EE-508: Hardware Foundations for Machine Learning

Transformers – Part 3

University of Southern California

Ming Hsieh Department of Electrical and Computer Engineering

Instructors:
Arash Saifhashemi

Flash Attention

Sveral slides and images taken from https://tridao.me

Motivation: Why Model Longer Sequences Efficiently?

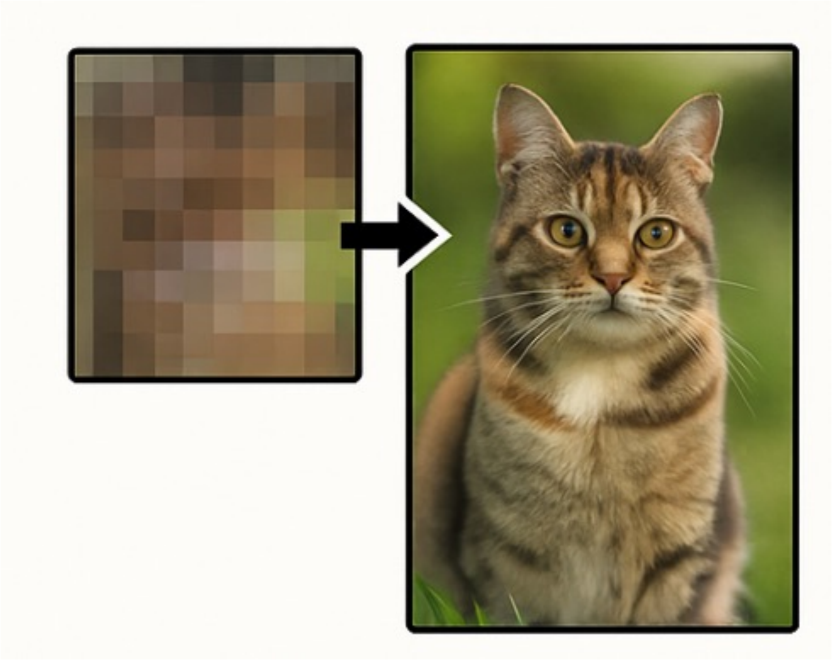
- **Unlocking Advanced Capabilities**
  - **Natural Language Processing (NLP):** Understanding long documents like books or manuals requires maintaining extended context across thousands of tokens.

"Once upon a time..." → ... → "And so, justice was served."

Understanding a story requires remembering distant context

Motivation: Why Model Longer Sequences Efficiently?

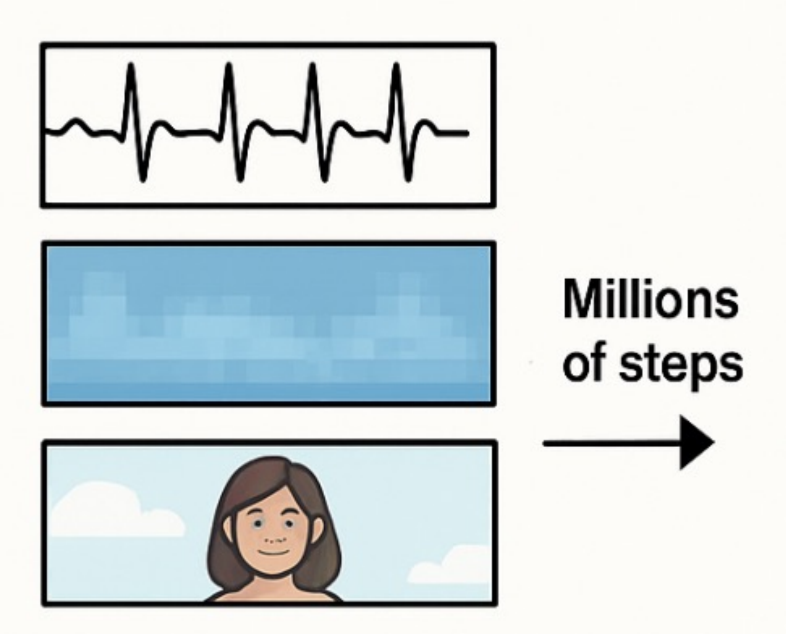
- **Enhancing Perceptual Fidelity**
  - **Computer Vision:** Modeling full-resolution images or videos enables richer, more accurate scene interpretation—especially with long spatial-temporal sequences



High-res input improves detail extraction and decision-making.

Motivation: Why Model Longer Sequences Efficiently?

- **Enabling Emerging Applications**
  - **Time-series, Audio, Video, Medical Imaging:** These domains produce massive sequences that benefit from efficient attention mechanisms—spanning millions of steps.



From top to bottom: ECG waveform, spectrogram, video frames

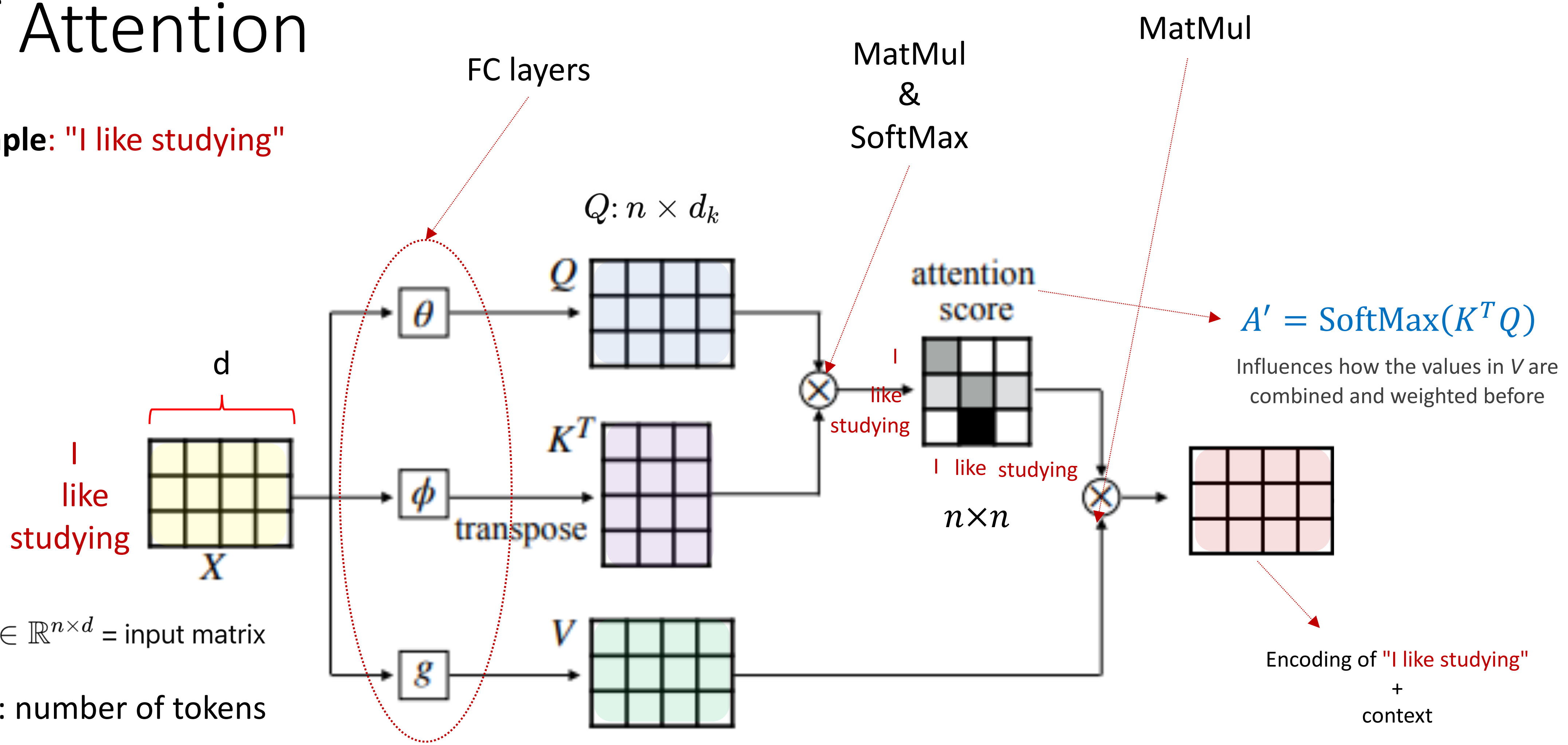
Sequential data across domains demands scalable attention.

