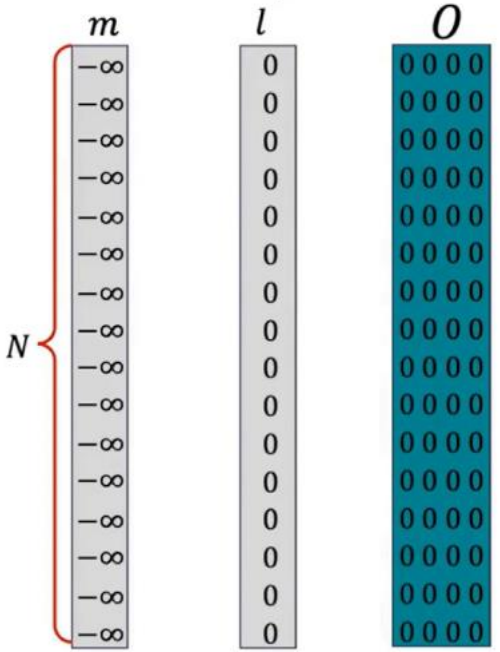
Adjusted normalization factor $l(x) = e^{m_1 - m(x)} l_1 + e^{m_2 - m(x)} l_2$

$$\text{Softmax}(\vec{x}) = \frac{\vec{f}(x)}{l(x)}$$

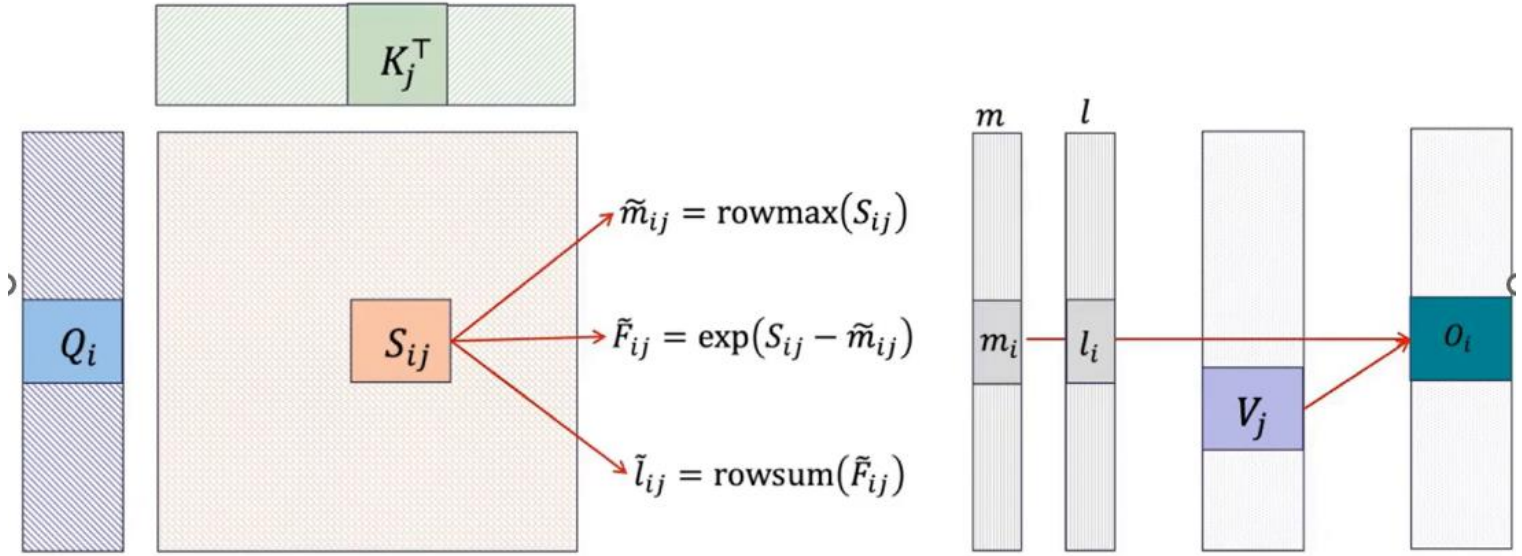
FlashAttention – Input and Initialization



Initialize intermediate vectors and the output:



