CS 2650: Paper Discussion

FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness

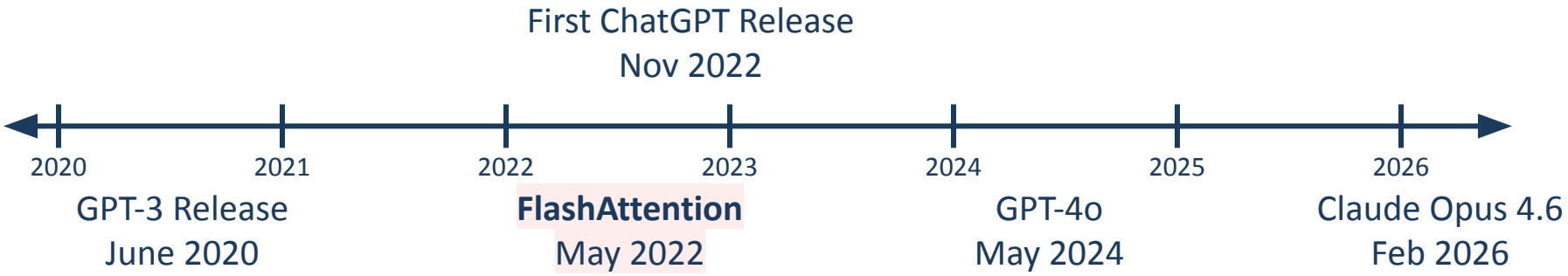
Tri Dao[†], Daniel Y. Fu[†], Stefano Ermon[†], Atri Rudra[‡], and Christopher Ré[†]

March 10, 2026

Annie Liu, Andrew Ma, Mert Akin, Ryan Liu


AI
LLM
Transformers
Attention

Timeline



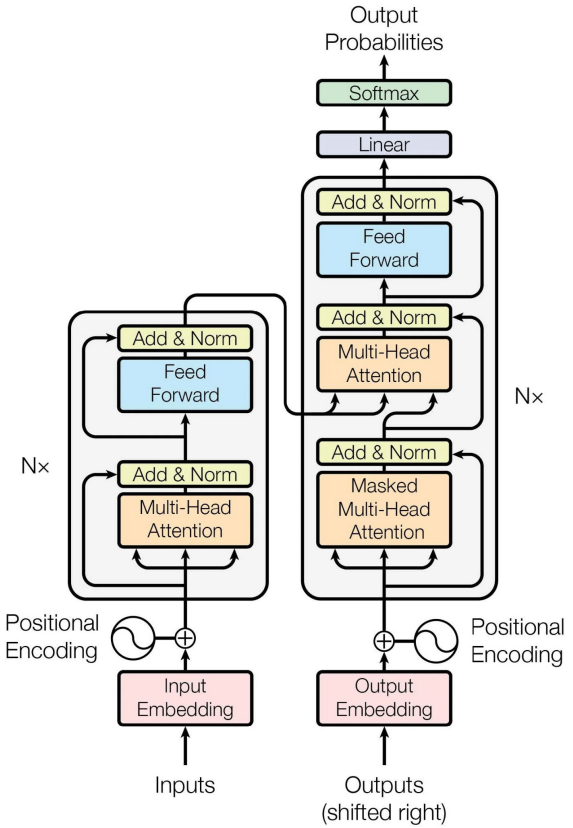
This paper was written before LLMs were mainstream

Introduction

Transformers Are Everywhere

Longer context enables:

- Long-document reasoning
- Code understanding
- Retrieval augmentation
- Genomics



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

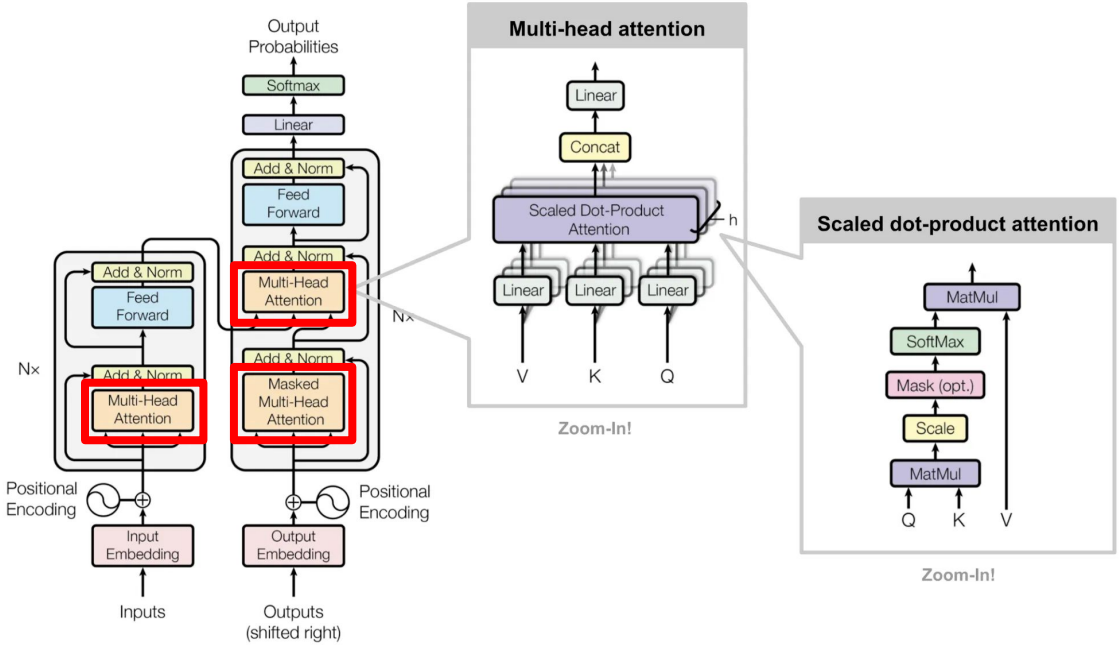
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

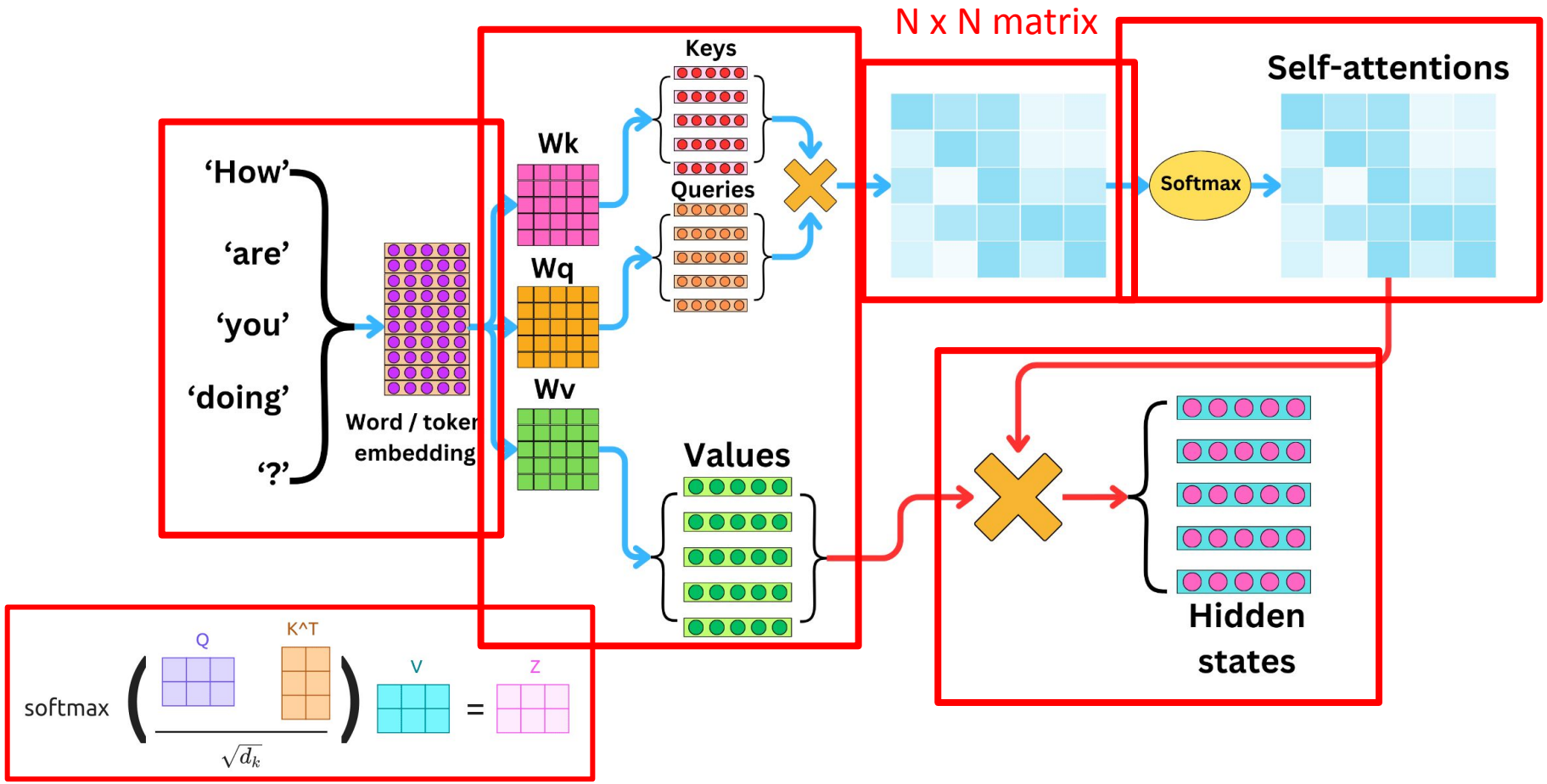
The Issue With Longer Contexts

What powers longer context?
Self-attention!

Scaling context is hard:
self-attention is quadratic in sequence length



⚠️ **Self-attention: time & memory complexity is $O(N^2)$ in sequence length N**



Source: https://newsletter.theaiedge.io/p/understanding-the-self-attention

Prior Work To Speed Attention

What they do

- Sparse attention (local windows, strided)
- Low-rank approximation
- Theoretically attractive
- Focus on reducing FLOPs:
  O(N) or O(N log N)

Why they fail in practice

- Ignore IO cost entirely
- No real wall-clock speedup
  - FLOPs are not correlated with speed
- No widespread adoption

💡 **FLOPs don't tell the full story!**

If attention is **quadratic**,
is the **bottleneck**
compute OR ***memory movement***?

Compute-Bound vs Memory-Bound

Arithmetic Intensity = FLOPs per byte of memory access

💾 Memory-Bound

Time determined by memory access.
Computation is fast, but waiting for data.

Examples:

- Elementwise: activation, dropout
- Reduction: ***softmax***, layernorm, sum

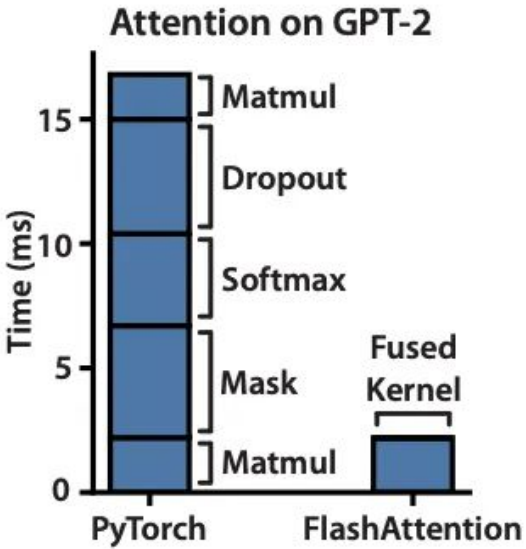
⚙️ Compute-Bound

Time determined by arithmetic ops.
HBM access is smaller bottleneck.