Challenge: Scaling Transformers to Long Sequences is Costly

- **Quadratic Attention Complexity**
  - Standard self-attention computes attention scores for every pair of tokens.
  - **Complexity:** $\mathcal{O}(n^2 \cdot d)$
- **Memory Bottleneck**
  - Attention matrix size: n×n
  - High memory use limits batch size and sequence length in practice.
- **Slow Inference and Training**
  - Every token attends to all others → computation doesn't scale linearly.
  - Especially problematic for sequences n>2,048

Self Attention

Example: "I like studying"



$X \in \mathbb{R}^{n \times d}$ = input matrix

n: number of tokens

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

This requires:

- A lot of read and write from memory
- A high number of calculations

What is Dropout in Attention?

- **Dropout** is a regularization method that randomly disables parts of the network during training to prevent overfitting.
  - This forces the network to not rely too heavily on any specific connection.
  - At inference time, all elements are used (scaled appropriately).
- **In Attention Mechanism**
  - Dropout is applied **after softmax** on the attention matrix:
  - Some attention weights are randomly **set to zero**.
  - Forces the model to **distribute attention** more evenly.

$$A' = \mathrm{Dropout}(A) = A \odot M$$

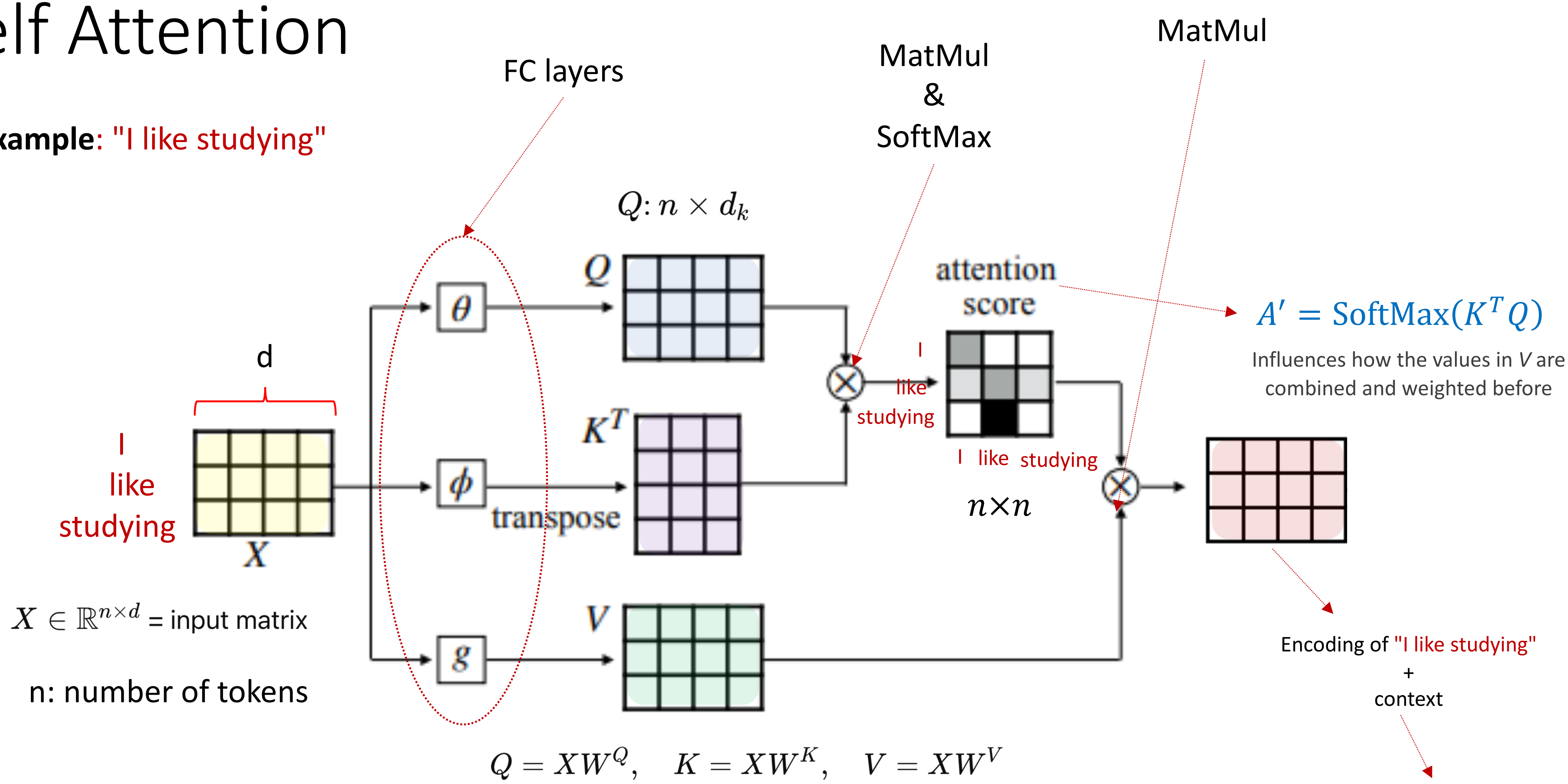
$M \sim \mathrm{Bernoulli}(p)$: a binary mask (1 with probability p, else 0)

$\odot$: element-wise multiplication

$$\mathbf{O} = \mathrm{Dropout}\left(\mathrm{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\right)\mathbf{V}$$

Self Attention

Example: "I like studying"



$$\mathbf{O} = \text{Dropout}\left(\text{Softmax}\left(\text{Mask}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\right)\right)\mathbf{V}$$

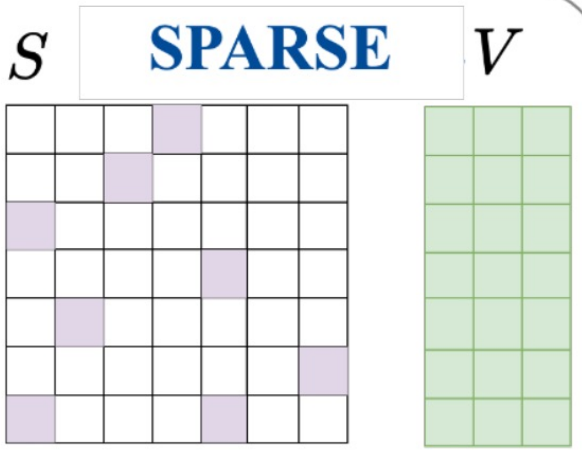
This requires:
- A lot of read and write from memory
- A high number of calculations