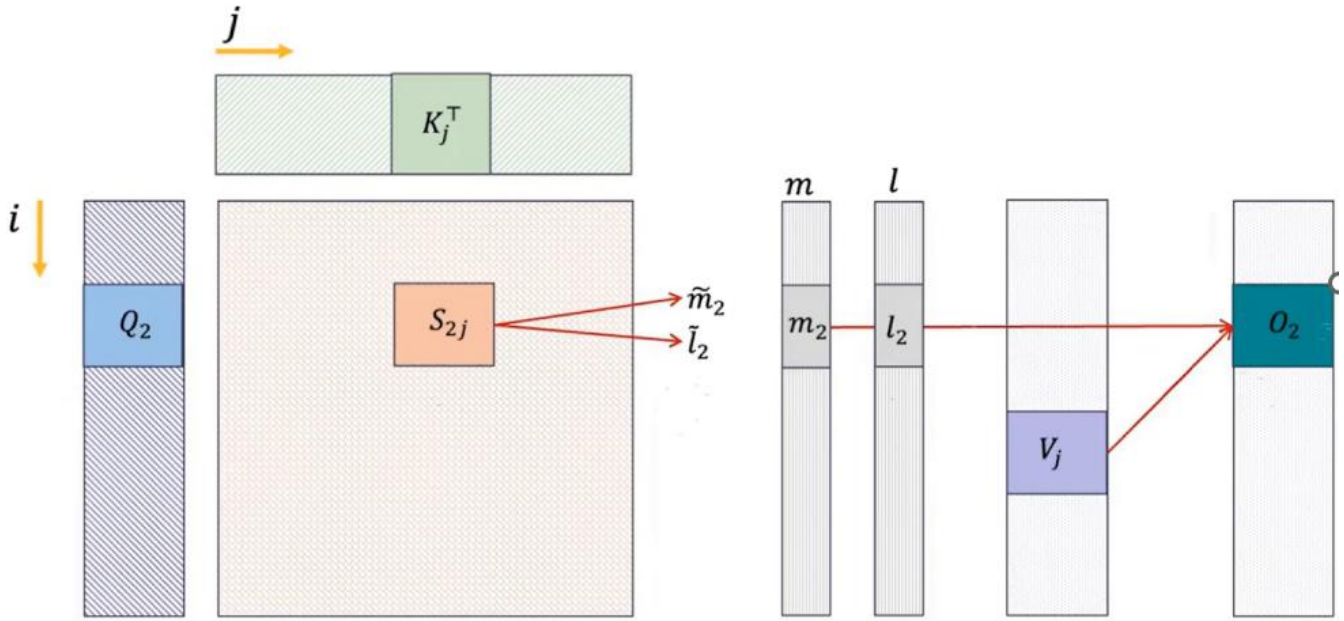
FlashAttention – Partial Updates



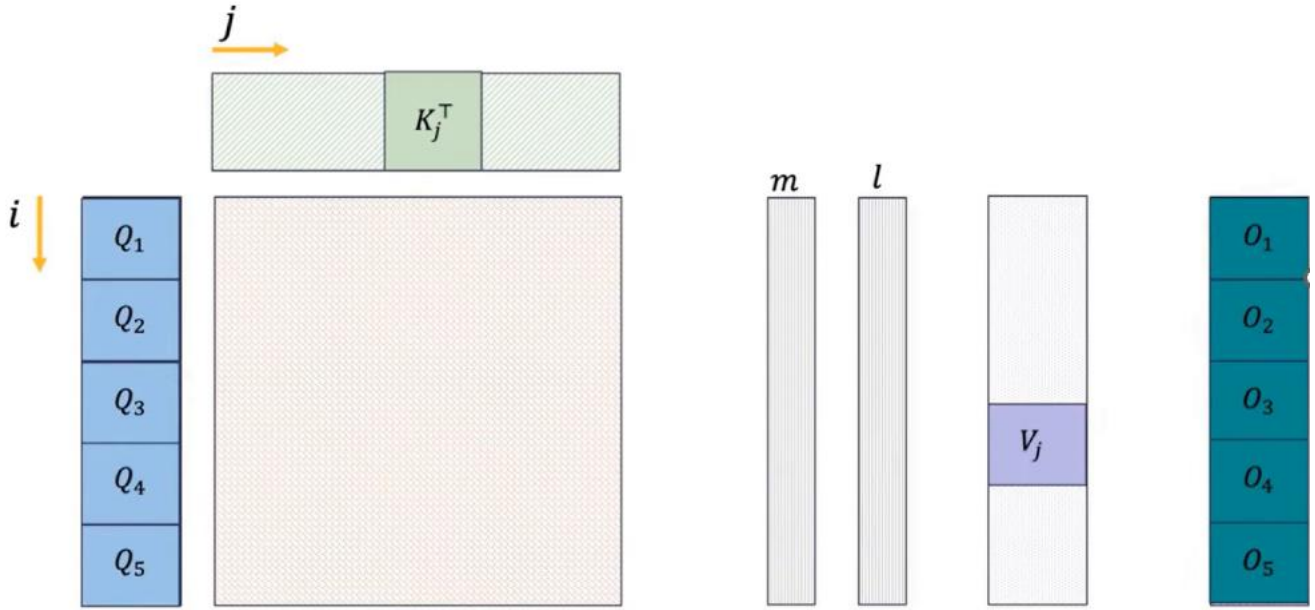
Partial update to block matrix O_i

$$O_i^{\text{new}} = \text{diag}(l_i^{\text{new}})^{-1}\left(\text{diag}(l_i)e^{m_i - m_i^{\text{new}}}O_i + e^{\tilde{m}_i - m_i^{\text{new}}}\tilde{F}_{ij}V_j\right)$$

FlashAttention – Partial Updates



FlashAttention – Partial Updates



Algorithm 1 — what it looks like in pseudocode

```
Divide Q into row-blocks Q₁…Q_Tr
Divide K, V into col-blocks K₁…K_Tc
Initialize O = 0,  ℓ = 0,  m = −∞

for j = 1 … Tc:                      # outer loop
    load K_j, V_j into SRAM
    for i = 1 … Tr:                  # inner loop
        load Q_i, O_i, ℓ_i, m_i
        S_ij = Q_i · K_j^T           # on-chip
        update m_i, ℓ_i  (online softmax)
        update O_i with new contribution
        write O_i, ℓ_i, m_i back to HBM

return O
```

What to notice

Two nested loops over blocks — each K, V loaded once

Inner loop revisits Q blocks Tc times

All math (matmul, softmax, output) fused into one CUDA kernel

Result is exact — bit-for-bit equivalent to standard attention

Recomputation: a clever space-time trade

The backward-pass dilemma

- Backprop through softmax needs the attention matrix P.
- Standard practice: store P (size N^2) during forward pass.
- But P is exactly what we're trying NOT to materialize.

Flash's trick

- Save only the softmax statistics (m, ℓ) — size N, not N^2.
- In the backward pass, reload Q, K, V tiles into SRAM.
- Recompute the relevant blocks of P on-chip as needed.
- Net effect: more arithmetic, but far fewer HBM reads.