Examples:

- Matrix multiplication (large inner dim)
- Convolution (many channels)

The Real Bottleneck to Tackle



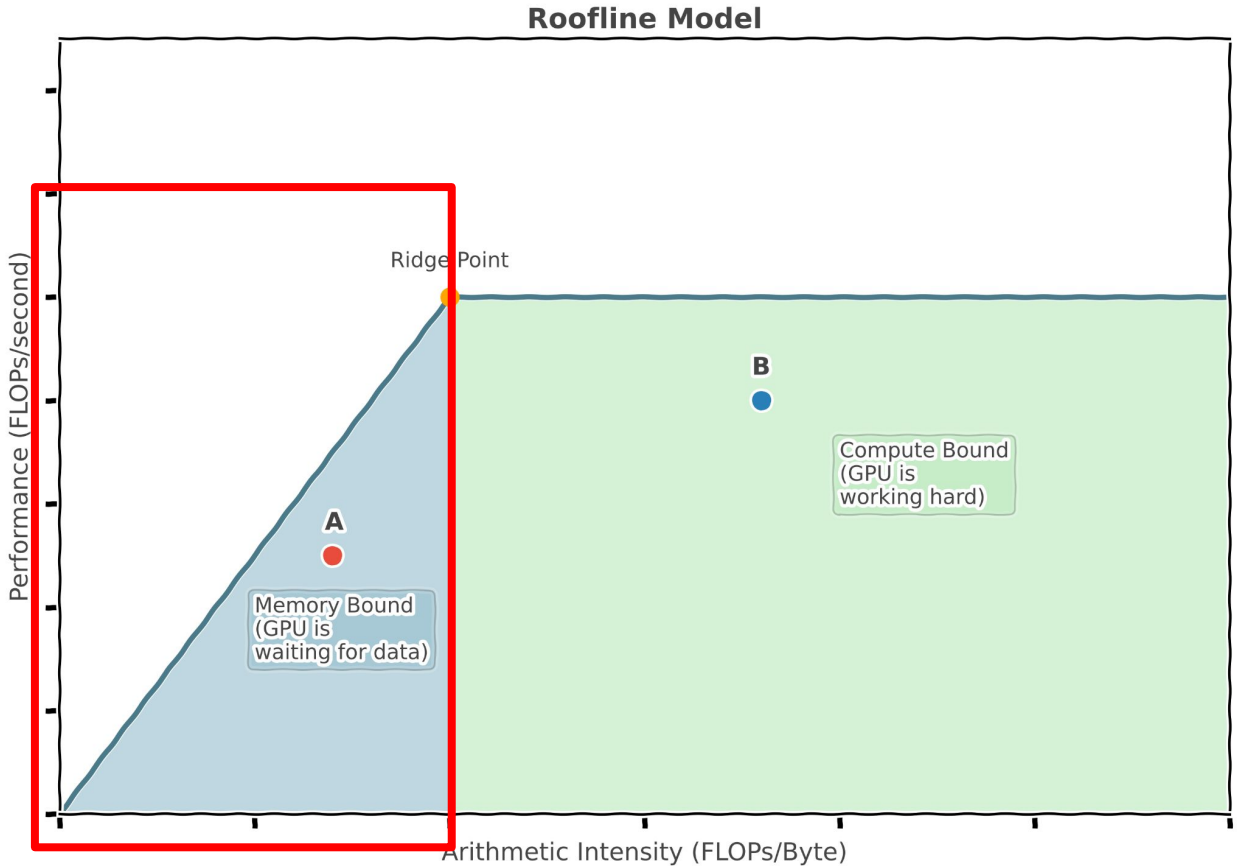
Modern GPUs make computations like matrix multiplication very fast.

BUT many operations in attention are **memory-bound** instead.
Reading/writing to GPU memory (HBM) dominates overall runtime.

This same bottleneck exists in databases, image processing, and many other fields.

💡 **FlashAttention tackles this memory movement bottleneck.**

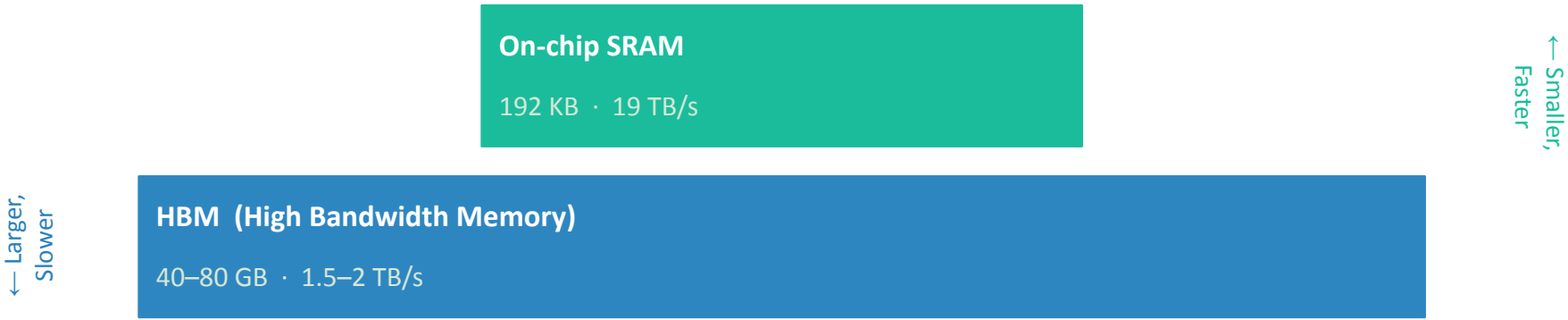
Compute-Bound vs Memory-Bound



Our focus!

GPU Memory Hierarchy

SRAM is 10× faster than HBM but MUCH tinier! (192 KB vs 40+ GB).
The goal: **keep data in SRAM as long as possible.**

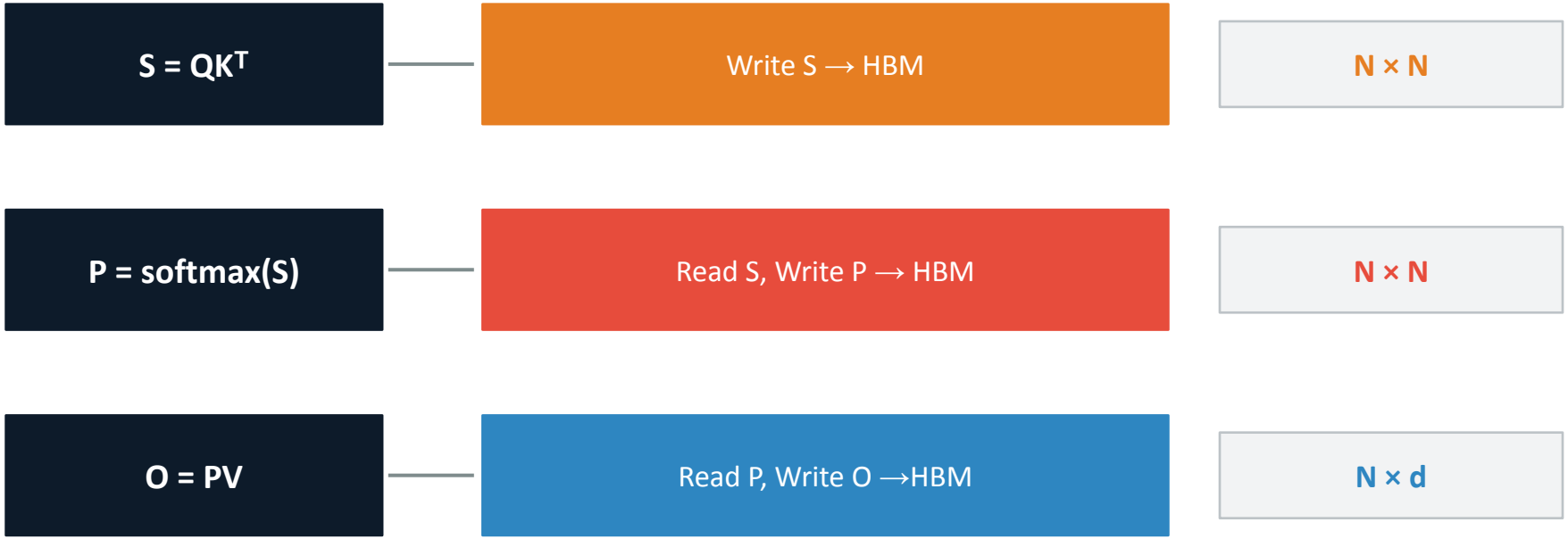


How GPU Kernels Execute

On GPUs, every operation runs as a kernel, so each kernel.:.

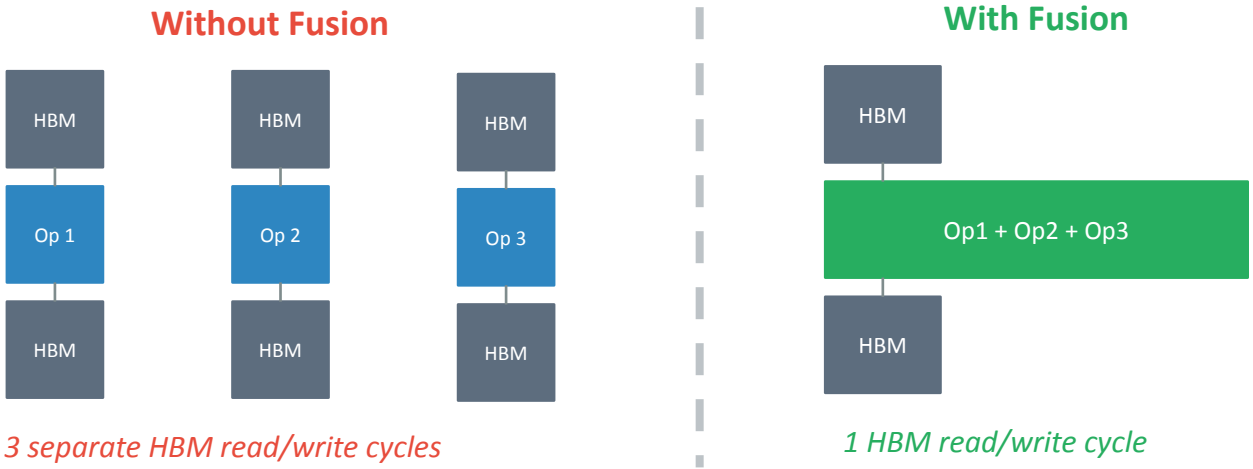
💡 **FlashAttention reduces how often we have to go back to HBM!**

How Memory Blows Up



S and P are both N×N → massive HBM traffic: write S, read S, write P, read P, write O

A Naive Fix: Kernel Fusion



⚠️ **But this doesn't work for attention!**

1. The attention matrix is too large to keep on-chip
2. Training requires storing intermediate matrices for the backward pass

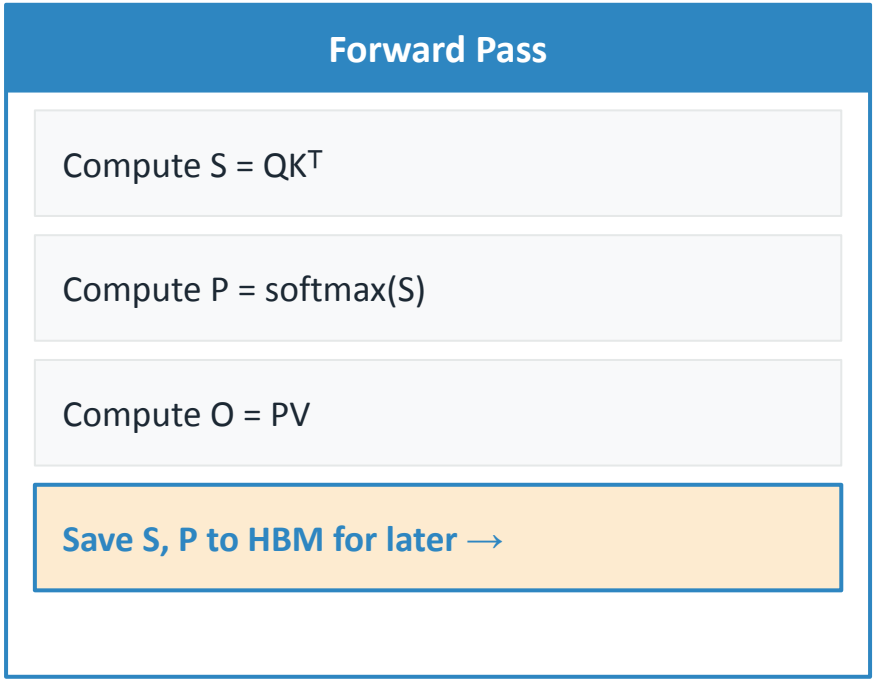
Why Can't We Just Discard S and P?

Inference:

1. Forward pass

Training:

1. **Forward pass**
   a. Store S, P to HBM
2. Compute loss
3. **Run backward pass to compute gradients**
   a. Load S, P from HBM
4. Update weights



Why Can't We Just Discard S and P?

Inference:

1. Forward pass

Training:

1. **Forward pass**
   a. Store S, P to HBM
2. Compute loss
3. **Run backward pass to compute gradients**
   a. Load S, P from HBM
4. Update weights

Backward Pass

Need P to compute $\partial L/\partial V$

Need S to compute $\partial L/\partial P$

Need P to compute $\partial L/\partial Q$, $\partial L/\partial K$

Reload S, P from HBM ←

Standard Attention: IO Complexity

HBM Accesses: $\Theta(Nd + N^2)$

Nd — Loading Q, K, V

Each of Q, K, V is $N \times d$
$\longrightarrow$ Linear in sequence length N

N^2 — Storing & Reading S, P

$S = QK^T$ and $P = \text{softmax}(S)$
both $N \times N \longrightarrow$ quadratic!