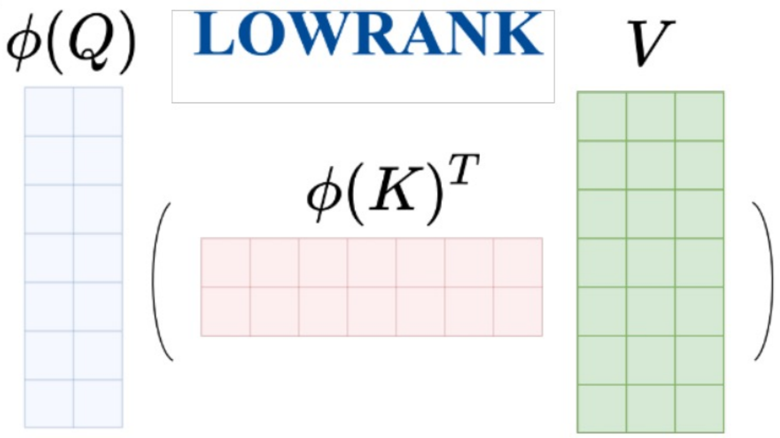
Challenge: Scaling Transformers to Long Sequences is Costly

- **Quadratic Attention Complexity**
  - Standard self-attention computes attention scores for every pair of tokens.
  - **Complexity:** $\mathcal{O}(n^2 \cdot d)$
- **Memory Bottleneck**
  - Attention matrix size: n×n
  - High memory use limits batch size and sequence length in practice.
- **Slow Inference and Training**
  - Every token attends to all others → computation doesn't scale linearly.
  - Especially problematic for sequences n>2,048

Background: Approximate Attention

Sparse Transformer
(Child et al. 19)
Reformer
(Kitaev et al. 20)
Routing Transformer
(Roy et al. 20)





Linformer
(Wang et al. 20)
Linear Transformer
(Katharopoulos et al. 20)
Performer
(Choromanski et al. 20)

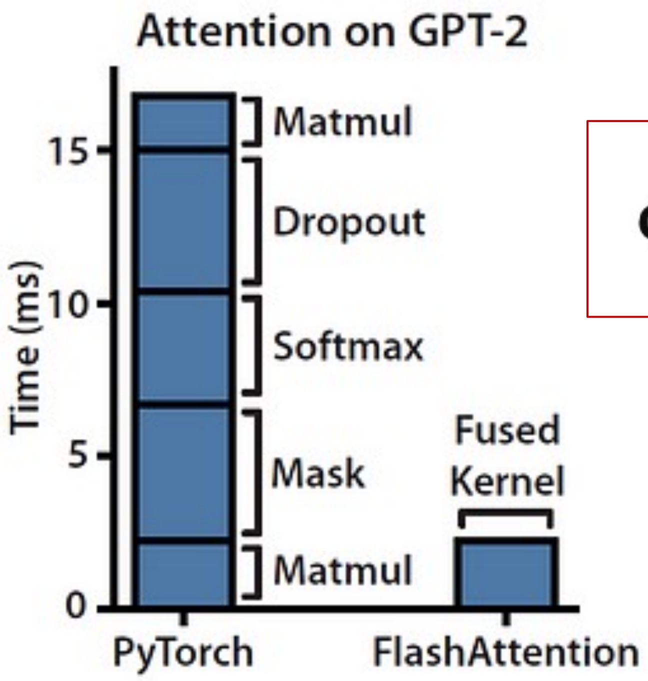
Approximate attention: tradeoff quality for ~~speed~~ fewer FLOPs

Survey: Tay et al. Long Range Arena : A Benchmark for Efficient Transformers. ICLR 2020.

Flash Attention

Is there a fast, memory-efficient, and exact attention algorithm?

Flash Attention

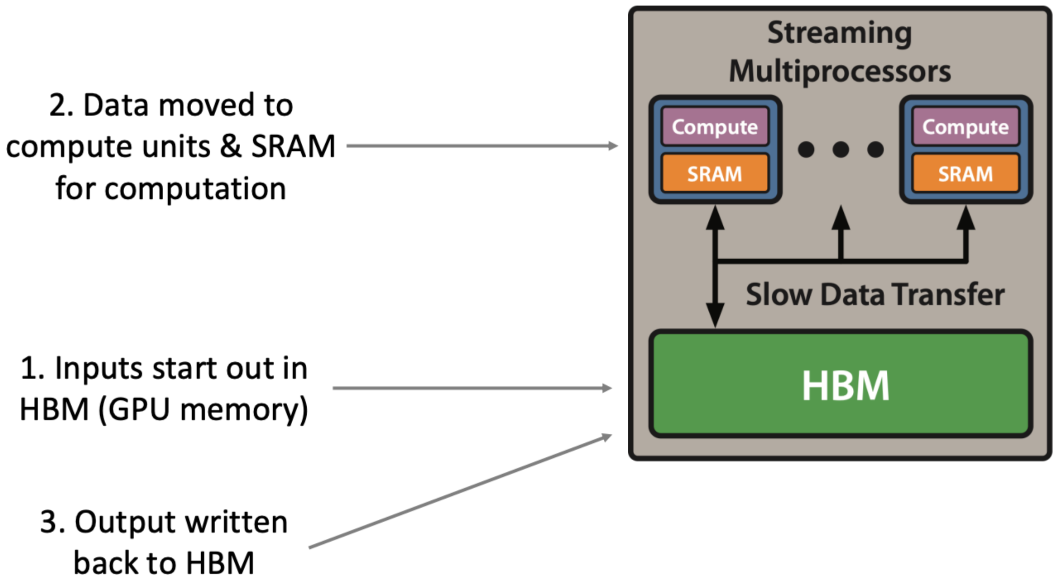


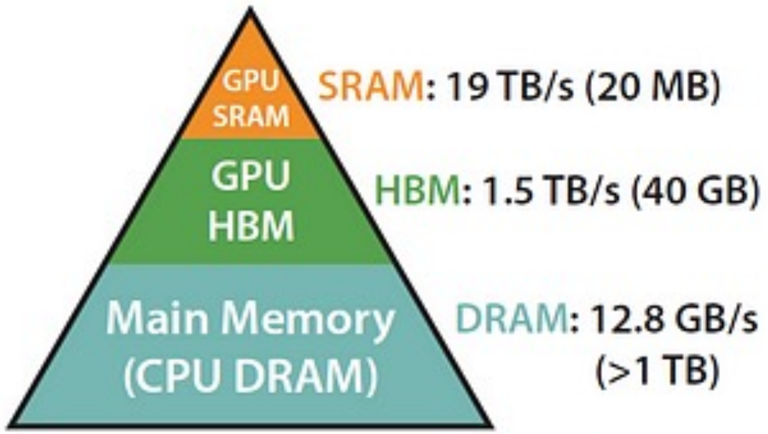
$$\mathbf{O} = \text{Dropout}\left(\text{Softmax}\left(\text{Mask}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\right)\right)\mathbf{V}$$

- **Performance Comparison**
  - **Standard PyTorch**: **(GPT-2)** Performs 4 separate memory-heavy operations.
  - **FlashAttention**: Fuses them into one kernel.
  - **Result**: Up to **5x speedup** on long sequences.

