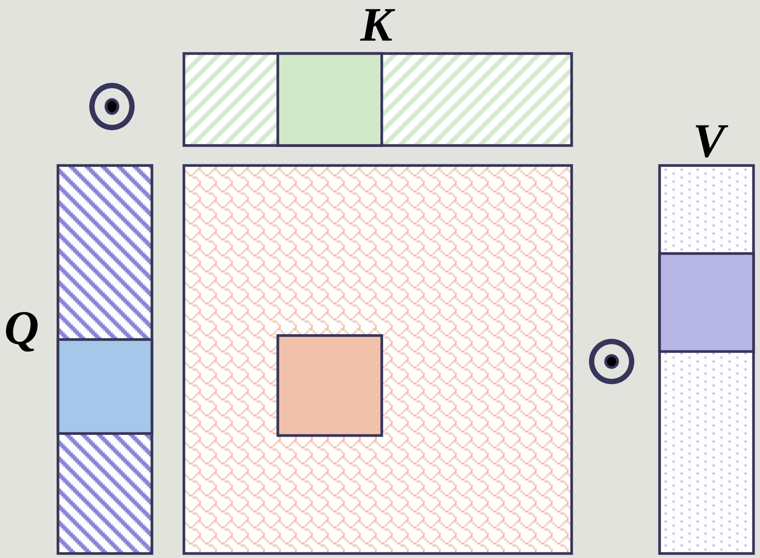
Flash Attention (v1)



Vahid Mirjalili (https://vmirly.github.io)

PyML Studio

Complexity of dot-production attention

- **Original:**
  - Memory $O(N^2)$
  - Computation complexity: $O(N^2)$

➔ Quadratic

Difficulty in increasing the context length

- **Approximate attention methods:** $O(N)$ ➔ Linear

But often they are not able to achieve wall-clock speedup.

Why?

- Reducing FLOP operations, at the cost of memory access overhead

FlashAttention Overview

Key Idea:

Avoid reading and writing the full attention matrix to and from DRAM

- Using a tiling mechanism
- I/O-aware, significantly fewer memory access
- Exact attention

- Wall-clock speedup at training:
  - BERT-Large (seq. length=512) ➔ 15% speedup
  - GPT-2 (seq. length=1k) ➔ 3x speedup