Recomputing P in the backward pass

The backward pass needs $P = \text{softmax}(QK^T/\sqrt{d})$ — but it was never saved to HBM

Recompute a tile instead

$S_{ij} = q_i k_j^T / \sqrt{d}$ → $P_{ij} = \exp(S_{ij} - m_i) / \ell_i$

m_i and ℓ_i were saved during the forward pass (O(N) total)

Without m_i, ℓ_i — 3 passes over K

Pass 1
scan all K → find global max m_i

↓

Pass 2
scan all K → compute $\ell_i = \Sigma \exp(S_{ij} - m_i)$

↓

Pass 3
$P_{ij} = \exp(S_{ij} - m_i) / \ell_i$

vs

With m_i, ℓ_i — 1 pass over K

m_i and ℓ_i already known
passes 1 and 2 are already done ✓

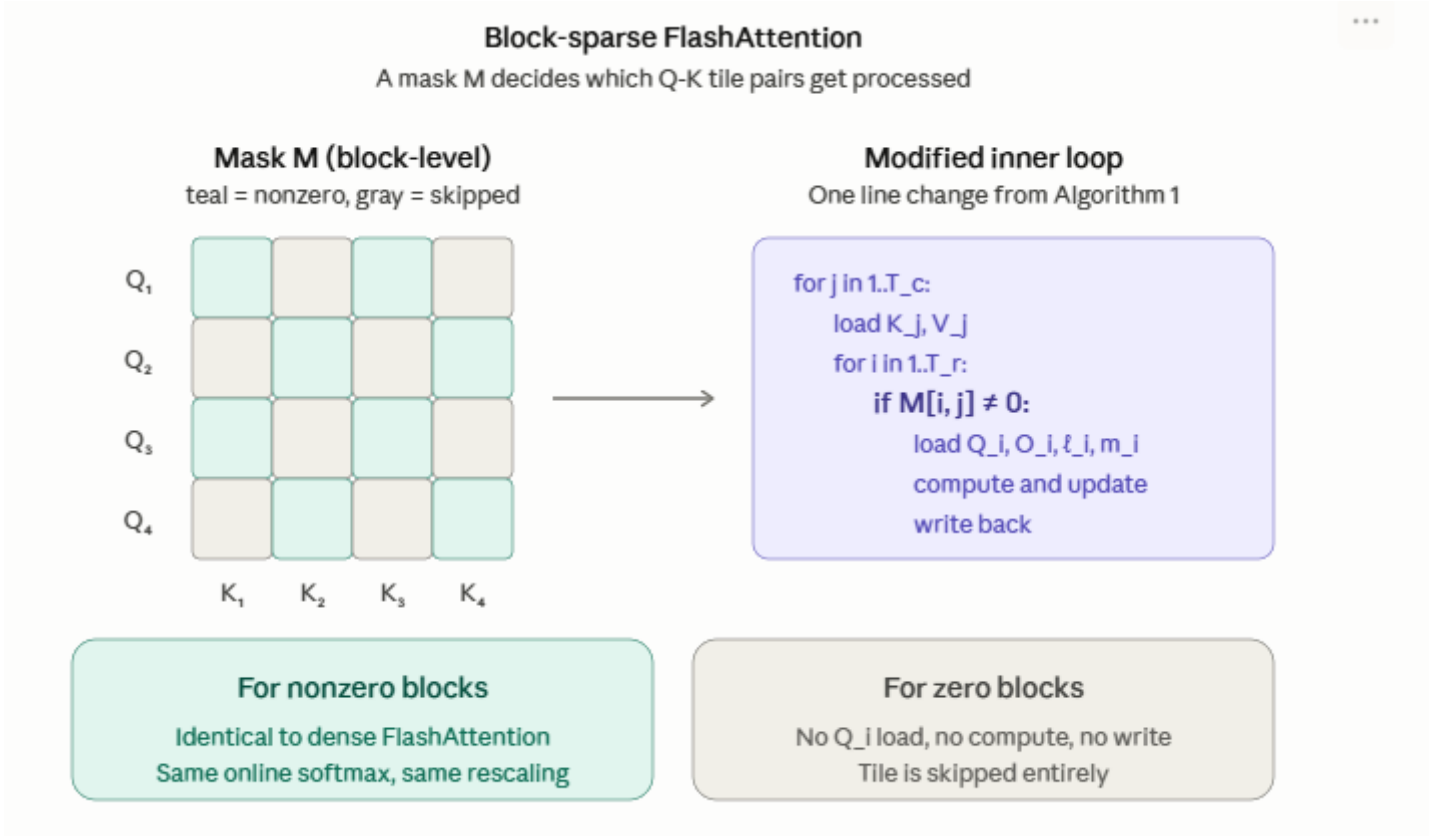
↓

Pass 1 (only pass)
$P_{ij} = \exp(S_{ij} - m_i) / \ell_i$

The exp is just a pointwise operation once m_i and ℓ_i are known.

m_i and ℓ_i are sufficient statistics for softmax — 2 scalars per row replace 2 full scans.

Block-sparse Flash Attention



How much memory traffic do we actually save?

Standard attention

$$\Theta(N^2 + N \cdot d)$$

Quadratic in N — dominated by reading/writing the N×N matrix.

FlashAttention

$$\Theta(N^2 \cdot d^2 \,/\, M)$$

M = SRAM size. With typical d=64 and M≈100KB, $d^2/M \ll 1$.