Note that Matmul does not consume too much time

Background: GPU Memory Model



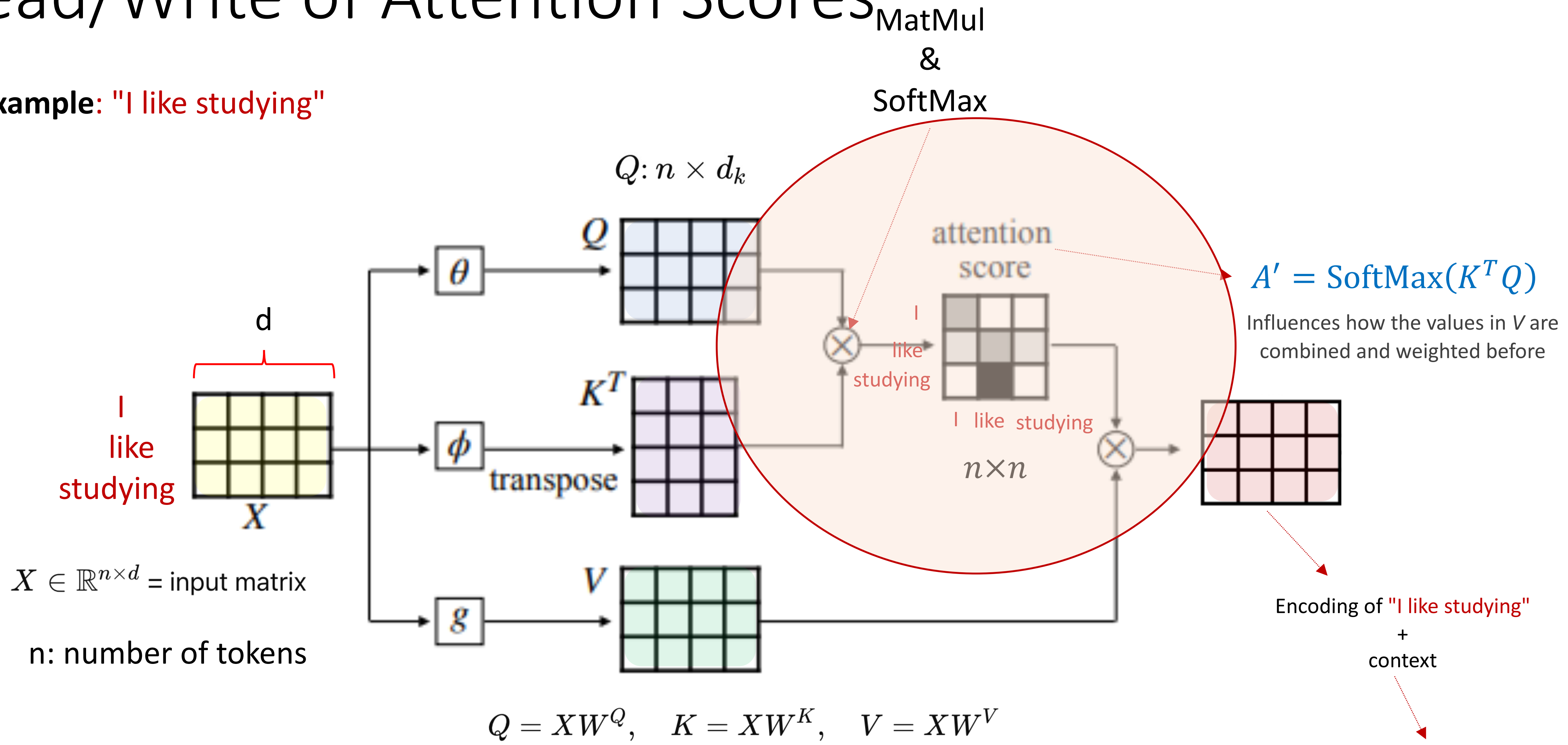


Memory Hierarchy with Bandwidth & Memory Size

The biggest cost is in moving data
Standard implementation requires repeated R/W from slow GPU memory

Read/Write of Attention Scores

Example: "I like studying"



$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

This requires:

- A lot of read and write from memory
- A high number of calculations

$$\mathbf{O} = \text{Dropout}\left(\text{Softmax}\left(\text{Mask}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\right)\right)\mathbf{V}$$

Exactly How Are We Saving?

- We want to compute:

$$\mathbf{O} = \mathrm{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$$

Traditional implementation (naive):

1. Compute **QKᵀ** → this is a full attention score matrix.
2. Store it in memory (often **written to DRAM**).
3. Apply softmax.
4. Multiply by V.

The attention matrix $\mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}$ is never materialized (i.e. written back into HBM)

Feature	Standard Attention	FlashAttention	Savings
Attention matrix	Materialized in HBM	Never stored	**Biggest win**
K/V access	Multiple reads (backward, dropout)	Streamed once	Fewer HBM reads
Intermediate writes	Heavy	None (in SRAM only)	Saves bandwidth
Memory usage	O(N^2)	O(N)	Scalability

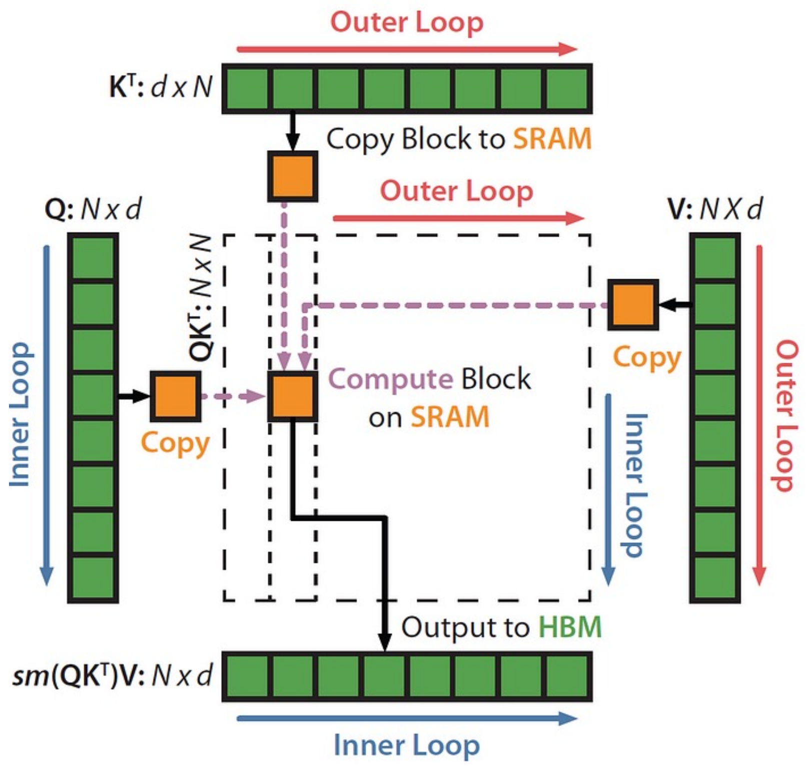
Why Write QK^T in Traditional Attention?

1. **Modular Computation**:
   1. Traditional frameworks (like PyTorch, TensorFlow) compute QK^T → softmax → multiply by V as **separate steps**.
   2. Intermediate result (QK^T) must be **stored temporarily** to proceed to the next stage.
2. **Memory Constraints**:
   1. QK^T is a large **n × n** matrix.
   2. Too big to keep entirely in fast on-chip memory (registers/shared memory).
   3. So it's **offloaded to slower global memory** (i.e., written to DRAM or GPU memory).
3. **Backward Pass (Training)**:
   1. QK^T may be needed for **gradients during backpropagation**, so it's stored to avoid recomputation.

FlashAttention fuses these steps to avoid the write entirely.