FlashAttention

Key Idea

Avoid reading and writing the full attention matrix to and from DRAM

Challenges

1. Computing the softmax step without accessing the whole $QK^\top$

2. Backward-pass without storing the attention matrix

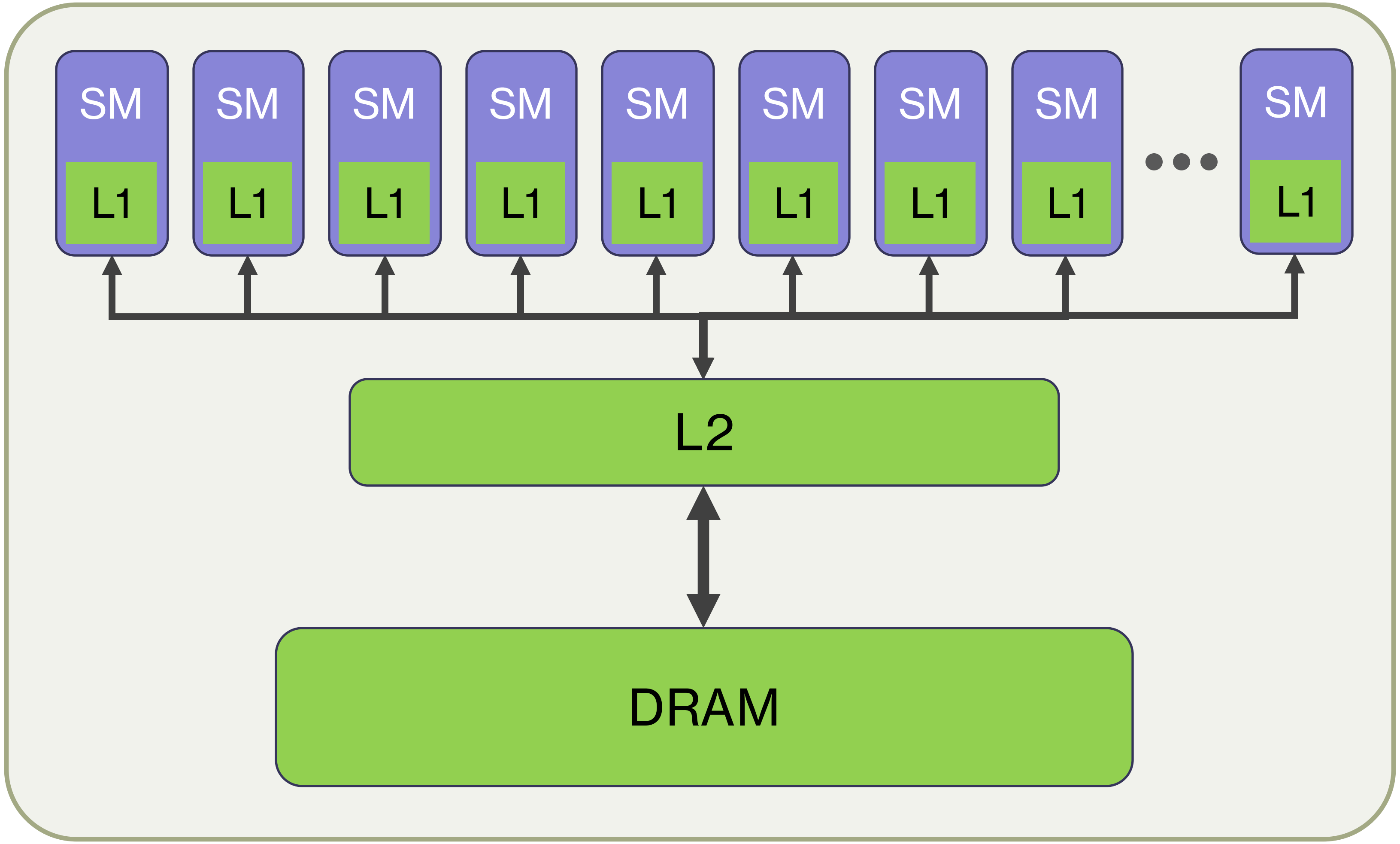
Solutions

- Tiling mechanism

- Only storing the normalization factor of softmax
- Recomputing attention on-chip