Block sparse FlashAttention

$$\Theta(N \cdot d + N^2 \cdot d^2 \cdot s \,/\, M)$$

M = SRAM size. With typical d=64 and M≈100KB, $d^2/M \ll 1$.

In practice: up to 9× fewer HBM accesses

And the authors prove a matching lower bound: no exact attention algorithm can do asymptotically better in HBM accesses across all SRAM sizes. FlashAttention is essentially optimal.

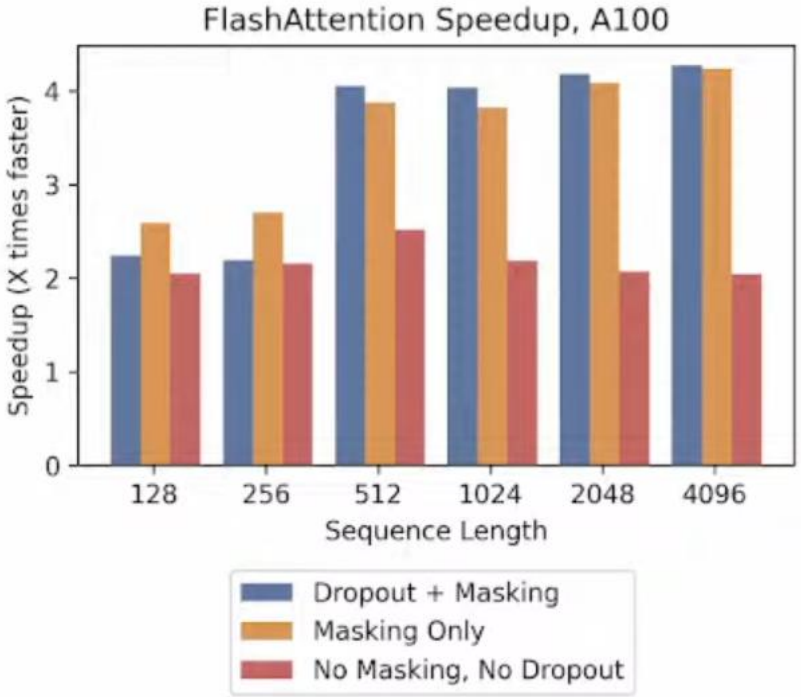
Why does this recomputation trick translate into longer sequences?

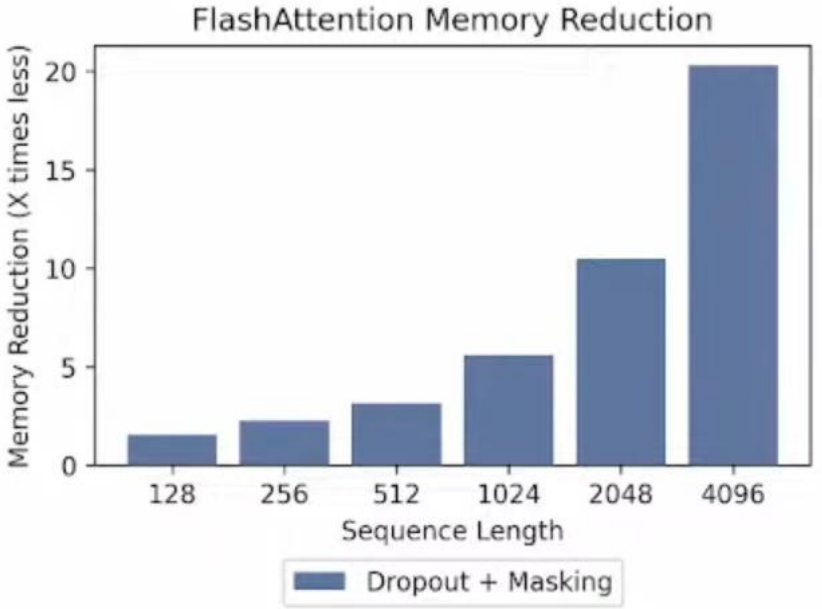
EMPIRICAL RESULTS

SPEEDUP:
Faster end-to-end Training of Transformers

Memory Savings:
Longer sequences able to compute
Higher quality transformers even at long sequences

FlashAttention: 2-4x speedup, 10-20x memory reduction





2-4x speedup — with no approximation!

10-20x memory reduction — memory linear in sequence length

Faster Training: MLPerf Record for Training BERT-large

MLPerf: (highly optimized) standard benchmark for training speed

Time to hit an accuracy of 72.0% on MLM from a fixed checkpoint, averaged across 10 runs on 8xA100 GPUs

BERT Implementation	Training time (minutes)
Huggingface [91]	55.6 ± 3.9
Nvidia MLPerf 1.1 [63]	20.0 ± 1.5
FLASHATTENTION	**17.4** ± 1.4

FlashAttention outperforms the previous MLPerf record by 15%
(and 3.2x faster than Huggingface BERT)

Long-range Arena. We compare vanilla Transformer (with either standard implementation or FlashAttention) on the long-range arena (LRA [80]) benchmark. We measure accuracy, throughput, and training time of all models. Each task has a different sequence length varying between 1024 and 4096. We follow the implementation and experimental setting in Tay et al. [80]and Xiong et al. [90].[3] Table 3 shows that FlashAttention achieves up 2.4× speed-up compared to standard attention. Block-sparse FlashAttention is faster than all of the approximate attention methods that we have tested.

Table 3: The performance of standard attention, FlashAttention, block-sparse FlashAttention, and approximate attention baselines on the Long-Range-Arena benchmarks.