The N^2 term dominates for long sequences. This is what we must eliminate.

Can we compute
exact attention
without writing ***NxN*** matrices
to memory?

Yes, with FlashAttention!

Solution at a Glance

💡 **What is FlashAttention?**

An algorithm that computes **exact attention** (no approximation) with far fewer memory accesses.

✅ **Implementation**

Implemented in **CUDA** for fine-grained memory control.

Fuses all attention ops into one GPU kernel.

Results

2–3×

Faster attention computation
=> faster training of existing methods

O(N)

Lower bound for exact attention
=> enables training of larger models

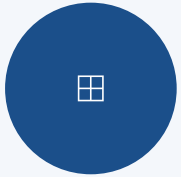
Extendible

Useful primitive for other approximate attention algorithms

Methodology

FlashAttention: Two Ideas, One Elegant Solution

Goal: compute exact attention without materializing the N×N matrix in slow memory



Tiling

Split Q, K, V into small blocks that fit in fast SRAM. Compute attention tile-by-tile entirely on-chip — never forming the full N×N matrix.

Works trivially for matmul

Online Softmax

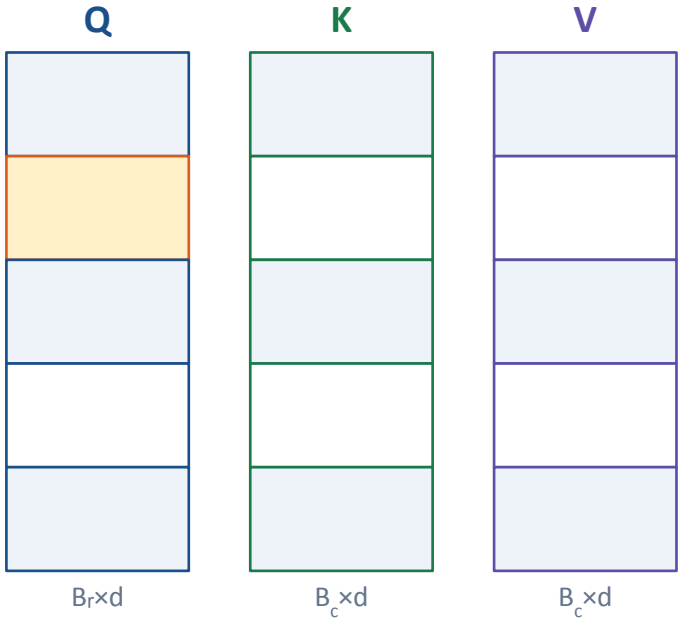
Track running max m and sum ℓ as we process each tile. Merge statistics incrementally with rescaling — exact softmax without seeing the full row.

Solves the softmax tiling problem

Together → One fused CUDA kernel · O(N) memory · Exact result · 2–4× faster

Tiling: Compute Attention Block by Block

Q, K, V split into blocks:



load block

On-chip SRAM — 19 TB/s · ~20 MB

tile ($B_r \times B_c$)

GPU HBM — 1.5 TB/s · 40 GB

Q, K, V, O (full matrices live here)

Block sizes: $B_c = \lceil M/4d \rceil$ and $B_r = \min(\lceil M/4d \rceil, d)$ — chosen so 4 blocks fit in SRAM of size M. That's it.

Matmul tiles trivially — partial sums just accumulate. But softmax needs the full row. So what do we do?

Tiling: Why Softmax Breaks It

$$\text{softmax}(x)_i = \exp(x_i) \,/\, \Sigma_j \exp(x_j) \quad \leftarrow \textit{needs ALL j simultaneously}$$

What happens if we naïvely softmax each tile separately (row of N attention scores, split into T_c tiles of B_c each)?

Tile 1

x_1 x_2 x_3

local denom:
$\exp(x_1 - \tilde{m}) + \exp(x_2 - \tilde{m}) + \exp(x_3)$

Tile 2

x_4 x_5 x_6

local denom:
$\exp(x_4 - \tilde{m}) + \exp(x_5 - \tilde{m}) + \exp(x_6 - \tilde{m})$

Tile 3

x_7 x_8 x_9

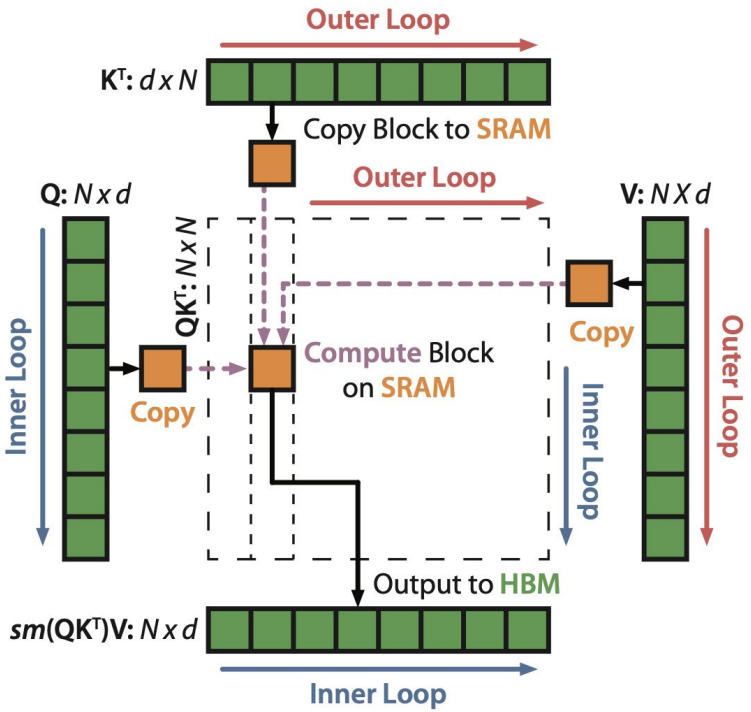
local denom:
$\exp(x_7 - \tilde{m}) + \exp(x_8 - \tilde{m}) + \exp(x_9 - \tilde{m})$

✗ **Each tile normalizes by its own local sum** → the three softmax outputs use different denominators and cannot be combined.

Softmax couples ALL elements of a row. You can't tile it naively — there's no fix after the fact.

Tiling: Loop Structure

Outer loop iterates over K/V blocks (j) — inner loop iterates over Q blocks (i)



For each K/V block j (outer), every Q block i (inner) computes tile S_{ij} on SRAM — merged online, written to HBM.

What Fits in SRAM?

B_r

Q Block Rows

Number of query rows loaded at once into SRAM

B_c

K, V Block Cols

Number of key/value rows loaded alongside each Q tile

M

SRAM Budget

Total on-chip SRAM — the constraint everything must fit within

Block size constraint: $2B_r \times d + 2B_c \times d \leq M \quad \rightarrow \quad B_c = \lceil M / 4d \rceil, B_r = \min(\lceil M / 4d \rceil, d)$

What lives on-chip simultaneously:

Try it yourself!

https://andrewma5.github.io/flashattention-softmax-demo/

Why Online Softmax is Non-Trivial

✗ Naïve Attempt

```
For each tile j:

  softmax_j = exp(S_ij - ~m_ij) / rowsum(exp(S_ij - ~m_ij))
```

$O_{\square} = \Sigma_{\square}\ \text{softmax\_j} \cdot V_{\square} \leftarrow$ incompatible scale

Why it fails:
softmax_1 is normalized by Σ(tile 1)
softmax_2 is normalized by Σ(tile 2)

These are different denominators. The outputs live on incompatible scales — you cannot add them.

✓ What We Need

A way to process tiles one at a time and end up with the correct globally-normalized result:

1. See tile 1, get partial result $\tilde{O}_1$
2. See tile 2, update $\tilde{O}_1 \rightarrow \tilde{O}_2$ using tile 2 info
3. … Continue for all T_c tiles
4. ✓ Final result = softmax(QK^T)V ✓

Key: carry (m, ℓ) as running stats, rescale at each step

The challenge: find statistics you can carry across tiles that enable exact global normalization after the fact.

Online Softmax: The Solution

Numerically Stable Softmax

m(x)	= max x_i	*row max*
$f(x)_i$	= $\exp(x_i - m(x))$	*shifted exp*
$\ell(x)$	= $\Sigma_i f(x)_i$	*normalizer*
softmax(x)	= f(x) / ℓ(x)	*output*

Block Decomposition ($x^{(1)} \cup x^{(2)} \rightarrow x$)

m(x)	= max($m(x^{(1)})$, $m(x^{(2)})$)	*merge maxes*
$\ell(x)$	= $\ell(x) = e^{(m^{(1)}-m)} \cdot \ell^{(1)} + e^{(m^{(2)}-m)} \cdot \ell^{(2)}$	*rescale*
softmax(x)	= f(x) / ℓ(x)	*exact* ✓

Key: carry only **(m, ℓ)** per row → merge any two blocks exactly → one block at a time

Track only (m, ℓ) per row → exact softmax one block at a time, never needing the full row.

What Is Stored Per Row

m

Running Max

Maximum attention score seen so far in this row. Keeps exp() from overflowing. Updated each block.

ℓ

Running Sum

Sum of all exp(s − m) values seen so far. The normalization denominator. Rescaled when m changes.

O

Partial Output

Accumulated weighted sum of V rows so far. Rescaled each block. Final answer after all tiles.

Update rule per new block j:

$$m_{new} = \max(m_i, m_{ij}) \qquad \ell_{new} = e^{m_i - m_{i\,new}} \cdot \ell_i + e^{m_{ij} - m_{i\,new}} \cdot \ell_{ij} \qquad O_{new} = \left(\ell_i \cdot e^{m_i - m_{i new}} \cdot O_i + e^{m_{ij} - m_{i new}} \cdot \tilde{P}_{ij} \cdot V_j \right) / \ell_{new}$$

Numeric Example: Setup

N=4, d=1, B^c=2 → 2 tiles of size 2

Q, K split into blocks first. Each tile S_i□ = Q_i × K□^{T} computed on SRAM:

Tile j=1	Tile j=2
$Q_1 \times K_1^T$ on SRAM	$Q_1 \times K_2^T$ on SRAM
[1.0, 3.0]	[2.0, 0.5]

← *Outer loop j=1 first, then j=2* →

Running stats before we start:

m_i (running max)	ℓ_i (running sum)	O_i (output so far)
−∞	0	0

Key question: can we get exact softmax([1.0, 3.0, 2.0, 0.5])·V without ever seeing the full row at once?

Tile j=1: Local Statistics

Load [1.0, 3.0] into SRAM — compute local softmax stats

Tile j=1 = [1.0, 3.0]

1 **Local max — for numerical stability**

$\tilde{m}_{i1}$ = max(1.0, 3.0) = **3.0**

→ *Subtract this from all scores before exp → prevents overflow*

2 **Shift and exponentiate**

e^(1.0 − 3.0) = e^(−2.0) = **0.135**

e^(3.0 − 3.0) = e^(0.0) = **1.0**

→ *These are the unnormalized attention weights $\tilde{P}_{i1}$ for this tile*

3 **Local sum — tile's own denominator**

$\tilde{\ell}_{i1}$ = 0.135 + 1.0 = **1.135**

→ *This is only the denominator for tile 1 — not the global denominator yet*

After tile 1: local stats are $\tilde{m}_{i1}$ = 3.0, $\tilde{P}_{i1}$ = [0.135, 1.0], $\tilde{\ell}_{i1}$ = 1.135 — all tile-local, not global yet.

Tile j=1: Update Running Statistics

Merge tile 1 stats into global running stats — first iteration is trivial

Before tile 1:

m_i	ℓ_i	O_i
$-\infty$	0	0

New global max

$m_{new} = \max(-\infty, 3.0) = 3.0$

→ *First tile so trivially takes the tile's max*

New normalizer

$\ell_{new} = e^{(-\infty-3.0)} \cdot 0 + e^{(3.0-3.0)} \cdot 1.135 = 0 + 1.0 \cdot 1.135 = 1.135$

→ *Old ℓ_i=0 contributes nothing; new tile's $\tilde{\ell}_{i1}$ = 1.135 carries over directly*

Partial output after tile 1

$O^{(1)} = (0.135 \cdot V_1 + 1.0 \cdot V_2) / 1.135$

→ *Normalized by ℓ_i^{new}=1.135 — correct for tile 1 only, will be rescaled when tile 2 arrives*

Running stats after tile 1:

$m_i = 3.0$	$\ell_i = 1.135$	$O_i = O_i^{(1)}$

Now we load tile 2. The running stats (m, ℓ, O) are all we carry forward — no tile 1 data kept in SRAM.