FlashAttention: Block-wise Attention in SRAM

- **Goal:**
  - **Efficiently compute attention** without materializing the full attention matrix or overloading HBM.
- **Looping Mechanism**
  - **Outer Loop (Red)**: Iterates over **K/V blocks**
  - **Inner Loop (Blue)**: Streams over **Q rows**



Step	Description
1. Copy	Load blocks of Q, K, V from HBM → SRAM
2. Compute	Compute $\mathbf{QK}^\top$, softmax, and $\mathrm{softmax}(\mathbf{QK}^\top)\mathbf{V}$ inside fast SRAM
3. Accumulate	Combine block-wise softmax outputs using scaled sums
4. Output	Final result $\mathbf{O} = \mathrm{Attention}(Q, K, V)$ streamed back to HBM

Green: HBM
Orange: SRAM

- Avoids writing the full attention matrix to memory.
- Keeps **all intermediate ops in SRAM**.
- Enables attention on very **long sequences** with high speed and low memory usage.

How to Reduce HBM Reads/Writes: Compute by Blocks

Challenges:

- **Softmax Normalization Without Full Input**
  - Softmax requires access to the **entire row** to normalize values (i.e., compute exponentials and divide by the sum).
  - When processing in **blocks**, you don't have the whole row at once.
- **Backward Pass Without Storing Attention Matrix**
  - The standard method stores the full attention matrix from the forward pass for use during backpropagation.
  - But that matrix is **too large** to keep in SRAM or even HBM for long sequences.

Approaches

1.Tiling

1. Instead of computing attention all at once, **split it into blocks** (tiles).
2. Only load **one tile at a time** into fast GPU **SRAM**.
3. Perform computations (QK^T, softmax, etc.) block-by-block, reducing HBM traffic.

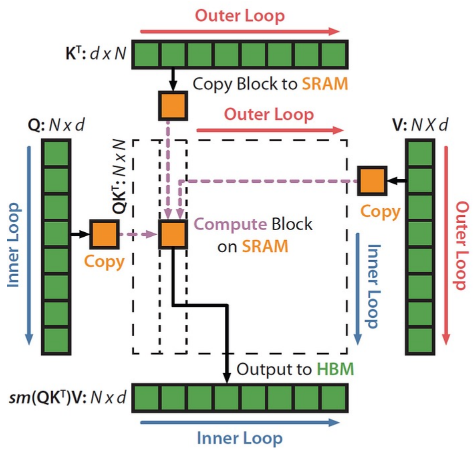
2.Recomputation

1. Instead of storing the huge attention matrix, **recompute it during the backward pass**.
2. This saves memory (important for long sequences) at the cost of **a bit more compute**.

FlashAttention smartly uses **tiling + recomputation** to:

- Minimize HBM reads/writes
- Fit large attention computations into fast memory
- Enable scaling to **very long sequences**

Key Insight Behind FlashAttention



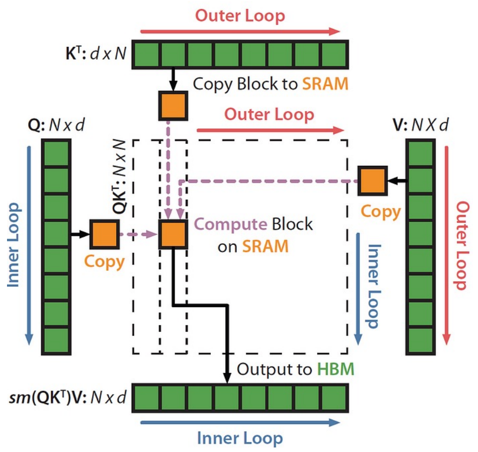
- **Decomposing Large Softmax via Blocked Scaling**
  - We can break softmax over a large matrix into smaller blocks with scaled softmax:

$$\mathrm{softmax}([A_1, A_2]) = [\alpha\, \mathrm{softmax}(A_1),\ \beta\, \mathrm{softmax}(A_2)]$$

$$\alpha = \frac{e^{m_1}}{e^{m_1} + e^{m_2}}, \quad \beta = \frac{e^{m_2}}{e^{m_1} + e^{m_2}}, \quad m_1 = \max A_1,\ m_2 = \max A_2$$

Key Insight Behind FlashAttention

- **Applied to Attention**
  - Each softmax is computed **locally** per block.
  - Normalization is done **globally via scaling**.
  - Enables **block-wise computation** that fits in **SRAM**, minimizing memory traffic.

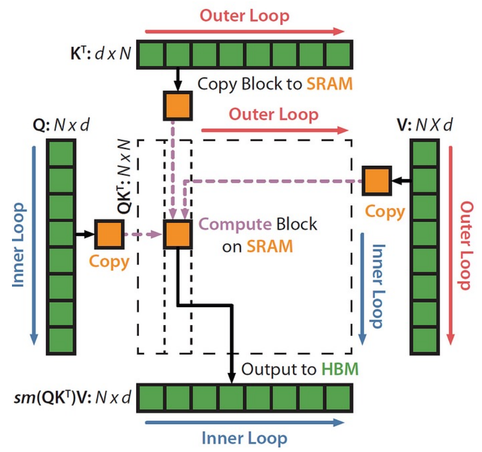


$$\mathrm{softmax}([A_1, A_2]) = [\alpha\, \mathrm{softmax}(A_1),\ \beta\, \mathrm{softmax}(A_2)]$$

$$\alpha = \frac{e^{m_1}}{e^{m_1} + e^{m_2}}, \quad \beta = \frac{e^{m_2}}{e^{m_1} + e^{m_2}}, \quad m_1 = \max A_1,\ m_2 = \max A_2$$

$$\mathrm{softmax}([A_1, A_2]) \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha\, \mathrm{softmax}(A_1) V_1 + \beta\, \mathrm{softmax}(A_2) V_2$$

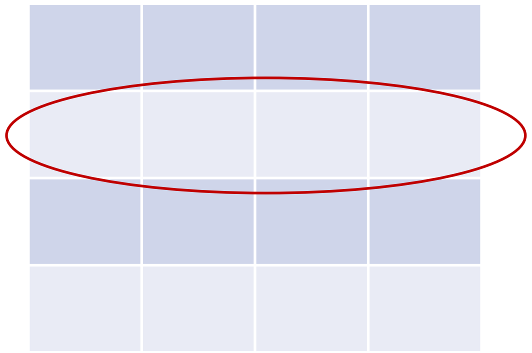
Tiling Softmax



$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

l: denominator

row i

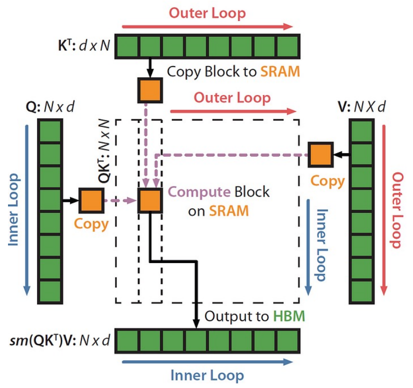


The denominator for row i is:

$$l_i = \sum_j \exp(A_{ij})$$

Tiling + Softmax Rescaling

- **Step 1: Compute Scores Block-by-Block**
  - Partition K to $K^{(1)}, K^{(2)}$
  - Keys $K^{(1)}, K^{(2)}$ are stored in HBM.
  - Compute score matrices (in SRAM):

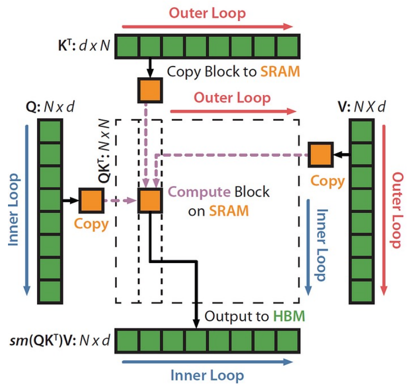


$$\mathrm{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

l: denominator

$$S^{(1)} = Q(K^{(1)})^\top, \quad S^{(2)} = Q(K^{(2)})^\top$$

Tiling + Softmax Rescaling



- **Step 1: Compute Scores Block-by-Block**
  - Partition K to $K^{(1)}, K^{(2)}$
  - Keys $K^{(1)}, K^{(2)}$ are stored in HBM.
  - Compute score matrices (in SRAM):