Models	ListOps	Text	Retrieval	Image	Pathfinder	Avg	Speedup
Transformer	36.0	63.6	81.6	42.3	72.7	59.3	-
FlashAttention	37.6	63.9	81.4	43.5	72.7	59.8	2.4×
Block-sparse FlashAttention	37.0	63.0	81.3	43.6	73.3	59.6	**2.8**×
Linformer [84]	35.6	55.9	77.7	37.8	67.6	54.9	2.5×
Linear Attention [50]	38.8	63.2	80.7	42.6	72.5	59.6	2.3×
Performer [12]	36.8	63.6	82.2	42.1	69.9	58.9	1.8×
Local Attention [80]	36.1	60.2	76.7	40.6	66.6	56.0	1.7×
Reformer [51]	36.5	63.8	78.5	39.6	69.4	57.6	1.3×
Smyrf [19]	36.1	64.1	79.0	39.6	70.5	57.9	1.7×

Better models with longer sequences

Table 4: GPT-2 small with FlashAttention, with 4× larger context length compared to Megatron-LM, is still 30% faster while achieving 0.7 better perplexity. Training time on 8×A100 GPUs is reported.

Model implementations	Context length	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Megatron-LM	1k	18.2	4.7 days (1.0×)
GPT-2 small - FlashAttention	1k	18.2	**2.7 days (1.7×)**
GPT-2 small - FlashAttention	2k	17.6	3.0 days (1.6×)
GPT-2 small - FlashAttention	4k	**17.5**	3.6 days (1.3×)

The Pathfinder task

Given a 128×128 (or 256×256) black-and-white image fed pixel-by-pixel, classify whether two highlighted points are connected by a path.

It's a true test of long-range reasoning. Until now, every Transformer either ran out of memory or got chance accuracy.

Table 6: We report the first Transformer model that can achieve non-random performance on Path-X and Path-256.

Model	Path-X	Path-256
Transformer	✗	✗
Linformer [84]	✗	✗
Linear Attention [50]	✗	✗
Performer [12]	✗	✗
Local Attention [80]	✗	✗
Reformer [51]	✗	✗
SMYRF [19]	✗	✗
FlashAttention	**61.4**	✗
Block-sparse FlashAttention	56.0	**63.1**

Benchmarking

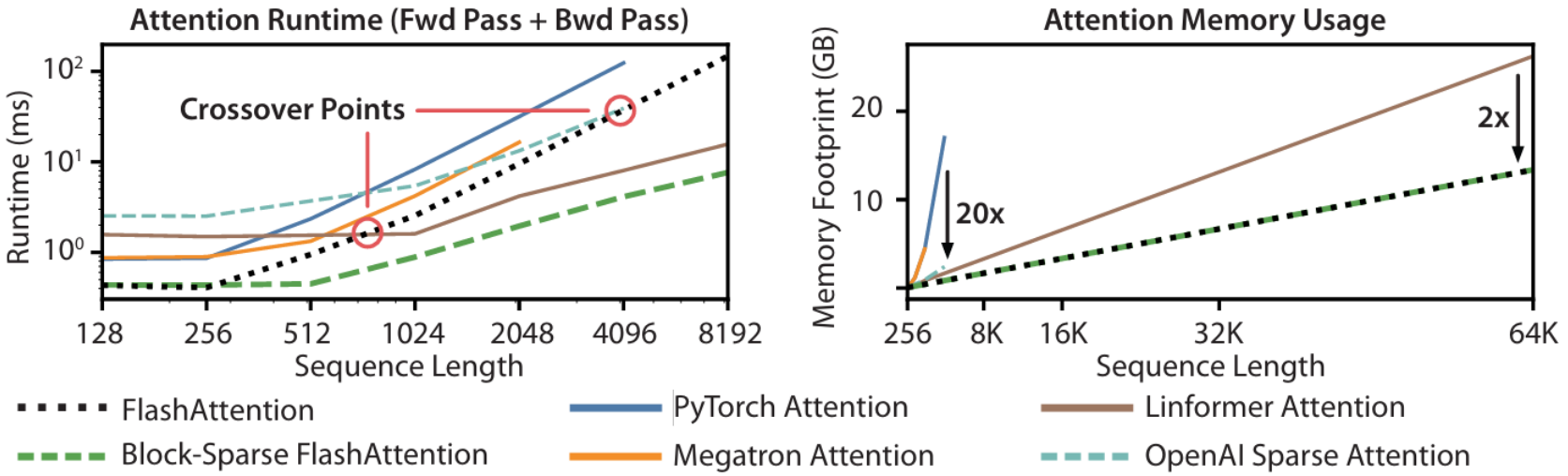


Figure 3: **Left:** runtime of forward pass + backward pass. **Right:** attention memory usage.

Takeaways

Optimize what actually matters

On modern GPUs, memory bandwidth — not arithmetic — is usually the bottleneck. The fastest algorithm is the one that moves the least data.

Tiling + online softmax = exact, not approximate

FlashAttention is mathematically identical to standard attention. There's no quality tradeoff.

Recomputation can be a win

Doing more FLOPs to avoid HBM writes is often the right call — counter to the usual FLOP-counting intuition.

Better systems unlock new capabilities

Longer context isn't just faster — it lets Transformers solve problems they couldn't before, like Path-X and Path-256.

FlashAttention-2

Faster Attention with Better Parallelism and Work Partitioning
· Tri Dao, 2023

Where FlashAttention-1 Left Us

FlashAttention-1 won

✓ Linear memory (not $O(N^2)$)
✓ less HBM traffic
✓ 2–4× wall-clock speedup
✓ No approximation
✓ Drop-in replacement — no model architecture changes

Issues still needed to be solved which flash attention -2 addresses

✓ Flash Attention still only reaches 25–40% of theoretical max FLOPs/s

While Optimized GEMM can reach upto 80–90% of max device throughput

Why does improving TFLOPS/s matter?