Tile j=2: Local Statistics

Load [2.0, 0.5] into SRAM — compute local softmax stats

Tile j=2 = [2.0, 0.5]

1 **Local max**

$\tilde{m}_{i2} = \max(2.0, 0.5) = 2.0$

→ *Different from $m_i = 3.0$ — this is exactly the incompatibility from the previous slides*

2 **Shift by local max and exponentiate**

$e^{(2.0 - 2.0)} = e^{(0.0)} = 1.0$

$e^{(0.5 - 2.0)} = e^{(-1.5)} = 0.223$

→ *$\tilde{P}_{i2} = [1.0, 0.223]$ — unnormalized weights for tile 2*

3 **Local sum**

$\tilde{\ell}_{i2} = 1.0 + 0.223 = 1.223$

→ *Tile 2's local denominator — normalized by $\tilde{m}_{i2}=2.0$, not the global max m=3.0*

$\tilde{m}_{i2} = 2.0 \neq m_i = 3.0$ — the running max from tile 1 is larger. Online softmax handles this with rescaling.

Tile j=2: Merge Stats — Online Softmax

This is the key step — rescale to the new global max

Running stats coming in from tile 1:

$m_i = 3.0$ | $\ell_i = 1.135$

New global max

$m_{new} = \max(3.0, 2.0) = 3.0$ ← unchanged

→ *Tile 2's max (2.0) is smaller than the running max (3.0) — no change needed*

Tile j=2: Merge Stats — Online Softmax

This is the key step — rescale to the new global max

Running stats coming in from tile 1:

$m_i = 3.0$ | $\ell_i = 1.135$

New global max

$m^{new} = \max(3.0, 2.0) = 3.0$ ← unchanged

→ *Tile 2's max (2.0) is smaller than the running max (3.0) — no change needed*

Rescale old ℓ_i to new max

$e^{(3.0-3.0)} \cdot 1.135 = 1.0 \cdot 1.135 = 1.135$

→ *Since max didn't change, $e^{(m_i - m_i^{new})} = e^0 = 1.0$ — old ℓ_i carries over unchanged*

Scale new tile's $\tilde{\ell}_{i2}$ to global max

$e^{(2.0-3.0)} \cdot 1.223 = 0.368 \cdot 1.223 = 0.450$

→ *$e^{(\tilde{m}_{i2} - m_i^{new})} = e^{(2.0-3.0)} = 0.368$ — corrects tile 2's sum to be consistent with global max 3.0*

Tile j=2: Merge Stats — Online Softmax

This is the key step — rescale to the new global max

Running stats coming in from tile 1:

$m_i = 3.0$ | $\ell_i = 1.135$

New global max

$m_{new} = \max(3.0, 2.0) = 3.0$ ← unchanged

→ *Tile 2's max (2.0) is smaller than the running max (3.0) — no change needed*

Rescale old ℓ_i to new max

$e^{(3.0-3.0)} \cdot 1.135 = 1.0 \cdot 1.135 = 1.135$

→ *Since max didn't change, $e^{(m_i - m_i^{new})} = e^0 = 1.0$ — old ℓ_i carries over unchanged*

Scale new tile's $\tilde{\ell}_{i2}$ to global max

$e^{(2.0-3.0)} \cdot 1.223 = 0.368 \cdot 1.223 = 0.450$

→ *$e^{(\tilde{m}_{i2} - m_i^{new})} = e^{(2.0-3.0)} = 0.368$ — corrects tile 2's sum to be consistent with global max 3.0*

New global normalizer

$\ell_{new} = 1.135 + 0.450 = 1.585$

→ *This is the exact softmax denominator for the full row [1.0, 3.0, 2.0, 0.5]*

$\ell_i^{new} = 1.585$ is the exact global denominator — computed without ever seeing all 4 scores simultaneously.

Tile j=2: Update Output

Rescale old output, add new tile contribution, normalize

① **Rescale old $O_i^{(1)}$ to new global max**

$$e^{(m^{(1)} - m^{new})} \cdot O^{(1)} = e^{(3.0-3.0)} \cdot O^{(1)} = 1.0 \cdot O^{(1)}$$

→ *Max didn't change so e^0 = 1.0 — $O_i^{(1)}$ carries over with no modification*

Tile j=2: Update Output

Rescale old output, add new tile contribution, normalize

① **Rescale old $O_i^{(1)}$ to new global max**

$$e^{(m_{i1} - m_i^{new})} \cdot O_i^{(1)} = e^{(3.0 - 3.0)} \cdot O_i^{(1)} = 1.0 \cdot O_i^{(1)}$$

→ *Max didn't change so e^0 = 1.0 — $O_i^{(1)}$ carries over with no modification*

② **New tile contribution — scaled to global max**

$$e^{(\tilde{m}_{i2} - m_i^{new})} \cdot \tilde{P}_{i2} \cdot V = e^{(2.0 - 3.0)} \cdot [1.0,\ 0.223] \cdot V = 0.368 \cdot \tilde{P}_{i2} \cdot V$$

→ *$e^{(\tilde{m}_{i2} - m_i^{new})} = 0.368$ rescales tile 2's weights to be consistent with global max*

Tile j=2: Update Output

Rescale old output, add new tile contribution, normalize

① **Rescale old $O_i^{(1)}$ to new global max**

$$e^{(m^{(1)} - m^{new})} \cdot O^{(1)} = e^{(3.0-3.0)} \cdot O^{(1)} = 1.0 \cdot O^{(1)}$$

→ *Max didn't change so e^0 = 1.0 — $O_i^{(1)}$ carries over with no modification*

② **New tile contribution — scaled to global max**

$$e^{(\tilde{m}_{i2} - m^{new})} \cdot \tilde{P}_{i2} \cdot V = e^{(2.0-3.0)} \cdot [1.0,\ 0.223] \cdot V = 0.368 \cdot \tilde{P}_{i2} \cdot V$$

→ *$e^{(\tilde{m}_{i2} - m_i^{new})} = 0.368$ rescales tile 2's weights to be consistent with global max*

③ **Combine and normalize by ℓ_i^{new}**

$$O^{new} = (\ell^{(1)} \cdot 1.0 \cdot O^{(1)} + 0.368 \cdot P_{i2} \cdot V) / 1.585$$

→ *$\ell_i = 1.135$ weights the old output, $0.368 \cdot \tilde{P}_{i2}$ weights the new tile*

$$O_i^{new} = \mathrm{diag}(\ell_i^{new})^{-1} \left(\mathrm{diag}(\ell_i) \cdot e^{(m_i - m_i^{new})} \cdot O_i + e^{(\tilde{m}_{ij} - m_i^{new})} \cdot \tilde{P}_{ij} V_j \right)$$

After both tiles: O_i^{new} is the exact attention output for this query row — S was never written to HBM.

Verify: Is the Answer Exact?

Compare online softmax result vs standard softmax on full row

Standard Softmax (full row)

Subtract global max = 3.0:

exp(1.0−3.0) = 0.135

exp(3.0−3.0) = 1.0

exp(2.0−3.0) = 0.368

exp(0.5−3.0) = 0.082

Sum = 0.135+1.0+0.368+0.082

= 1.585

Online Softmax (our result)

After tile 1:

$\ell_1 = 1.135, \quad m_1 = 3.0$

After tile 2:

rescale: $0.368 \cdot 1.223 = 0.450$

$\ell_{new} = 1.135 + 0.450$

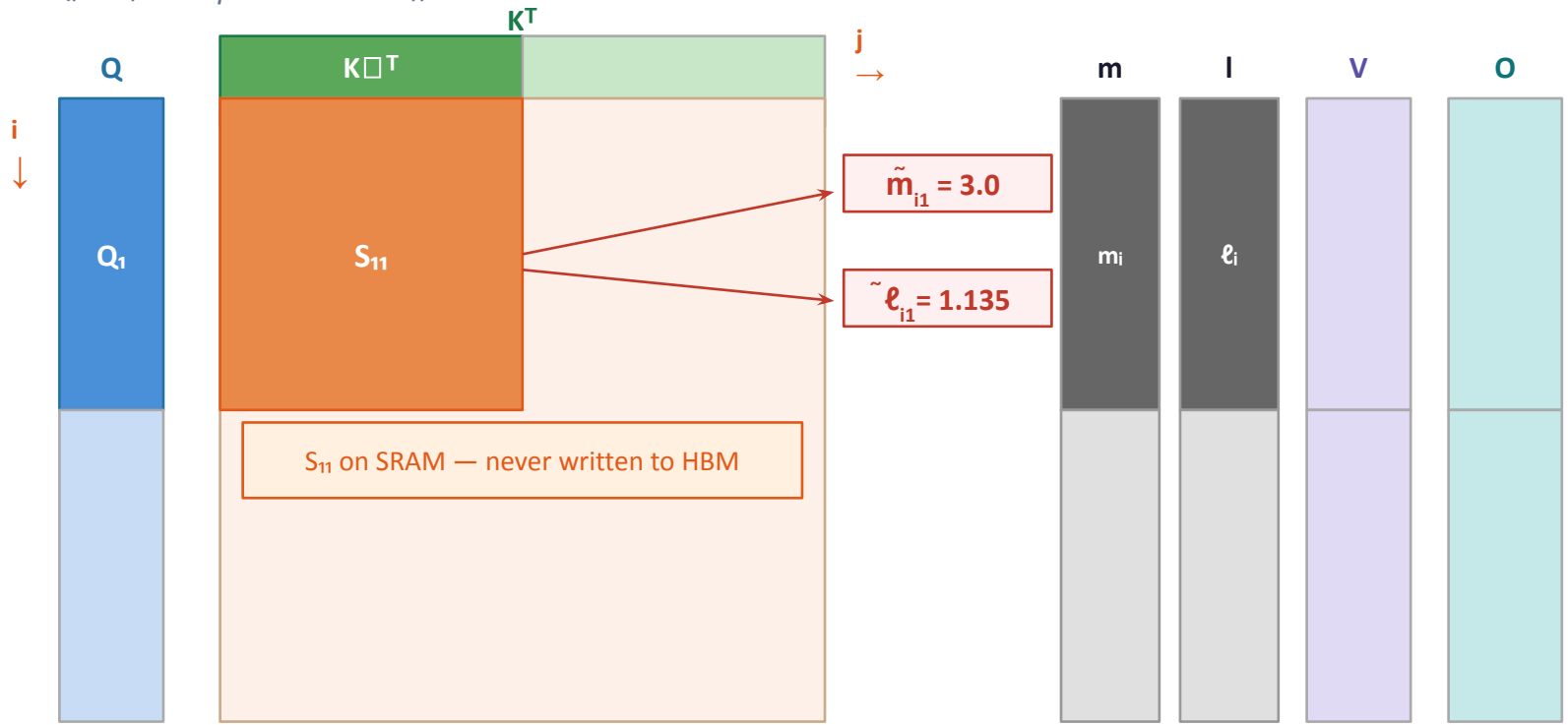
= 1.585 ✓

Both give $\ell = 1.585$ → same softmax weights → same output ✓

Exact match — not an approximation. Block size affects speed, not correctness. This is what Theorem 1 proves.

Tile j=1: What's Happening Spatially

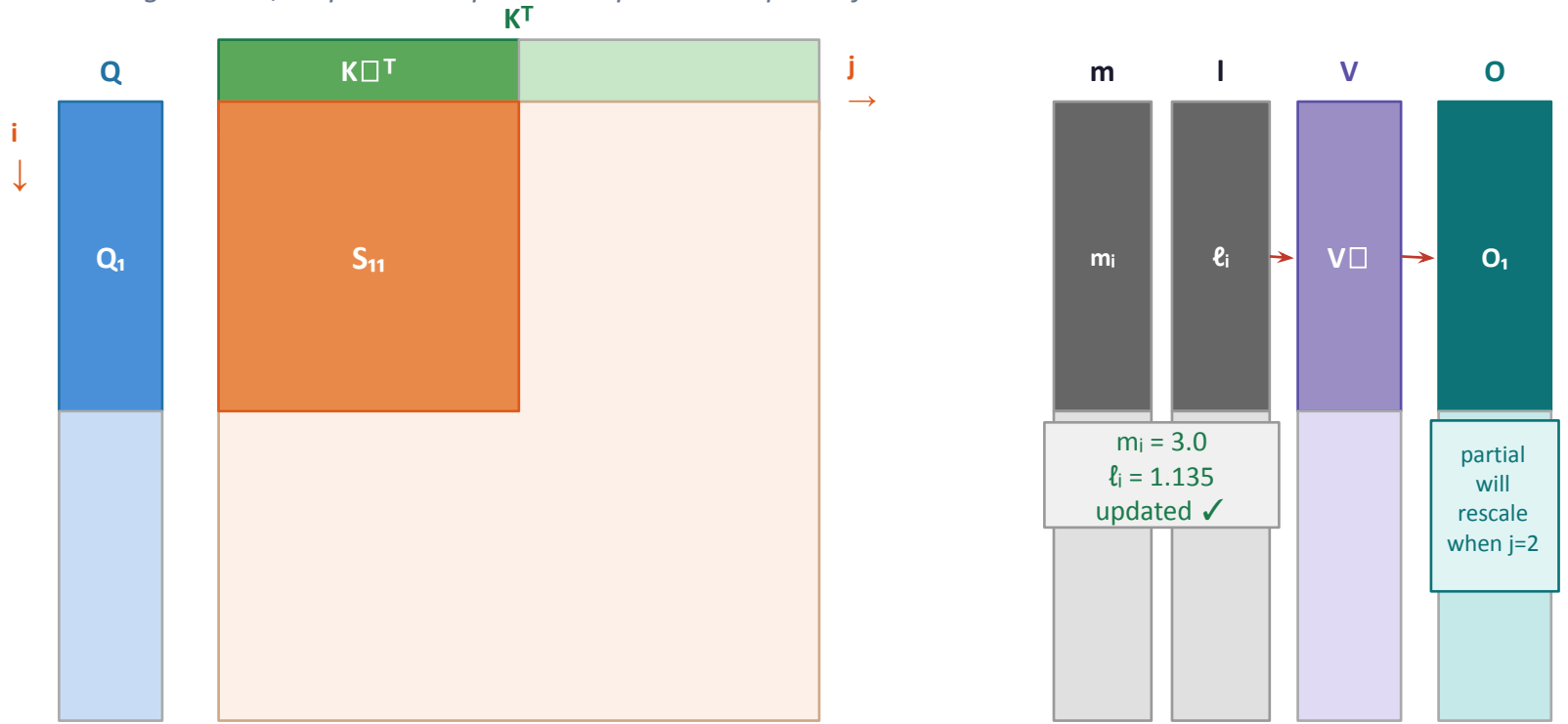
$Q_1 \times K_1^T$ computed as tile S_{11} on SRAM — local max and local sum extracted