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

l: denominator

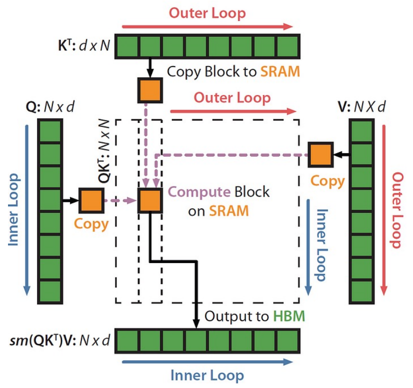
$$S^{(1)} = Q(K^{(1)})^\top, \quad S^{(2)} = Q(K^{(2)})^\top$$

- **Step 2: Apply Softmax Locally**
  - Accumulate denominator terms:

$$A^{(1)} = \exp(S^{(1)}), \quad A^{(2)} = \exp(S^{(2)})$$

$$l^{(1)} = \sum_i \exp(S_i^{(1)}), \quad l^{(2)} = l^{(1)} + \sum_i \exp(S_i^{(2)})$$

Tiling + Softmax Rescaling



- **Step 1: Compute Scores Block-by-Block**
  - Partition K to $K^{(1)}, K^{(2)}$
  - Keys $K^{(1)}, K^{(2)}$ are stored in HBM.
  - Compute score matrices (in SRAM):

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

l: denominator

$$S^{(1)} = Q(K^{(1)})^\top, \quad S^{(2)} = Q(K^{(2)})^\top$$

- **Step 2: Apply Softmax Locally**
  - Accumulate denominator terms:

$$A^{(1)} = \exp(S^{(1)}), \quad A^{(2)} = \exp(S^{(2)})$$

$$l^{(1)} = \sum_i \exp(S_i^{(1)}), \quad l^{(2)} = l^{(1)} + \sum_i \exp(S_i^{(2)})$$

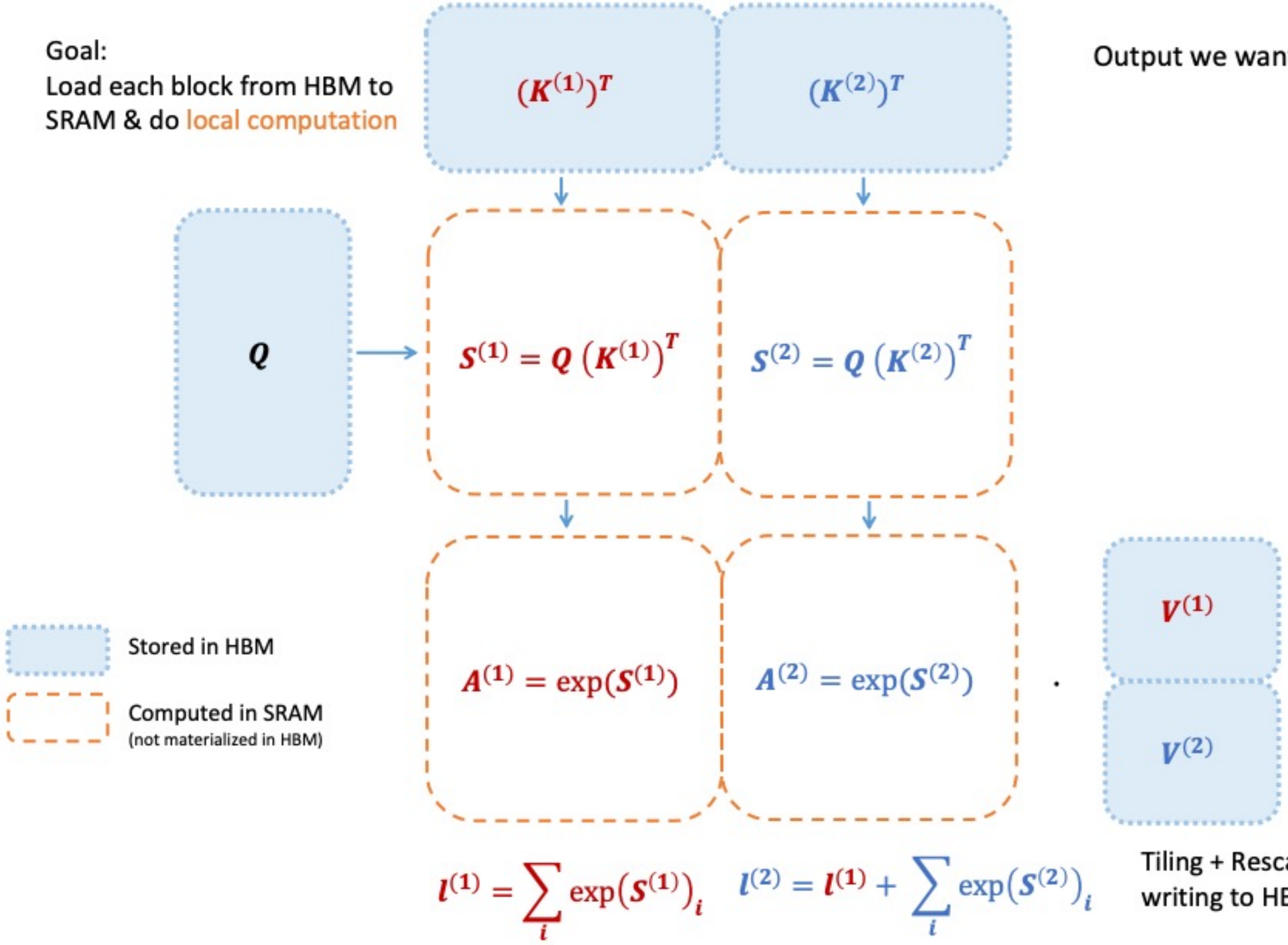
- Step 2: Compute Partial Outputs
  - Locally compute:

$$O^{(1)} = \frac{A^{(1)}}{l^{(1)}} V^{(1)}$$

  - Rescale using global denominator:

$$O = \frac{l^{(1)}}{l^{(2)}} O^{(1)} + \frac{A^{(2)}}{l^{(2)}} V^{(2)}$$

Tiling + Softmax Rescaling (Visual Representation)



Output we want: $l = \sum_i \exp\left(S^{(1)}\right)_i + \sum_i \exp\left(S^{2}\right)_i$

$$O = \frac{A^{(1)}}{l} \cdot V^{(1)} + \frac{A^{(2)}}{l} \cdot V^{(2)}$$

$$O^{(1)} = \frac{A^{(1)}}{l^{(1)}} V^{(1)} \quad \text{(local partial output)}$$

$$O = \frac{l^{(1)}}{l^{(2)}} O^{(1)} + \frac{A^{(2)}}{l^{(2)}} V^{(2)}$$

Tiling + Rescaling allows local computation in SRAM, without writing to HBM, and get the right answer!