Faster than reading the attention matrix from memory

Understanding GPU Basics



NVIDIA A100 TensorCore

108 SMs

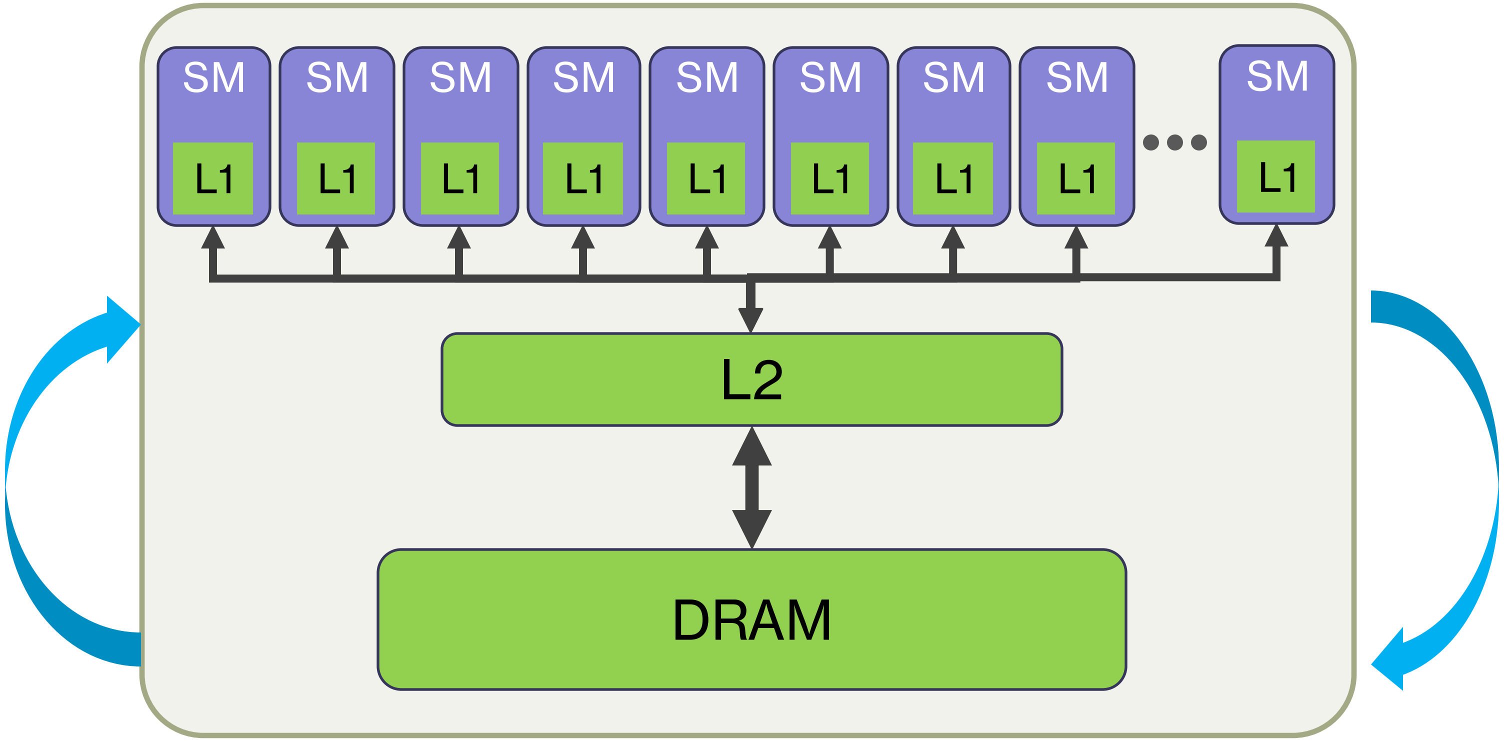
Types of DRAM:
- SDRAM
- GDDR
- HBM

Size: 80GB of HBM2e
Bandwidth: 2039 GB/s

GPU Execution

(2)
Computation at SMs

(1)
Loading data from memory to registers and SRAM



(3)
Writing output back to DRAM

Performance of computing function $y = f(x)$

- Time for memory access: T_{mem}
- Time for mathematical computations: T_{math}

Total time:

$$T_f = \max(T_{mem}, T_{math})$$

If $T_{math} > T_{mem}$ ➔ compute-bound

If $T_{mem} > T_{math}$ ➔ memory-bound

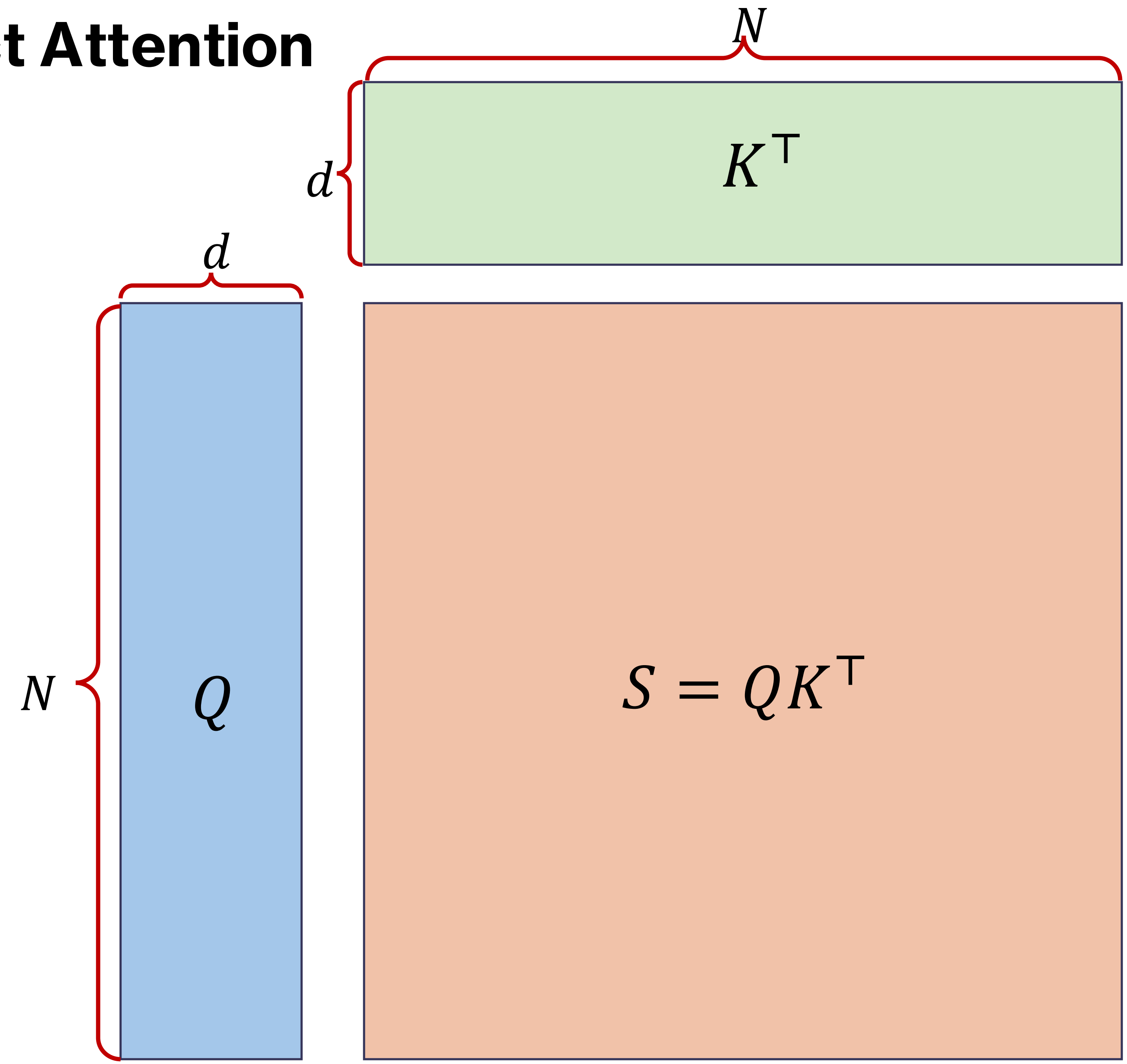
$$BW_{math} = \#\, ops\ per\ second$$

$$BW_{mem} = \#\, bytes\ \ per\ second$$