$Q_1 \times K_1^T$ produces tile S_{11} on SRAM — local max $\tilde{m}_{i1}$ and local sum $\tilde{\ell}_{i1}$ extracted, S discarded.

Tile j=1: Partial Output Written

Running stats m_i, ℓ_i updated — partial output O_1 computed from S_{11} and written back to HBM

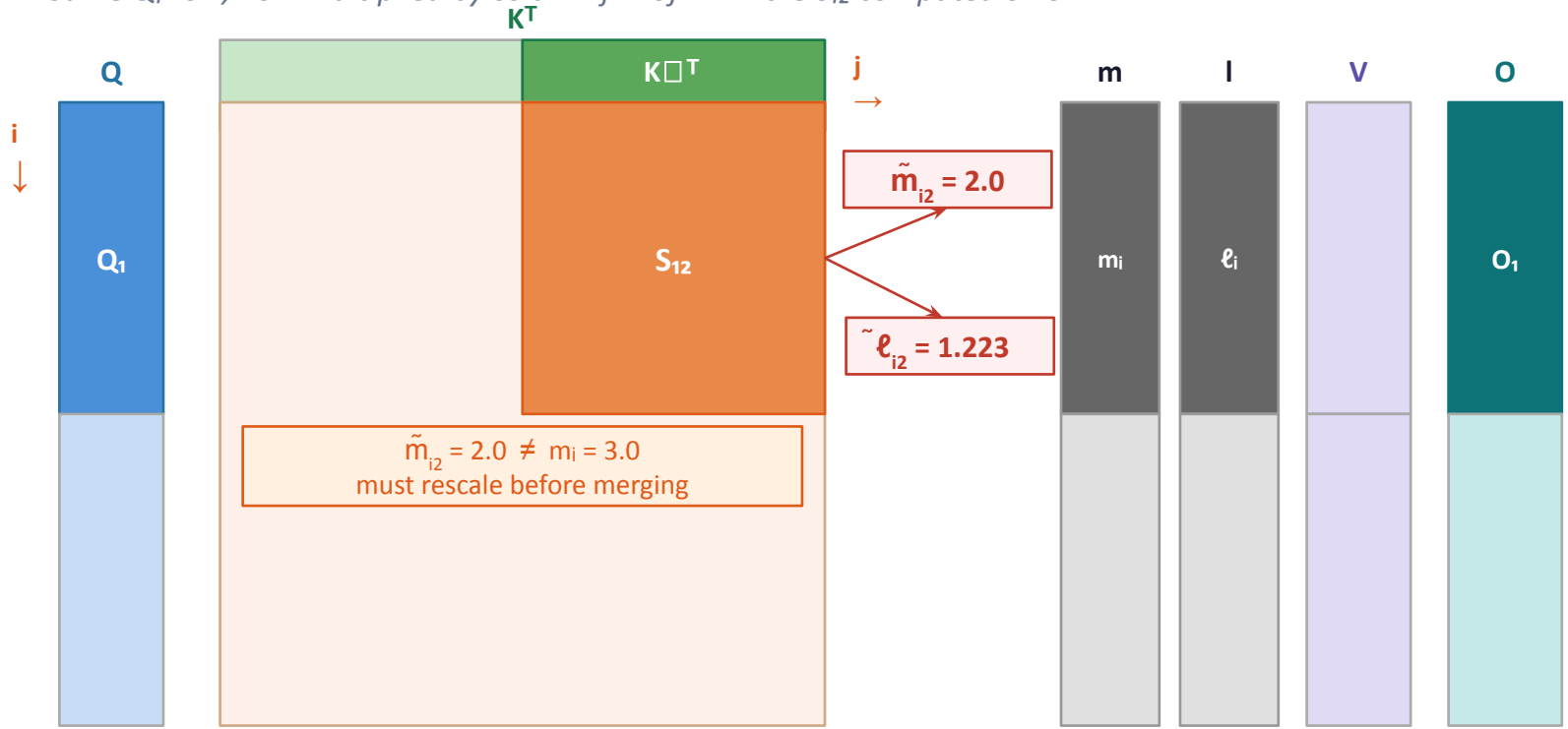


$$O_i^{(1)} = (0.135 \cdot v_1 + 1.0 \cdot v_2) \div 1.135$$

After tile j=1: S_{11} discarded from SRAM, m_i=3.0, ℓ_i=1.135, O_1 written to HBM.

Tile j=2: What's Happening Spatially

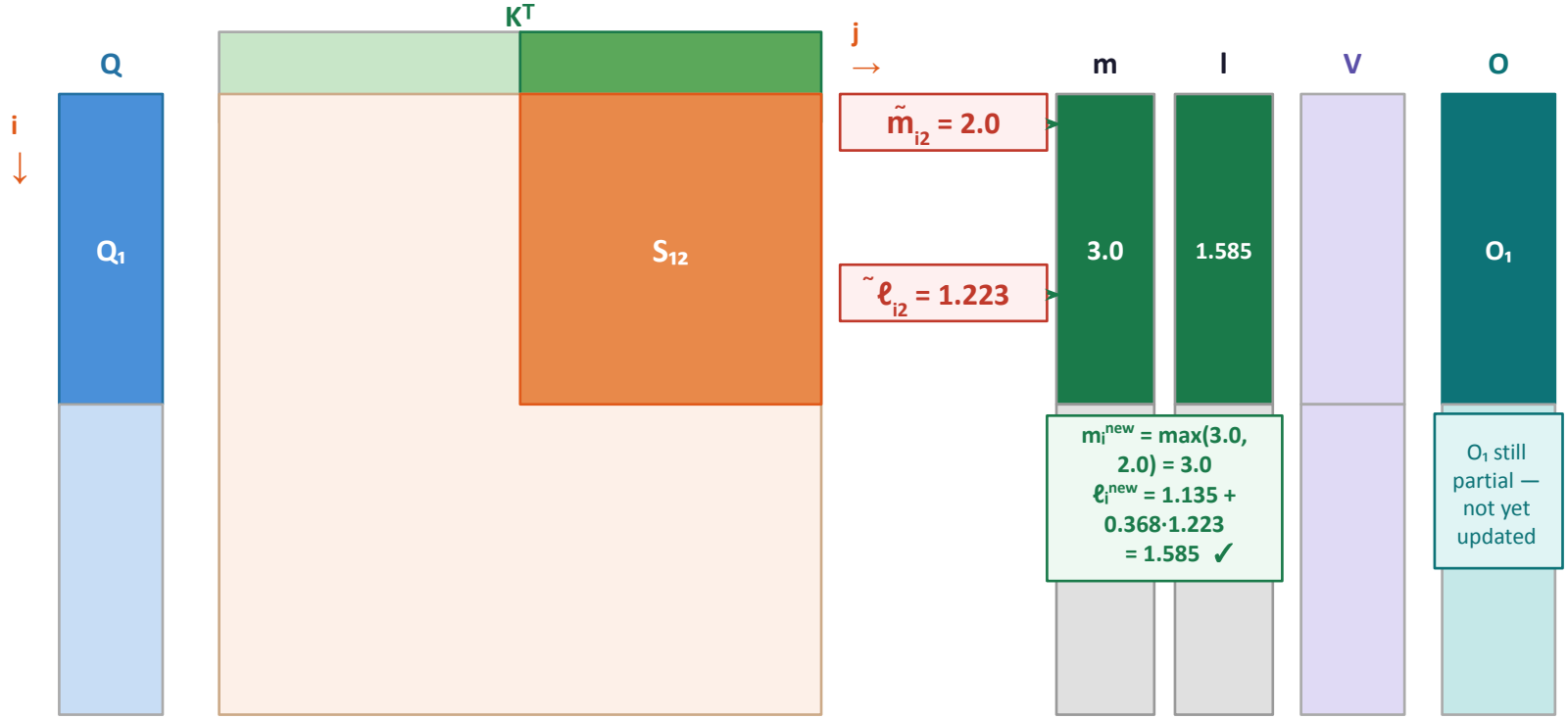
Same Q_1 row, now multiplied by column j=2 of K^T — tile S_{12} computed on SRAM



$Q_1 \times K_2^T$ produces S_{12} — local max $\tilde{m}_{i2}$=2.0 differs from running max m_i=3.0. Rescaling required.

Tile j=2: Merge Stats into Running Totals

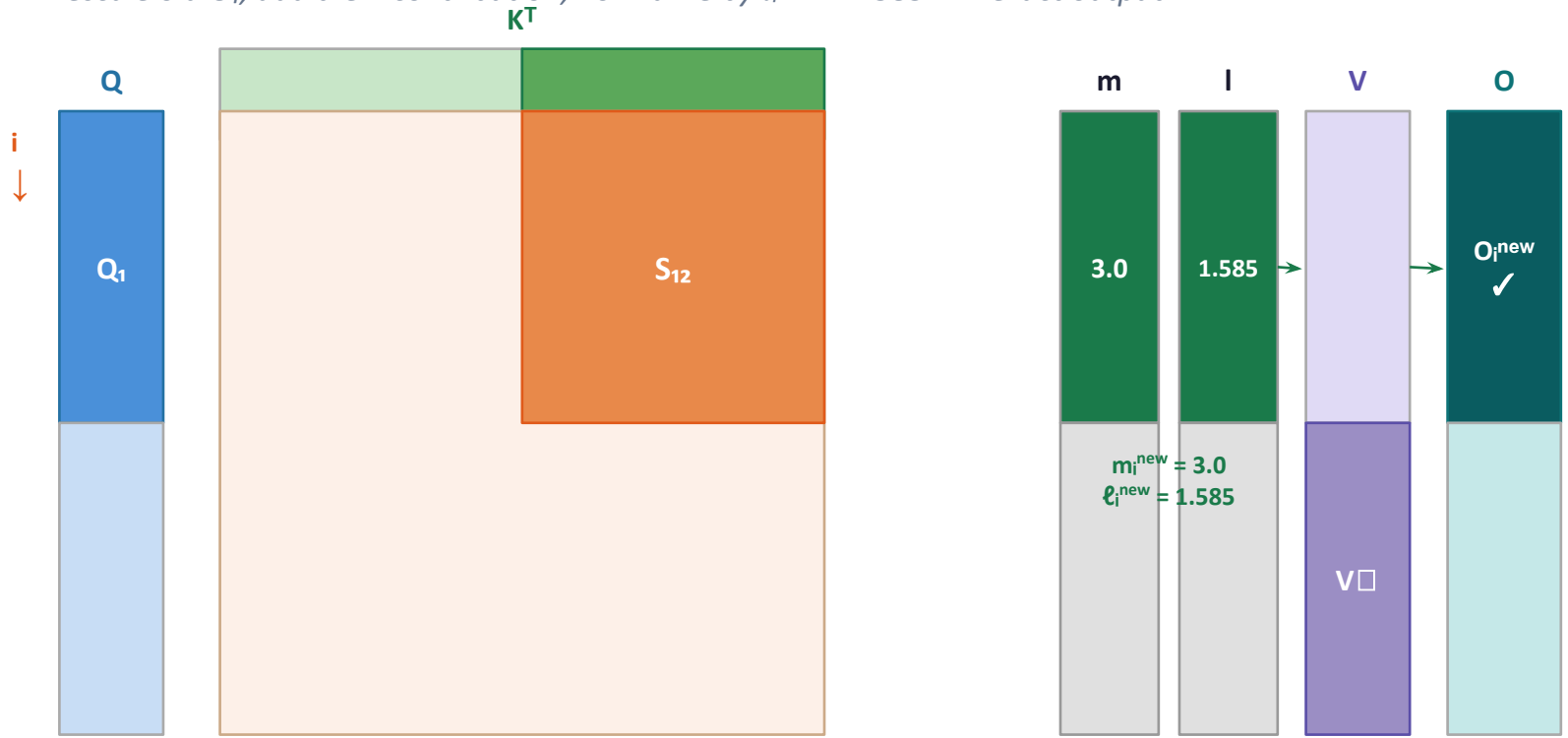
Rescale $\tilde{m}_{i2}$ and $\tilde{\ell}_{i2}$ to global max — write m_i^{new}=3.0 and ℓ_i^{new}=1.585 into m/l columns



$\tilde{m}_{i2}$=2.0 and $\tilde{\ell}_{i2}$=1.223 rescaled to global max 3.0 → merged into m_i^{new}=3.0, ℓ_i^{new}=1.585. O not yet updated.