Tiling + Softmax Rescaling (Summary)

1. **Compute local scores:**

$$S^{(1)} = Q(K^{(1)})^\top, \quad S^{(2)} = Q(K^{(2)})^\top$$

2. **Apply local softmax approximation:**

$$A^{(1)} = \exp(S^{(1)}), \quad A^{(2)} = \exp(S^{(2)})$$

3. **Accumulate denominators:**

$$l^{(1)} = \sum \exp(S^{(1)}), \quad l^{(2)} = l^{(1)} + \sum \exp(S^{(2)})$$

4. **Compute and rescale outputs:**

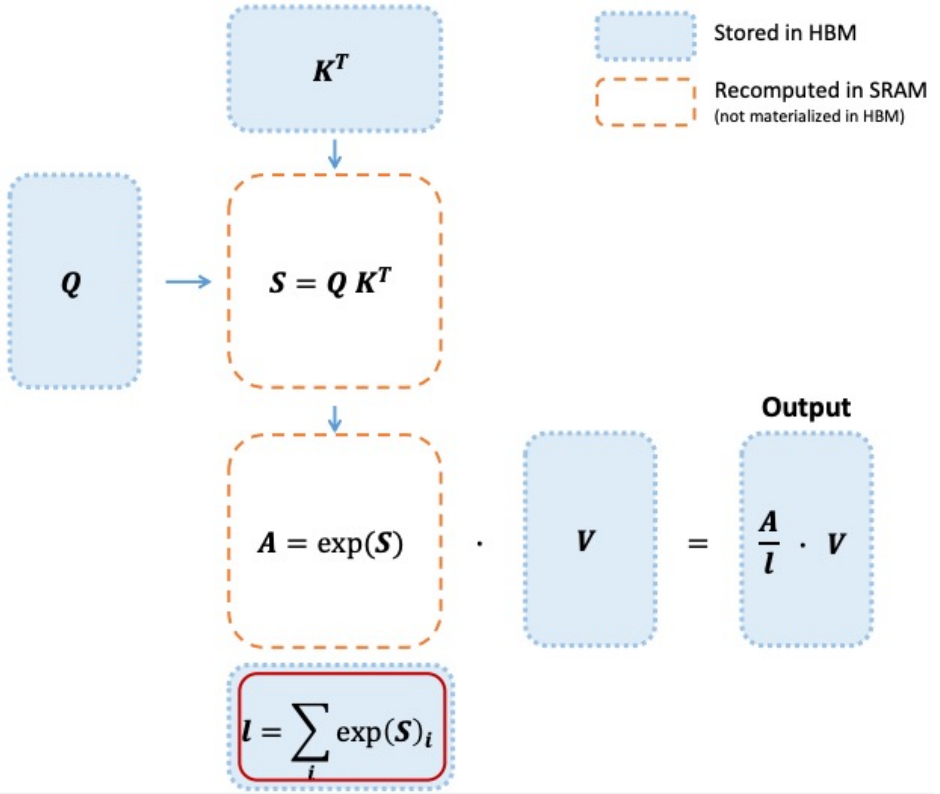
$$O^{(1)} = \frac{A^{(1)}}{l^{(1)}} V^{(1)} \qquad O = \frac{l^{(1)}}{l^{(2)}} O^{(1)} + \frac{A^{(2)}}{l^{(2)}} V^{(2)}$$

- This fixes the softmax to act **as if it was applied over the full $K^{(1)}, K^{(2)}$,** even though we processed one block at a time.
  - Eliminates full attention matrix storage
  - Enables **efficient long-sequence attention**

Recomputation in Backward Pass (FlashAttention)

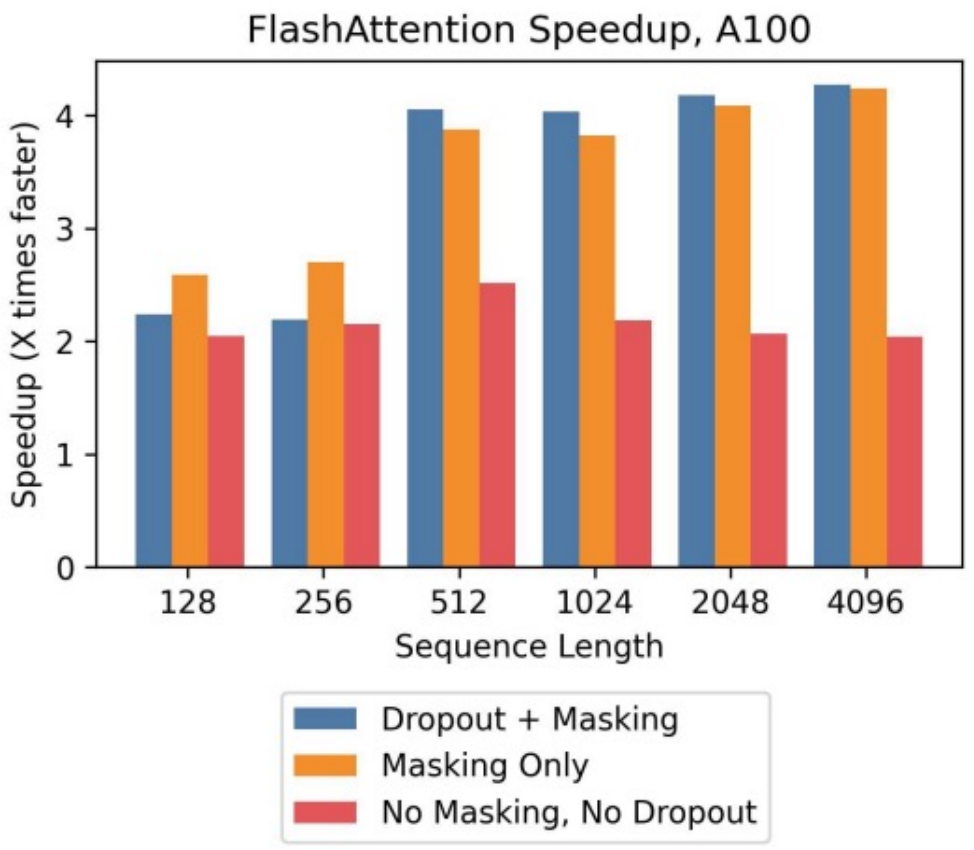
- **What's the problem in standard attention?**
  - During backpropagation, we need the attention again.
  - In standard attention, this large N×N matrix is **stored in HBM**, which is costly in memory.
- **What does FlashAttention do instead?**
  - Instead, it **recomputes** it during the backward pass from:
    - **Do not store** full A
    - **Only store** normalization vector l
    - Recompute A again (same Q and K, same result).
    - Since we stored **l** (normalizer for softmax), we can exactly **reconstruct softmax(QK^T)** in blocks.
    - Use this recomputed A to compute gradients.