$$T_{math} = \frac{\#\, ops}{BW_{math}}$$

$$T_{mem} = \frac{\#\, bytes}{BW_{mem}}$$

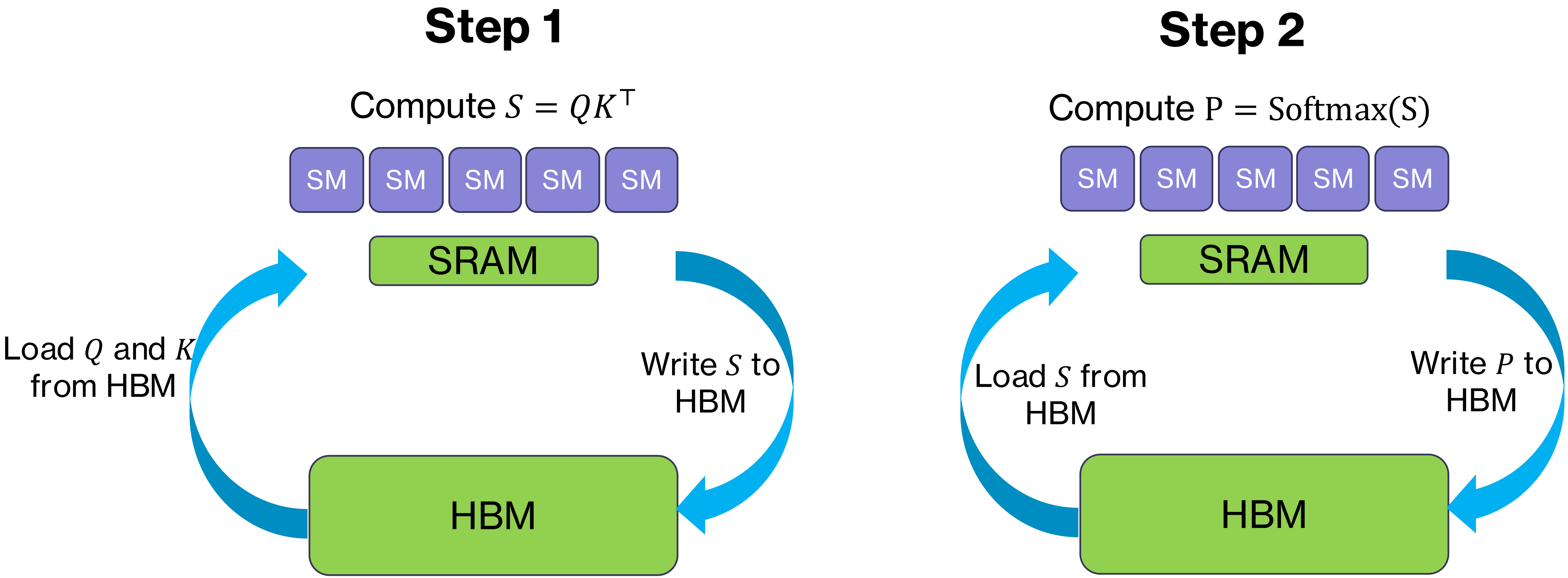
Dot-product Attention



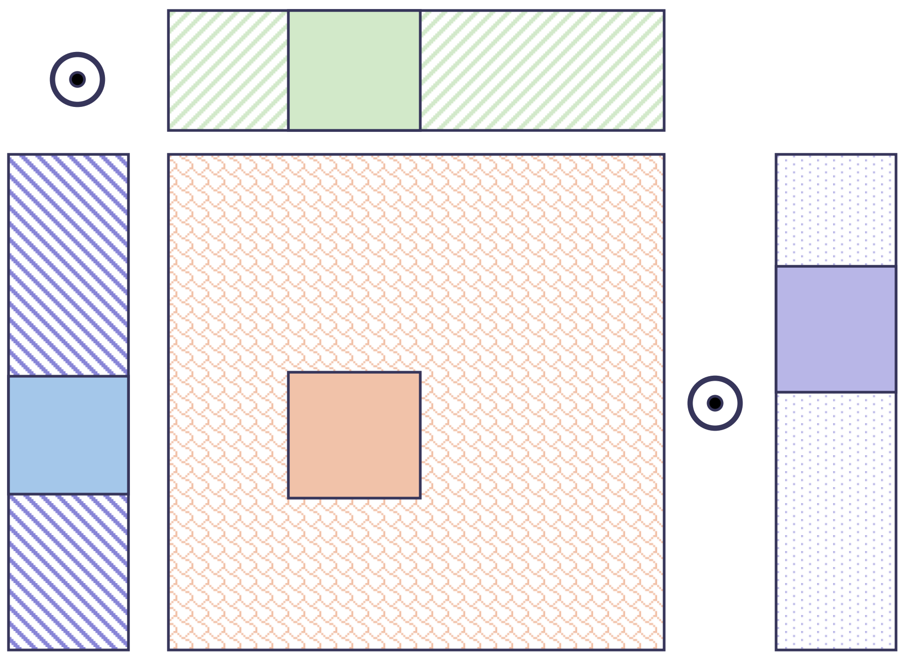
torch.matmul(Q, K.T)