Tile j=2: Final Output — Exact Result

Rescale old O_1, add tile 2 contribution, normalize by $\ell_i^{new} = 1.585 \longrightarrow$ exact output



$$O_i^{new} = \text{diag}(\ell_i^{new})^{-1} \cdot \left(\text{diag}(\ell_i) \cdot e^{(m_i - m_i^{new})} \cdot O_i + e^{(\tilde{m}_{ij} - m_i^{new})} \cdot \tilde{P}_{ij} V\square \right)$$

O_i^{new} is the exact attention output — S was never stored in HBM. This is what saves the memory.

What About Q_2?

Our numeric example only traces i=1. Here is the full 2×2 tile picture.

	j=1 (K_1^T)	j=2 (K_2^T)
i=1 (Q_1)	**S_{11}** $Q_1 \times K_1^T$ ✓ *covered in previous slides*	**S_{12}** $Q_1 \times K_2^T$ ✓ *covered in previous slides*
i=2 (Q_2)	**S_{21}** $Q_2 \times K_1^T$ *same process, independent of Q_1* *not shown*	**S_{22}** $Q_2 \times K_2^T$ *same process, independent of Q_1* *not shown*

Our numeric example traces the full i=1 row (S_{11} then S_{12}). Q_2 would repeat identically but independently.

Q_2 Repeats the Same Process Independently

No communication between query rows — each i has its own running stats

① **Own running stats**

Q_2 starts fresh with $m_i=-\infty$, $\ell_i=0$, $O_i=0$ — completely separate from Q_1

② **Same j sweep**

j=1 then j=2 — S_{21} and S_{22} computed, merged, output O_2 written to HBM row 2

③ **No shared state**

Q_1 and Q_2 never interact — this is why FlashAttention can parallelize across query blocks

④ **Memory stays bounded**

Only one tile of S ever lives on SRAM at a time, regardless of how many query rows there are

Each query row is self-contained — the online softmax runs independently per row, sweeping all K/V tiles.

Algorithm 1: FlashAttention — Full Picture

```
# Q,K,V ∈ ℝ□ˣ□  stored in HBM  →  output O ∈ ℝ□ˣ□

Init O□=0, ℓ□=0, m□=−∞  for all i

for j=1…T_c:  ▷ K,V tile loop
  Load K□,V□  HBM→SRAM
  for i=1…T_r:  ▷ Q tile loop
    Load Q□,O□,ℓ□,m□  HBM→SRAM
    S□□ = Q□K□□÷√d  ▷ on-chip
    m̃□□ = rowmax(S□□)
    P̃□□ = exp(S□□−m̃□□)
    ℓ̃□□ = rowsum(P̃□□)
    m□ⁿ□ᵂ = max(m□, m̃□□)
    ℓ□ⁿ□ᵂ = e^(m□−m□ⁿ□ᵂ)·ℓ□ + e^(m̃□□−m□ⁿ□ᵂ)·ℓ̃□□
    O□ⁿ□ᵂ = (ℓ□·e^(m□−m□ⁿ□ᵂ)·O□ + e^(m̃□□−m□ⁿ□ᵂ)·P̃□□V□) / ℓ□ⁿ□ᵂ
    Write O□,ℓ□,m□  SRAM→HBM
return O
```

Color Guide

- Init (O, ℓ, m)
- HBM ⟶ SRAM loads
- On-chip matmul (S_i□)
- Local softmax stats
- Online softmax + output
- Write back to HBM

① Init State

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,   ℓ□ = 0,   m□ = -∞  for all i
```

① Init State

1. **One row's state:** O_i, ℓ_i, m_i — initialized for every row of Q.
2. **No N×N matrix:** we only ever track O(N) values.
3. **$m_i = -\infty$:** ensures any real score will update the running max on the first tile.

② Outer Loop — Load K, V

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T_c:    ▷ K, V tile loop
   Load K□, V□    HBM → SRAM
```

② Outer Loop — Load K, V

1. **Outer loop over j:** iterates through T_c column blocks of K and V.
2. **Load once, reuse many:** K□ and V□ are loaded into SRAM once and held for all T_r inner iterations.
3. **Key savings:** this amortizes the HBM cost of K, V across every Q block.

③ Inner Loop — Load Q, O, ℓ, m

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T□:   ▷ K, V tile loop
   Load K□, V□   HBM → SRAM

   for i = 1…T_r:   ▷ Q tile loop
      Load Q□, O□, ℓ□, m□   HBM → SRAM
```

③ Inner Loop — Load Q, O, ℓ, m

1. **Inner loop over i:** iterates through T_r row blocks of Q.
2. **Load running state:** Q_i, O_i, ℓ_i, m_i come from HBM — these are the O(N) stats from prior iterations.
3. **Only 4 small loads per tile:** no full matrices ever loaded.

④ On-Chip Score Matrix

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T_c:    ▷ K, V tile loop
   Load K□, V□    HBM → SRAM

   for i = 1…T_r:    ▷ Q tile loop
      Load Q□, O□, ℓ□, m□    HBM → SRAM

      S□□ = Q□ K□□  ÷ √d    ▷ matmul on-chip
```

④ On-Chip Score Matrix

1. **$S_i□ \in \mathbb{R}^{BrxBc}$:** a small tile of the full N×N score matrix.
2. **Stays on SRAM:** computed entirely on-chip, never written to HBM.
3. **Heart of tiling:** standard attention writes all of N×N to HBM — FlashAttention never does.

💡 **S_{ij} and $\tilde{P}_{ij}$ are born and die in SRAM — they are never written to HBM.**

⑤ Local Softmax Statistics

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T□:   ▷ K, V tile loop
   Load K□, V□   HBM → SRAM

   for i = 1…T□:   ▷ Q tile loop
      Load Q□, O□, ℓ□, m□   HBM → SRAM

      S□□ = Q□ K□□ ÷ √d   ▷ matmul on-chip

      m̃□□ = rowmax(S□□)
      P̃□□ = exp(S□□ − m̃□□)
      ℓ̃□□ = rowsum(P̃□□)
```

⑤ Local Softmax Statistics

1. $\tilde{m}_{ij}$: local row max of this tile — used to shift exp for numerical stability.
2. $\tilde{P}_{ij}$: unnormalized attention weights for this tile — not yet globally correct.
3. $\tilde{\ell}_{ij}$: local sum — the denominator for this tile only.

💡 S_{ij} and $\tilde{P}_{ij}$ are born and die in SRAM — they are never written to HBM.

⑥ Rescale & Merge — Online Softmax

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T_c:    ▷ K, V tile loop
   Load K□, V□    HBM → SRAM

   for i = 1…T_r:    ▷ Q tile loop
      Load Q□, O□, ℓ□, m□    HBM → SRAM

      S□□ = Q□ K□□  ÷ √d    ▷ matmul on-chip

      m̃□□ = rowmax(S□□)
      P̃□□ = exp(S□□ − m̃□□)
      ℓ̃□□ = rowsum(P̃□□)

      m□ⁿᵉʷ = max(m□, m̃□□)
      ℓ□ⁿᵉʷ = e^(m□−m□ⁿᵉʷ) · ℓ□ + e^(m̃□□−m□ⁿᵉʷ) · ℓ̃□□
```

⑥ Rescale & Merge — Online Softmax

1. **m_i^{new}:** new global max = max of running max and this tile's max.
2. **Rescaling:** $e^{(m_i - m_i^{new})}$ corrects old ℓ_i; $e^{(\tilde{m}_{ij} - m_i^{new})}$ corrects new $\tilde{\ell}_{ij}$.
3. **Exact, not approximate:** this is the block-softmax decomposition — provably correct.

💡 S_{ij} and $\tilde{P}_{ij}$ are born and die in SRAM — they are never written to HBM.

⑦ Incremental Output Update

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T_c:    ▷ K, V tile loop
   Load K□, V□    HBM → SRAM

   for i = 1…T_r:    ▷ Q tile loop
      Load Q□, O□, ℓ□, m□    HBM → SRAM

      S□□ = Q□ K□□  ÷ √d   ▷ matmul on-chip

      m̃□□ = rowmax(S□□)
      P̃□□ = exp(S□□ − m̃□□)
      ℓ̃□□ = rowsum(P̃□□)

      m□ⁿᵉʷ = max(m□, m̃□□)
      ℓ□ⁿᵉʷ = e^(m□−m□ⁿᵉʷ) ·ℓ□ + e^(m̃□□−m□ⁿᵉʷ) ·ℓ̃□□

      O□ⁿᵉʷ = (ℓ□·e^(m□−m□ⁿᵉʷ) ·O□ + e^(m̃□□−m□ⁿᵉʷ) ·P̃□□V□) / ℓ□ⁿᵉʷ
```

⑦ Incremental Output Update

1. ① **Rescale old O_i:** multiply by $e^{(m_i - m_i^{new})}$ to adjust for updated global max.
2. ② **Add new tile:** $e^{(\tilde{m}_{ij} - m_i^{new})} \cdot \tilde{P}_{ij} V$□ — the highlighted yellow term was missing in the original slide!
3. ③ **Normalize:** divide by ℓ_i^{new} → exact result for all tiles seen so far.

💡 **S_{ij} and $\tilde{P}_{ij}$ are born and die in SRAM — they are never written to HBM.**

⑧ Write Back & Return

```
# Q,K,V ∈ ℝ□ˣ□  in HBM  →  output O ∈ ℝ□ˣ□

Init O□ = 0,  ℓ□ = 0,  m□ = −∞  for all i

for j = 1…T_c:    ▷ K, V tile loop
   Load K□, V□    HBM → SRAM

   for i = 1…T_r:    ▷ Q tile loop
      Load Q□, O□, ℓ□, m□    HBM → SRAM

      S□□ = Q□ K□□  ÷ √d   ▷ matmul on-chip

      m̃□□ = rowmax(S□□)
      P̃□□ = exp(S□□ − m̃□□)
      ℓ̃□□ = rowsum(P̃□□)