GPU Architecture & Execution Model

Memory Hierarchy

HBM — High Bandwidth Memory

40–80 GB 1.5–2.0 TB/s

slow ← bandwidth → fast

SRAM — On-chip Shared Memory

192 KB per SM × 108 SMs ~19 TB/s

10× faster than HBM

Registers + Tensor Cores

FP16/BF16 matmul 312 TFLOPs/s

non-matmul: only 19.5 TFLOPs/s → 16× gap

Execution Model

Streaming Multiprocessor (SM)

108 SMs on A100

Thread Block

scheduled onto one SM

Shared Memory (SRAM) — warps communicate here

Warp 0

32 threads

lockstep

Warp 1

32 threads

lockstep

Warp 2–N

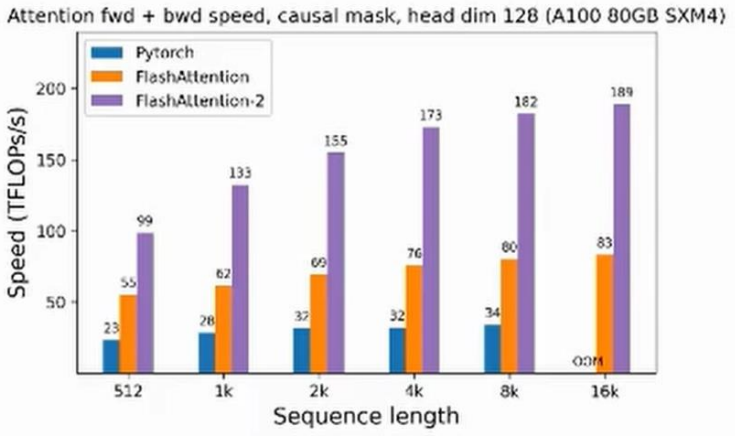
32 threads

lockstep

HBM → load inputs → **Registers / SRAM** → compute → write outputs → **HBM**

Three key ideas

Key ideas:
- Reduce non-matmul FLOPs
- Parallelize over seqlen dimension to improve occupancy
- Better work partitioning between warps to reduce communication



Fix 1 — Reduce Non-MatMul FLOPs

What Flash attention 1 does

$$O^{(2)} = \text{diag}\left(\frac{l^{(1)}}{l^{(2)}}\right)^{-1} O^{(1)} + \text{diag}(l^{(2)})^{-1} e^{S^{(2)} - m^{(2)}} V^{(2)}$$

This introduces:

- extra divisions
- extra scaling ops
- extra memory accesses

And stores both l (N) and m (N) vectors in HBM

FlashAttention-2 Optimization

FlashAttention-2 reduces non-matmul FLOPs by:

1. Keeping Output Unnormalized Until the End

Instead of repeatedly dividing/scaling every iteration:

$$\tilde{O}^{(2)} = e^{m^{(1)} - m^{(2)}} \tilde{O}^{(1)} + e^{S^{(2)} - m^{(2)}} V^{(2)}$$

Only final normalization is done once:

$$O = \text{diag}(l^{(final)})^{-1} \tilde{O}$$

Benefit

- Fewer divisions
- Fewer elementwise ops
- More time spent in GEMMs

Algorithm 1 FlashAttention-2 forward pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, block sizes B_c, B_r.

1: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

2: Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into T_r blocks $L_i, \dots, L_{T_r}$ of size B_r each.

3: **for** $1 \le i \le T_r$ **do**

4: Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.

5: On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}$, $\ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}$, $m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.

6: **for** $1 \le j \le T_c$ **do**

7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

8: On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

9: On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \mathrm{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{j-1} - m_i^{(j)}} \ell_i^{(j-1)} + \mathrm{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.

10: On chip, compute $\mathbf{O}_i^{(j)} = \mathrm{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})^{-1} \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.

11: **end for**

12: On chip, compute $\mathbf{O}_i = \mathrm{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$.

13: On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.

14: Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.

15: Write L_i to HBM as the i-th block of L.

16: **end for**

17: Return the output $\mathbf{O}$ and the logsumexp L.

2. Store Only LogSumExp

FA-1 stores:

- row max `m`
- row sum `l`

FA-2 stores only:

$$L = m + \log(l)$$

Benefit

- Less memory traffic
- Fewer bookkeeping operations

Algorithm 2 FlashAttention-2 Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ in HBM, vector $L \in \mathbb{R}^N$ in HBM, block sizes B_c, B_r.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_i, \dots, \mathbf{dO}_{T_r}$ of size $B_r \times d$ each, and divide L into T_r blocks $L_i, \dots, L_{T_r}$ of size B_r each.

Initialize $\mathbf{dQ} = (0)_{N \times d}$ in HBM and divide it into T_r blocks $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Divide $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$ in to T_c blocks $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$ and $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$, of size $B_c \times d$ each.

Compute $D = \mathrm{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$ (pointwise multiply), write D to HBM and divide it into T_r blocks $D_1, \dots, D_{T_r}$ of size B_r each.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  Initialize $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.

    On chip, compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$.

    Load $\mathbf{dQ}_i$ from HBM to SRAM, then on chip, update $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$, and write back to HBM.