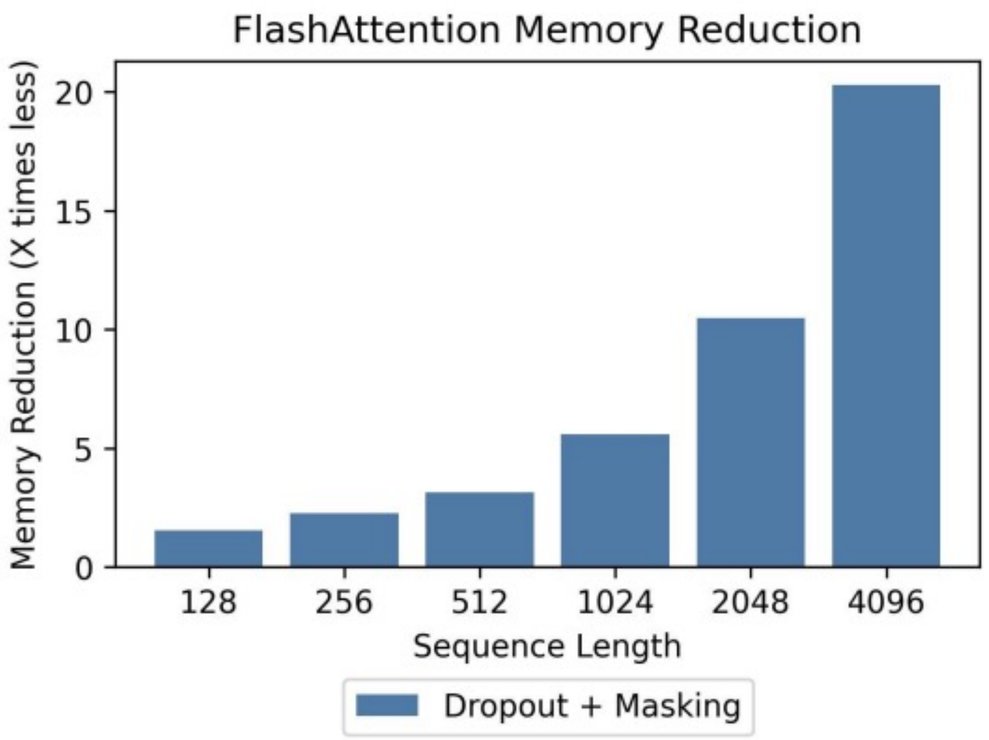
FlashAttention trades a small compute increase for **massive memory savings and speedup**.



Attention	Standard	FlashAttention
GFLOPs	66.6	75.2 (↑13%)
HBM reads/writes (GB)	40.3	4.4 (↓9x)
Runtime (ms)	41.7	7.3 (↓6x)

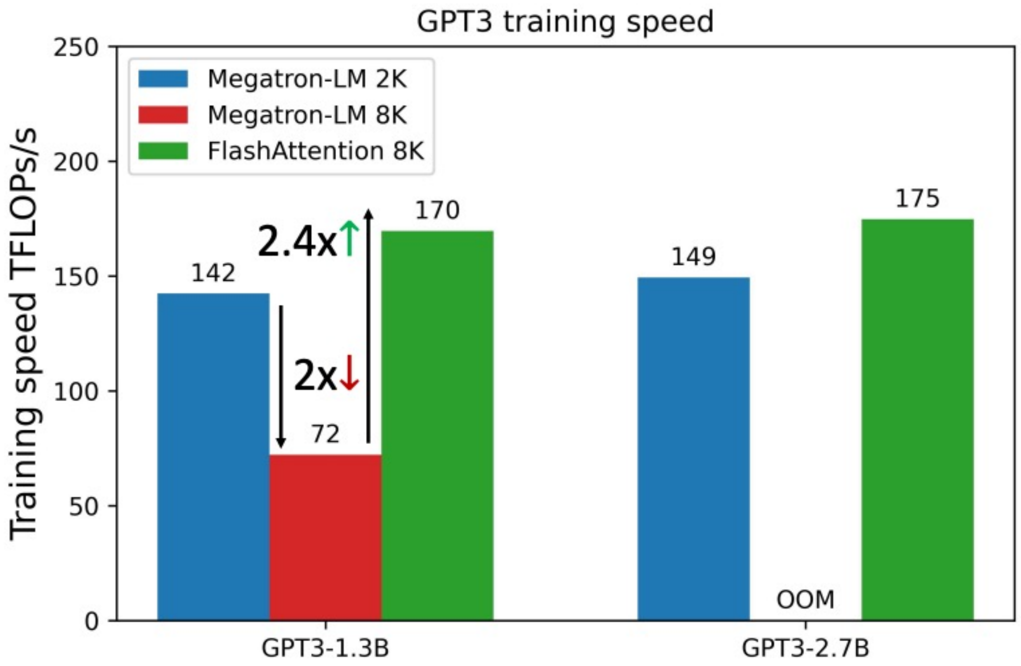
FlashAttention: 2-4x speedup, 10-20x memory reduction



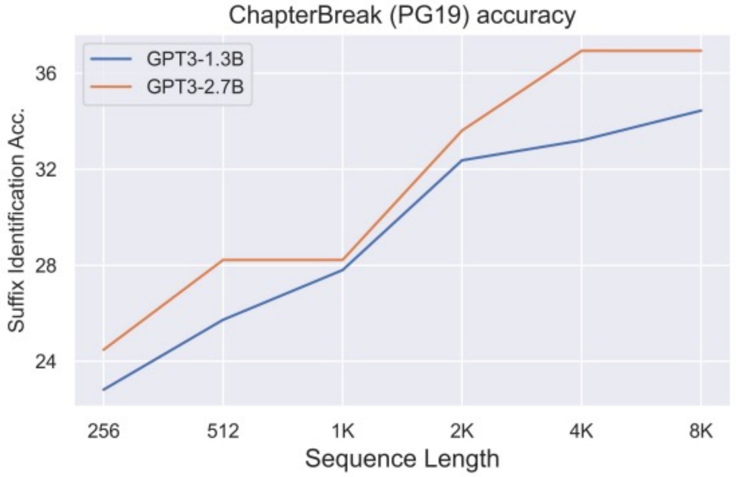


- 2-4x speedup — with no approximation
- 10-20x memory reduction — memory linear in sequence length

GPT3: Faster Training, Longer Context, Better Model



Model	Val perplexity on the Pile (lower better)
GPT-1.3B, 2K context	5.45
GPT-1.3B, 8K context	**5.24**
GPT-2.7B, 2K context	5.02
GPT-2.7B, 8K context	**4.87**



FlashAttention speeds up GPT-3 training by 2x, increase context length by 4x, improving model quality

- **Longer context** (enabled by FlashAttention) **improves perplexity** and **model understanding**.
- **ChapterBreak** is a benchmark task designed to evaluate a language model's ability to understand and **retain long-range context** — especially over several thousand tokens.
  - Models trained with longer context (e.g., 8K tokens) perform significantly better on ChapterBreak.
    - This shows FlashAttention helps models retain more context, which boosts performance on long-sequence tasks.

FlashAttention-2 — More Speed, Simpler Design

- **FlashAttention (v1):**
  - Tiled attention in GPU SRAM
  - Blockwise softmax with scaling
  - No full attention matrix in HBM
  - **Fused softmax and matmul**, but **dropout and masking were separate kernels**
  - **Better backward pass**: recomputation without storing attention matrix
  - **Faster** and **more memory-efficient**
  - **Enabled 8K+ token context training**

- **FlashAttention-2 (Additions Over FA-1):**
  - **Fully fused CUDA kernel**: softmax + dropout + masking + matmul
  - **Improved parallelism**: better use across batch, heads, blocks
  - **Results:**
    - **Up to 2× faster** than FA-1
    - **5× faster** than standard attention

FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning
Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, Christopher Ré
arXiv:2307.08691 [cs.LG], July 2023

FlashAttention-3 — Additional Enhancements

- **New Additions Over FA-2:**
  - **Sliding Window & Sparse Attention**: restricts each token to attend only to a local window, improving efficiency while retaining useful context (used in models like Mistral)
  - **Multi-head parallelism** using tensor cores
  - **2-bit softmax** (quantized exponentials)
- **Purpose:**
  - Built for **modern long-sequence LLMs**: Mistral, Yi, DeepSeek, etc.
  - Efficient up to **128K+ tokens**

FlashAttention-3: Faster Attention with Better Parallelism and Memory Efficiency
Tri Dao, Mohammad Shoeybi, Alexander G. Matveev, Dan Fu, Jim Gschwind, Bill Jia, Sharan Chetlur, Shoumik Palkar, Quynh Nguyen, Christopher Ré
arXiv:2402.17764 [cs.LG], February 2024