      m□ⁿᵉʷ = max(m□, m̃□□)
      ℓ□ⁿᵉʷ = e^(m□−m□ⁿᵉʷ) ·ℓ□ + e^(m̃□□−m□ⁿᵉʷ) ·ℓ̃□□

      O□ⁿᵉʷ = (ℓ□ ·e^(m□−m□ⁿᵉʷ) ·O□ + e^(m̃□□−m□ⁿᵉʷ) ·P̃□□V□)  /  ℓ□ⁿᵉʷ
```

⑧ Write Back & Return

1. **Write O_i, ℓ_i, m_i:** only O(N) scalars go back to HBM per inner iteration.
2. **S_{ij} and $\tilde{P}_{ij}$ discarded:** born and die on SRAM — never touch HBM.
3. **After all loops:** $O = \text{softmax}(QK^T)V$ exactly, with O(N) extra memory.

💡 **S_{ij} and $\tilde{P}_{ij}$ are born and die in SRAM — they are never written to HBM.**

Complexity Analysis : Define Variables

Variables

Symbol	Variable	Meaning
N	Sequence length	Number of tokens in the input sequence
d	Head dimension	Size of each query / key / value vector (e.g. 64 or 128)
M	Fast memory (SRAM)	Number of scalar floats that fit in on-chip GPU shared memory
Q, K, V	Input matrices	Query, Key, Value matrices ∈ ℝ^(N×d)
O	Output matrix	Attention output ∈ ℝ^(N×d)
B_n, B_r	Tiling block size	**B_n** — block size along columns (K/V dimension), **B_r** — block size along rows (Q dimension)

Why No N×N Matrix Is Ever Written

❌ Standard Attention

$S = QK^T$
→ *write N×N to HBM*

$P = \text{softmax}(S)$
→ *read + write N×N*

$O = PV$
→ *read N×N, write N×d*

HBM accesses: $O(Nd + N^2)$

VS

✅ FlashAttention

Load Q_i, K_j, V_j
SRAM tile load (HBM read)

Compute S_{ij} in SRAM
stays on-chip — never written to HBM

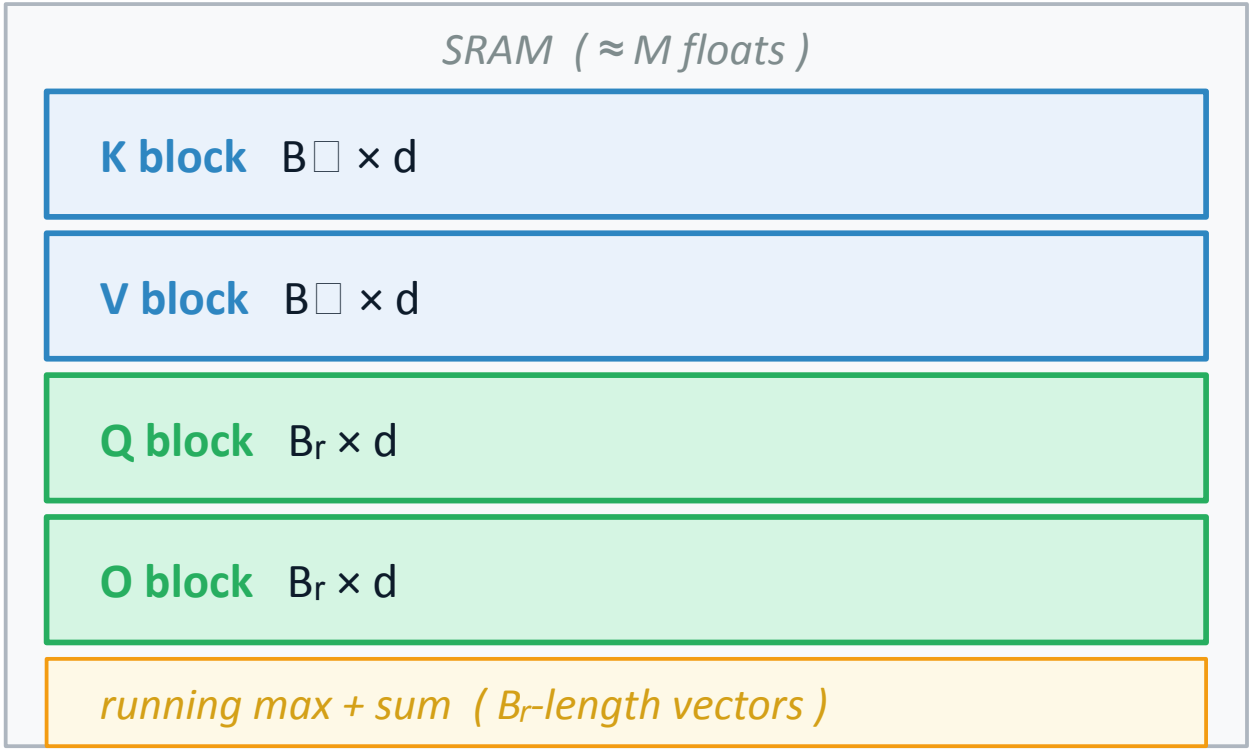
Update (m, ℓ, O) in SRAM
stays on-chip — never written to HBM

Write O_i once
written back each (i, j) iteration — S_{ij} never written

HBM accesses: $O(N^2d^2 / M)$ — far less!

FlashAttention: Why $\Theta(N^2d^2 / M)$

What lives in SRAM



SRAM constraint:

$$2B_{\square}d + 2B_r d + 2B_r \lesssim M$$

$$B_{\square}, B_r = \Theta(M/d)$$

FlashAttention: Why $\Theta(N^2d^2 / M)$

Where N^2d^2/M comes from

Outer loop over K/V blocks: N/B□ iterations

Inner loop over Q blocks: N/B_r iterations

Each step: load Q block ($B_r \times d$) from HBM

FlashAttention: Why $\Theta(N^2d^2 / M)$

Total Q reads = $(N/B_c)(N/B_r)(B_r d) = N^2 d / B_c$

Plug in $B_c = \Theta(M/d)$:

$N^2 d / (M/d) = \Theta(N^2 d^2 / M)$

Tight Lower Bound: FlashAttention Is Optimal

1

Must compute all N^2 dot-products $s_{ij} = q_i^T k_j$ for exact attention

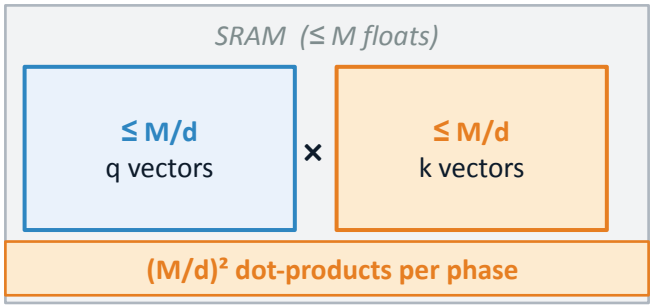
Tight Lower Bound: FlashAttention Is Optimal

2 Split execution into phases — each phase holds ≤ M scalars in SRAM

3 Each phase holds ≤ M/d query vectors and ≤ M/d key vectors

4 Dot-products per phase ≤ (M/d)(M/d) = $(M/d)^2$

Phase capacity



Tight Lower Bound: FlashAttention Is Optimal

5 Phases needed $\geq N^2 / (M/d)^2 = N^2d^2 / M^2$

6 Each phase moves $\Omega(M)$ data: needs fresh vectors from HBM

7 **Total I/O $\geq \Omega(N^2d^2/M^2) \cdot \Omega(M)$**

$$\Omega(N^2d^2 / M)$$

Lower bound matches FlashAttention: $\Theta(N^2d^2 / M)$ is optimal for exact attention **when $M \geq d^2$**

Gap in Proof?

6

Each phase moves $\Omega(M)$ data: needs fresh vectors from HBM

Block-Sparse FlashAttention

Core Idea

Restrict attention to a sparse mask — only compute scores s_{ij} where the mask allows. Skip entire K/V blocks that are fully masked.

Sparsity mask on N×N attention

Block-Sparse FlashAttention

IO Benefit

If fraction s of blocks are kept, I/O drops to $\Theta(s * N^2d^2 / M)$ — a factor of s^{-1} speedup over dense FlashAttention.

How it works

1. Determine a block sparsity mask $M_{ij} \in \{0,1\}$ for each (Q-block, K/V-block) pair
2. In the outer loop, skip any K/V block where mask = 0. No loads, no compute
3. Active blocks run identical FlashAttention kernel. Same tiling, same online softmax

When to use Block Sparse?

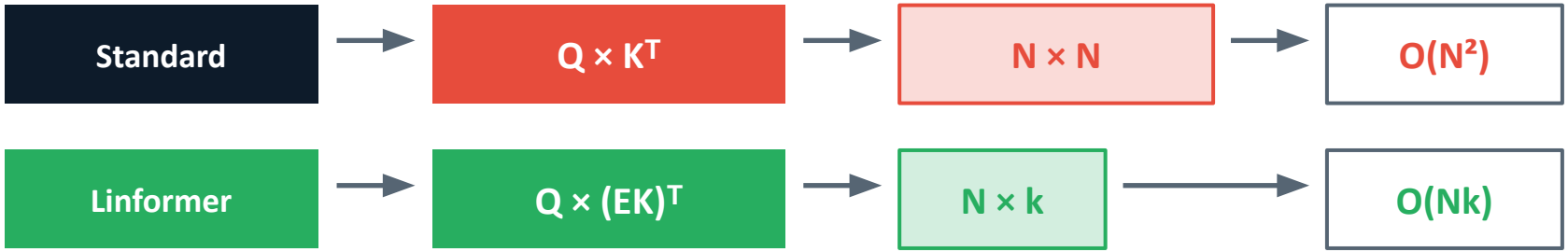
Approximation vs speed up

If you have a structured subset of blocks that approximates full attention

=> easily scale attention to longer sequences

Experiments

Linformer (2020)



E is a learned projection that compresses K, V from N → k rows (k ≪ N)