    On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_i^{(j)\top} \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.

  end for

  Write $\mathbf{dK}_j, \mathbf{dV}_j$ to HBM.

end for

Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

Fix 2 — Parallelize Over Sequence Length(along Tr)

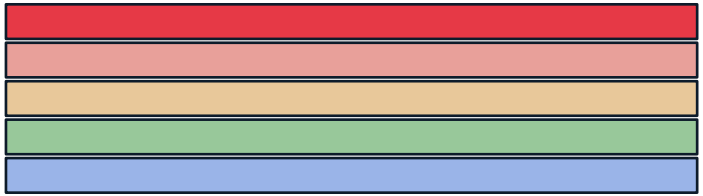
FlashAttention-1

Parallel over:
- batch size
- number of heads

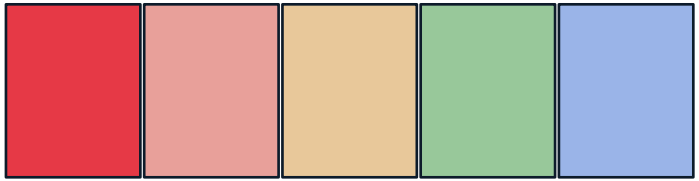
FlashAttention-2

Parallel over:
- batch size
- number of heads
- sequence length ← NEW

Forward pass: each thread block handles a block of ROWS



Backward pass: each thread block handles a block of COLUMNS → atomic adds sync dQ



Why Q is inner loop in Backward pass and outer loop in forward pass?

PARALLELISING THE ALGORITHM

1) Forward pass

- **Outer loop (over row blocks i)** is **embarrassingly parallel**.
- We assign different row blocks to different thread blocks (workers). No communication needed.
- Also parallelize over batch and heads as in FlashAttention.
- Higher parallelism over sequence length → higher occupancy, especially when batch size or #heads is small.
- Loop order swapped vs. FlashAttention-1:
  outer loop over row blocks i, inner **loop over** column blocks j.
- First suggested & implemented in Triton by Phil Tillet. 3

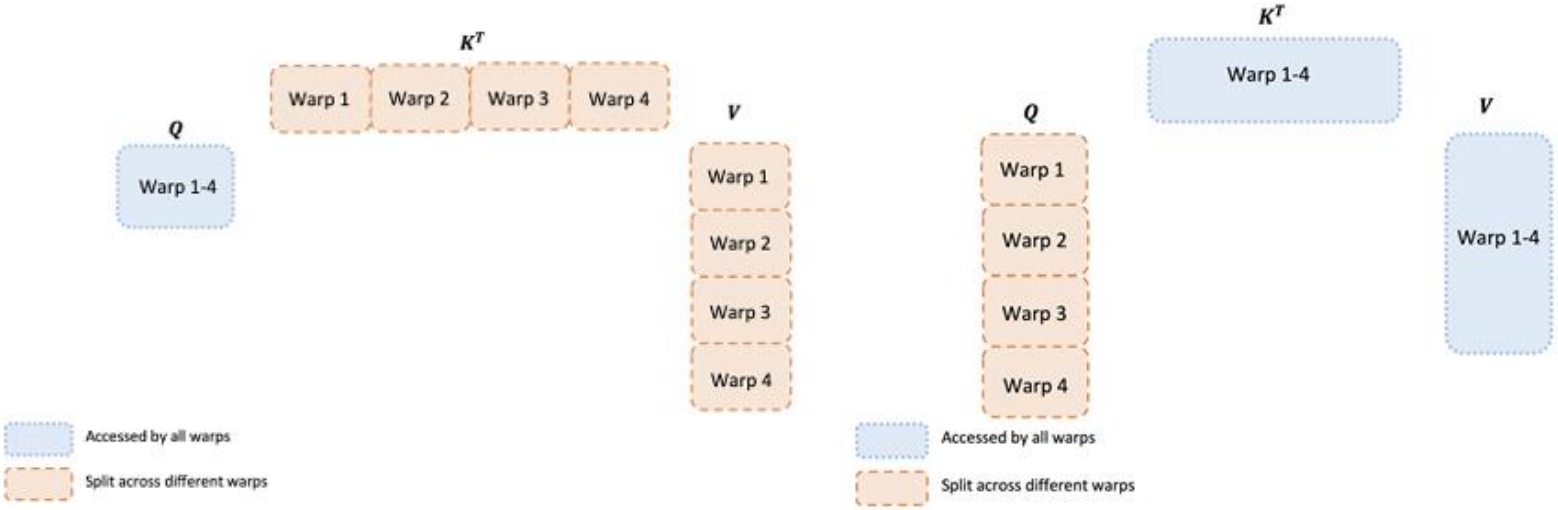
2) Backward pass

- Different column blocks j share work **only when updating dQ**.
- In Algorithm 2:

$$dQ_i \leftarrow dQ_i + dS^{(j)} K_j$$

- Thus we parallelize over **column blocks** (sequence length).
- Schedule 1 thread block per column block j.
- Use **atomic adds** to safely accumulate into dQ from multiple thread blocks.

Fix 3 — Work Partitioning between warps



(a) FlashAttention (b) FlashAttention-2

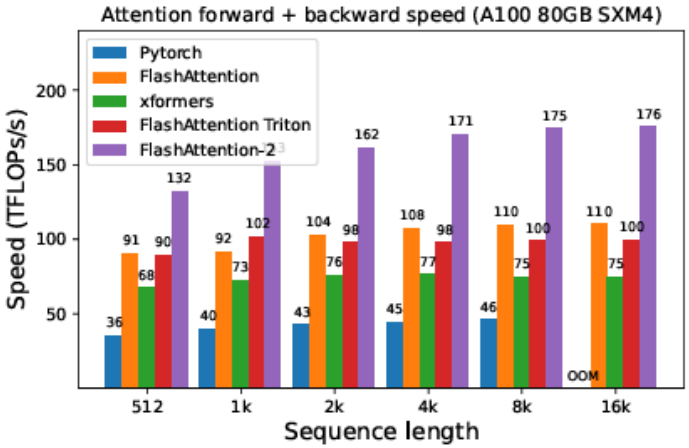
Figure 3: Work partitioning between different warps in the forward pass

How does splitting Q instead of K,V speed up?

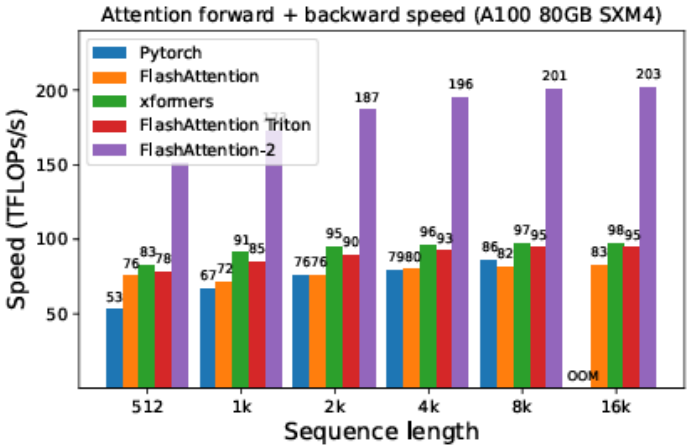
FlashAttention-2 Warp Partitioning

	Forward Pass	Backward Pass
FlashAttention-1	Uses **split-K**	Uses **split-K**
Problem	Warps compute partial outputs that must be reduced	Gradients have cross-dependencies (`dQ,dK,dV`)
Overhead	Shared-memory writes + synchronization + reduction	Even more synchronization and shared-memory traffic
FlashAttention-2 Fix	Split **Q rows across warps** instead of K/V	Similar warp partitioning to avoid split-K
Result	Each warp computes independent output rows	Reduces inter-warp communication
Benefit	No output reduction needed	Fewer shared-memory reads/writes
Outcome	Higher Tensor Core utilization	Faster backward pass

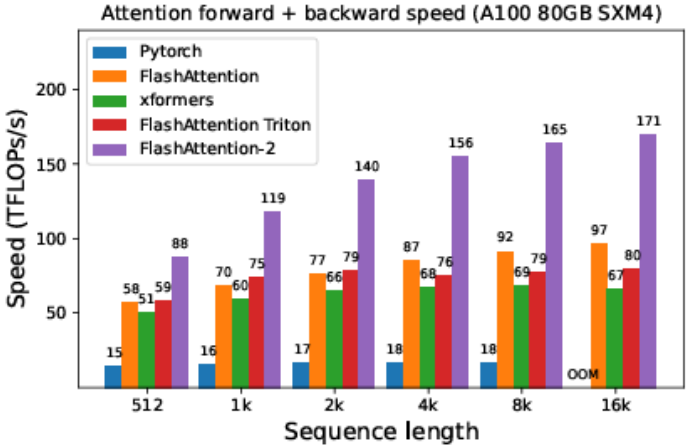
EMPIRICAL RESULTS



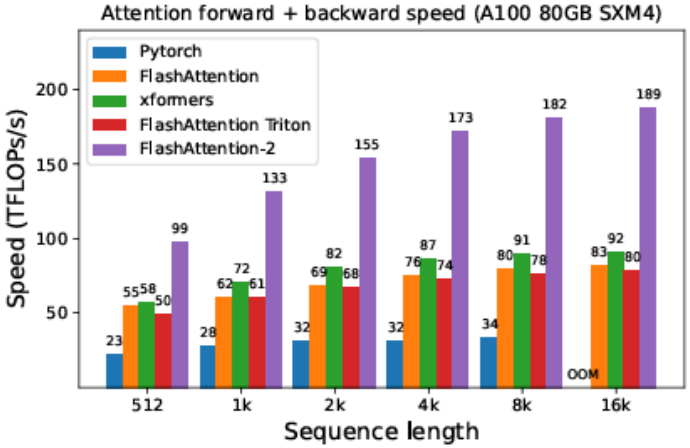
(a) Without causal mask, head dimension 64



(b) Without causal mask, head dimension 128



(c) With causal mask, head dimension 64



(d) With causal mask, head dimension 128

End-to-End Training (GPT-style, 8×A100 80GB)

Table 1: Training speed (TFLOPs/s/GPU) of GPT-style models on 8×A100 GPUs. FlashAttention-2 reaches up to 225 TFLOPs/s (72% model FLOPs utilization). We compare against a baseline running without FlashAttention.

Model	Without FlashAttention	FlashAttention	FlashAttention-2
GPT3-1.3B 2k context	142 TFLOPs/s	189 TFLOPs/s	196 TFLOPs/s
GPT3-1.3B 8k context	72 TFLOPS/s	170 TFLOPs/s	220 TFLOPs/s
GPT3-2.7B 2k context	149 TFLOPs/s	189 TFLOPs/s	205 TFLOPs/s
GPT3-2.7B 8k context	80 TFLOPs/s	175 TFLOPs/s	225 TFLOPs/s

The Core Insight

Hardware reality
SRAM 10× faster than HBM

Tiling + recompute
avoid $O(N^2)$ HBM traffic

Parallelize N
sequence dimension matters

Split-Q not Split-K
no inter-warp sync

2× over FA-1
at A100, fwd+bwd

50-73% peak
FLOPs/s utilization

16k ctx = 8k cost
context length unlocked

Exact attention
zero approximation

THANKS

NISHA KUMAR