The catch: This is an approximation. Compressing K and V throws away fine-grained token interactions.

Reduces FLOPs on paper, but ignores memory IO cost → limited real-world speedup. No widespread adoption.

What FlashAttention Enabled

Faster Training

Before:

GPT-2 small: ~9.5 days (HuggingFace)

Even optimized Megatron-LM: ~4.7 days

With FlashAttention:

New BERT speed record (beat MLPerf)

GPT-2: up to 3.5× faster, same quality

Longer Context

Before:

Most models: ~2K token context

Approx. methods (Linformer) could reach ~64K, but quality suffered (not exact)

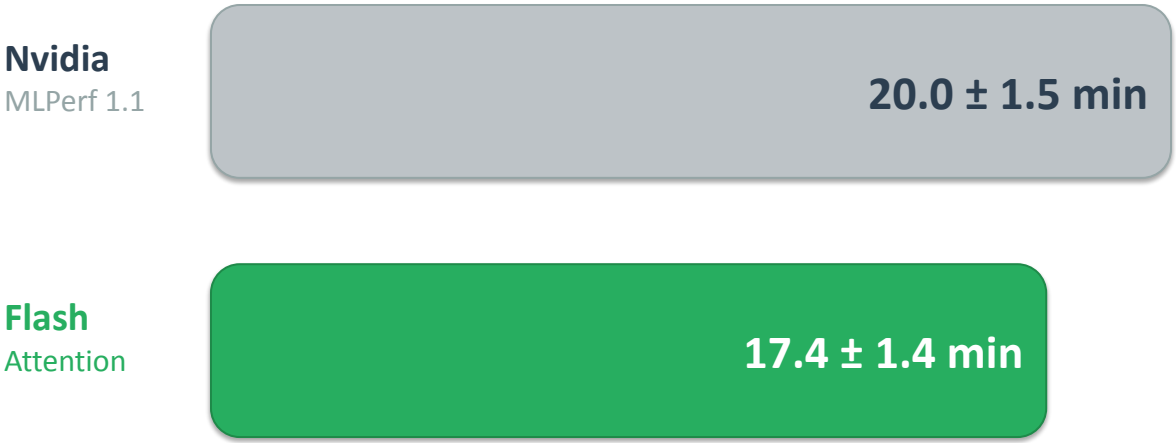
With FlashAttention:

Exact attention at 4K+ context

Better model quality, no approximation tradeoff

BERT Training Speed Record

BERT-large, 8×A100 GPUs, target 72.0% masked LM accuracy

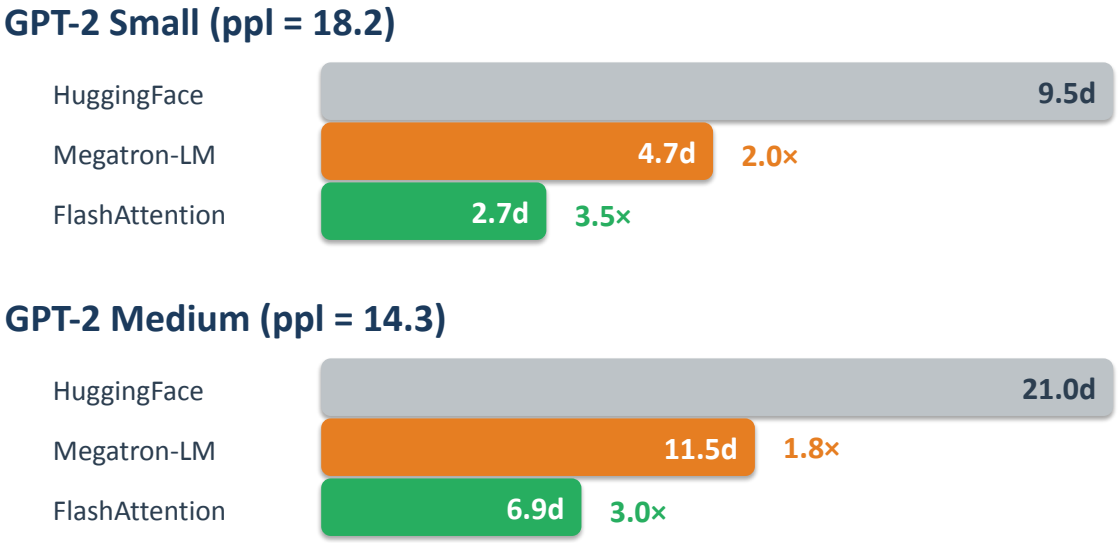


15% faster
than MLPerf record

Fastest single-node BERT training at time of publication
Same model, same hyperparameters, same target accuracy

GPT-2 Training Speedup

8×A100 GPUs, OpenWebText dataset



3× faster

than HuggingFace

Exact same perplexity

Same model quality, dramatically faster training

FlashAttention also enables longer context lengths to be trained effectively

Longer Context = Better Models

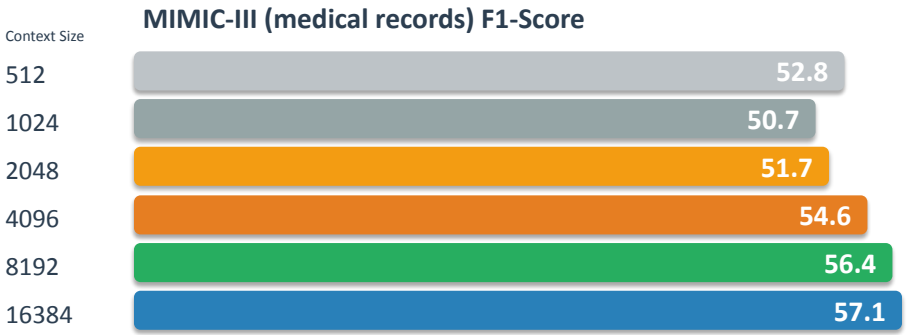
GPT-2 Small: Longer context, better perplexity

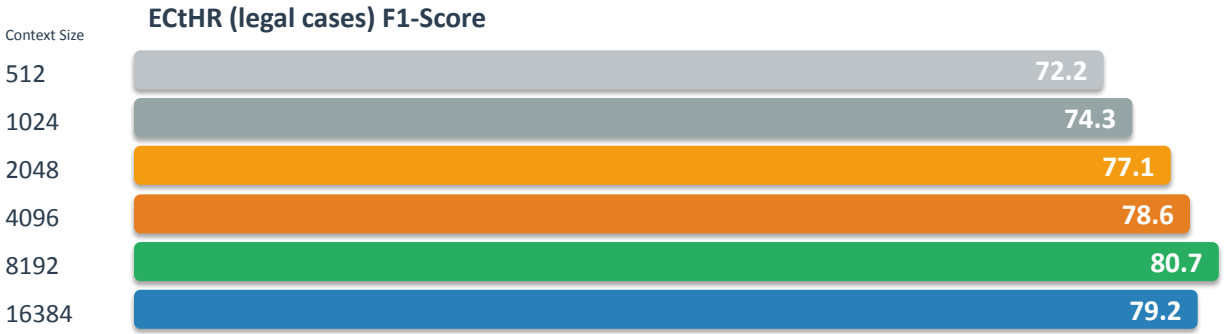
Implementation	Context	Perplexity	Speed
Megatron-LM	1K	18.2	4.7 days
FlashAttention	1K	18.2	2.7 days
FlashAttention	2K	17.6	3.0 days
FlashAttention	4K	**17.5**	3.6 days

4× longer context, 0.7 better perplexity, still 30% faster

Longer Context = Better Models

Long Document Classification Experiments

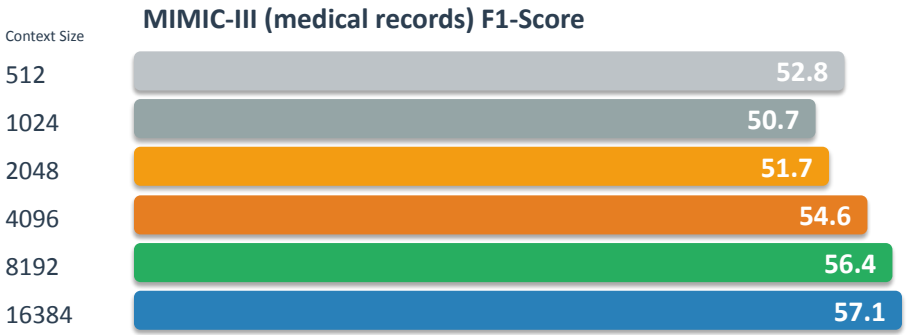


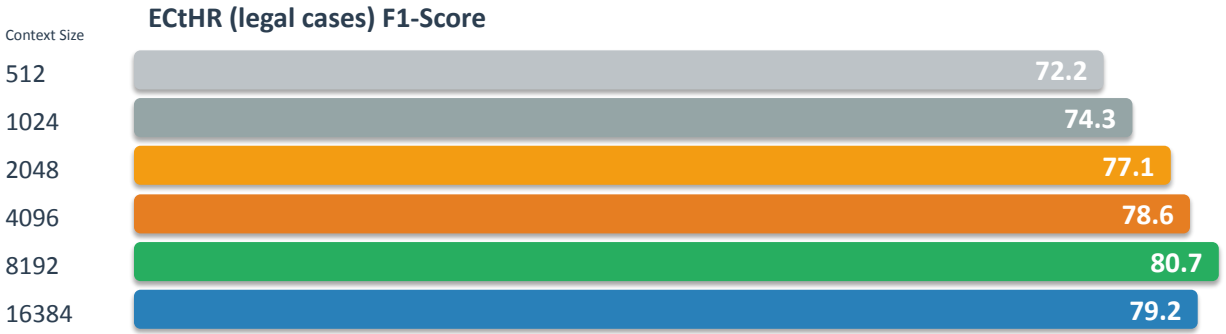


Longer Context models perform better on Long Document Classification

Longer Context = Better Models

Long Document Classification Experiments



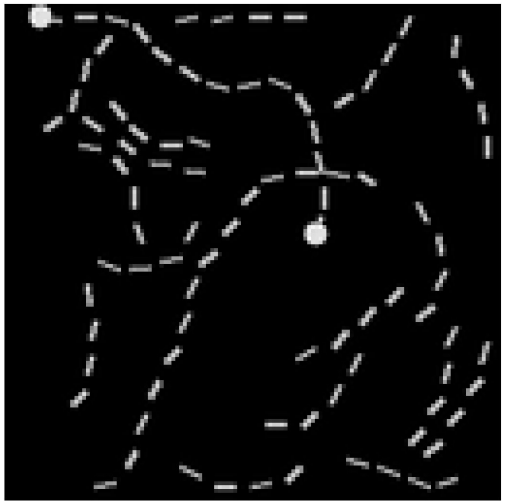


Experiment Design: Truncate document for smaller context windows.

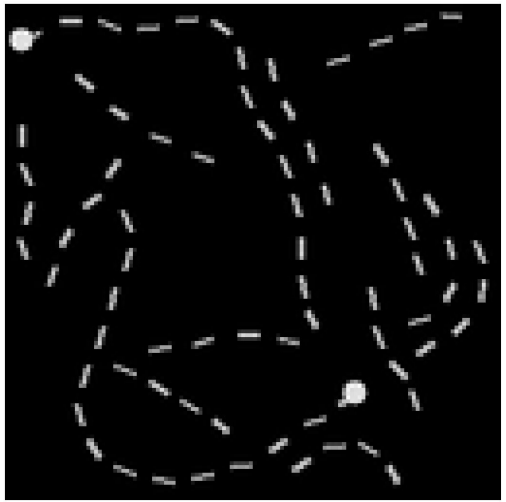
Is this valid?

Path-X and Path-256 Benchmarks

Test LLM ability to find long range spatial dependencies



Positive Example



Negative Example

Long Range Arena: A Benchmark for Efficient Transformers (Tay et. al, 2020)

First Transformer to Solve Path-X

Sequence length 16K — all prior Transformers failed

Model	Path-X (16K)	Path-256 (64K)
Transformer	✗ random	✗ random
Linformer	✗ random	✗ random
Linear Attention	✗ random	✗ random
Performer	✗ random	✗ random
Local Attention	✗ random	✗ random
Reformer	✗ random	✗ random
SMYRF	✗ random	✗ random
FlashAttention	✓ 61.4%	✗ random
Block-Sparse FlashAttention	✓ 56.0%	✓ 63.1%

Entirely new capability: longer context unlocks tasks no prior Transformer could solve

First Transformer to Solve Path-X

Model	Path-X (16K)	Path-256 (64K)
Transformer	✗ random	✗ random
Linformer	✗ random	✗ random
Linear Attention	✗ random	✗ random
Performer	✗ random	✗ random
Local Attention	✗ random	✗ random
Reformer	✗ random	✗ random
SMYRF	✗ random	✗ random
FlashAttention	✓ 61.4%	✗ random
Block-Sparse FlashAttention	✓ 56.0%	✓ 63.1%

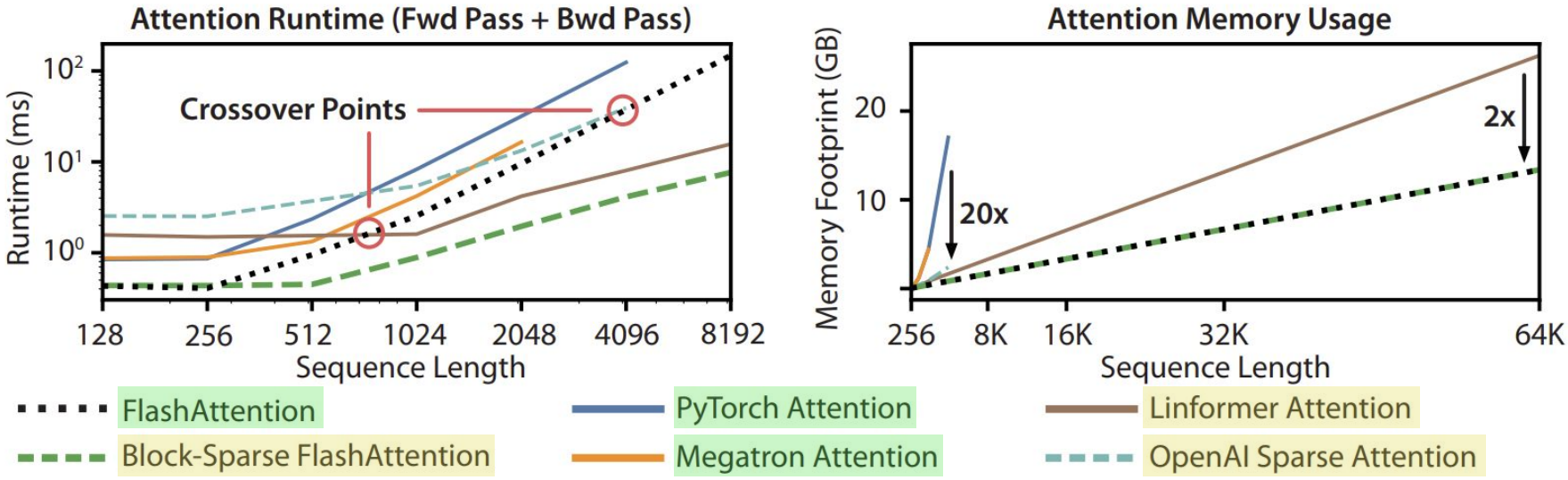
Trains to 64k input tokens, but still fails

Unclear why this is random

FlashAttention performs better than Block-Sparse FlashAttention because it is exact

Training Runtime and Memory

Comparing runtime and memory footprint of multiple implementations across different sequence lengths



Limitations and Next Steps

Limitations

Requires CUDA

Hand-written GPU kernels
Not portable across
architectures

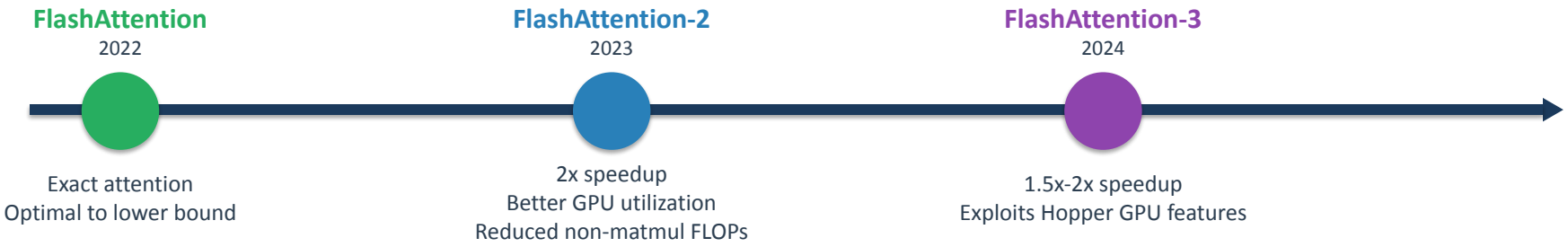
Single-GPU Optimal

Multi-GPU adds another
layer of IO analysis
(inter-GPU transfers)

Head Dim Constraint

Less speedup at d=128
vs d=64 (larger blocks
need more SRAM)

Next Steps and Extensions



FLOPs aren't the whole picture. IO and GPU awareness is equally, if not more, important

Thank You for Listening

Any Questions?