GPT2:
$d = 64$
$N = 1024$

```
S = torch.matmul(Q, K.transpose(-2, -1))
P = torch.softmax(S, dim=-1)
output = torch.matmul(P, V)
```

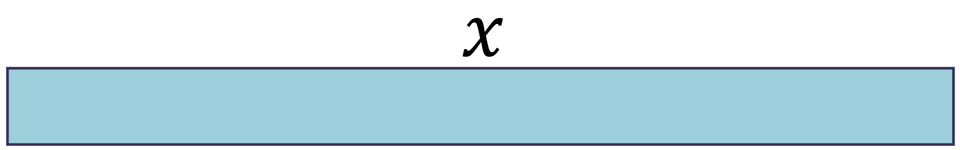


FlashAttention Algorithm



- Tiling (for the forward pass)
  - Loading blocks of Q, K, and V and computing the output partially
- Re-computation (for the backward pass)
  - Using the stored normalization factors for each row

Reformulating Softmax Function



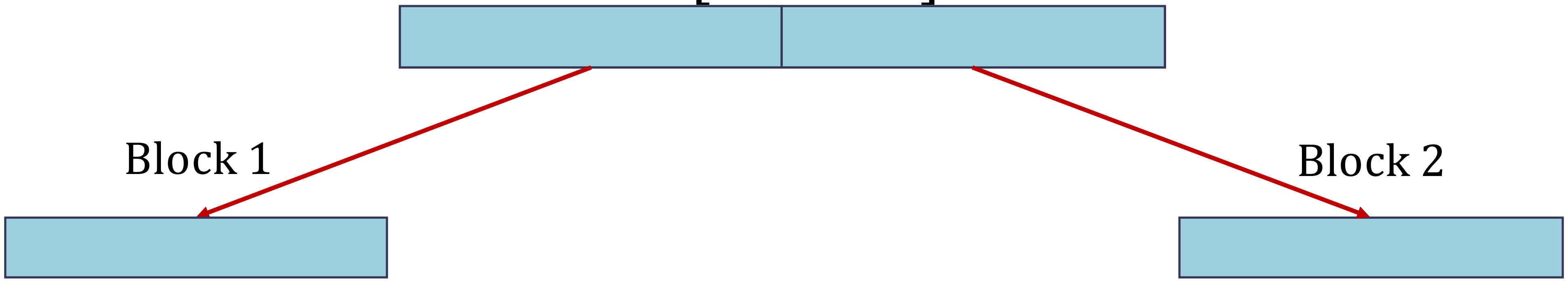
$$\mathrm{Softmax}(x)_i = \frac{\exp(x_i)}{\sum_k \exp(x_k)} \times \frac{\exp(-m)}{\exp(-m)}$$

$$\Rightarrow \quad \frac{\exp(x_i - m)}{\sum_k \exp(x_k - m)}$$

Re-formulating softmax:

$$\left.\begin{aligned} \vec{f}(x) &= \left[e^{x_1-m}, e^{x_2-m}, e^{x_3-m}, \quad \ldots \quad e^{x_N-m}\right] \\ l(x) &= \sum_j f(x)_j \end{aligned}\right\} \quad \mathrm{Softmax}(x) = \frac{\vec{f}(x)}{l(x)}$$

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



Block 1

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

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

$$l_1 = \sum_j e^{x_j^{\{1\}} - m_1}$$

Block 2

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

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

$$l_2 = \sum_j e^{x_j^{\{2\}} - m_2}$$

How can we compute softmax of the entire vector x?

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

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

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

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

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

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

Overall max

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

Adjusted $\vec{f}_1$ *and* $\vec{f}_2$

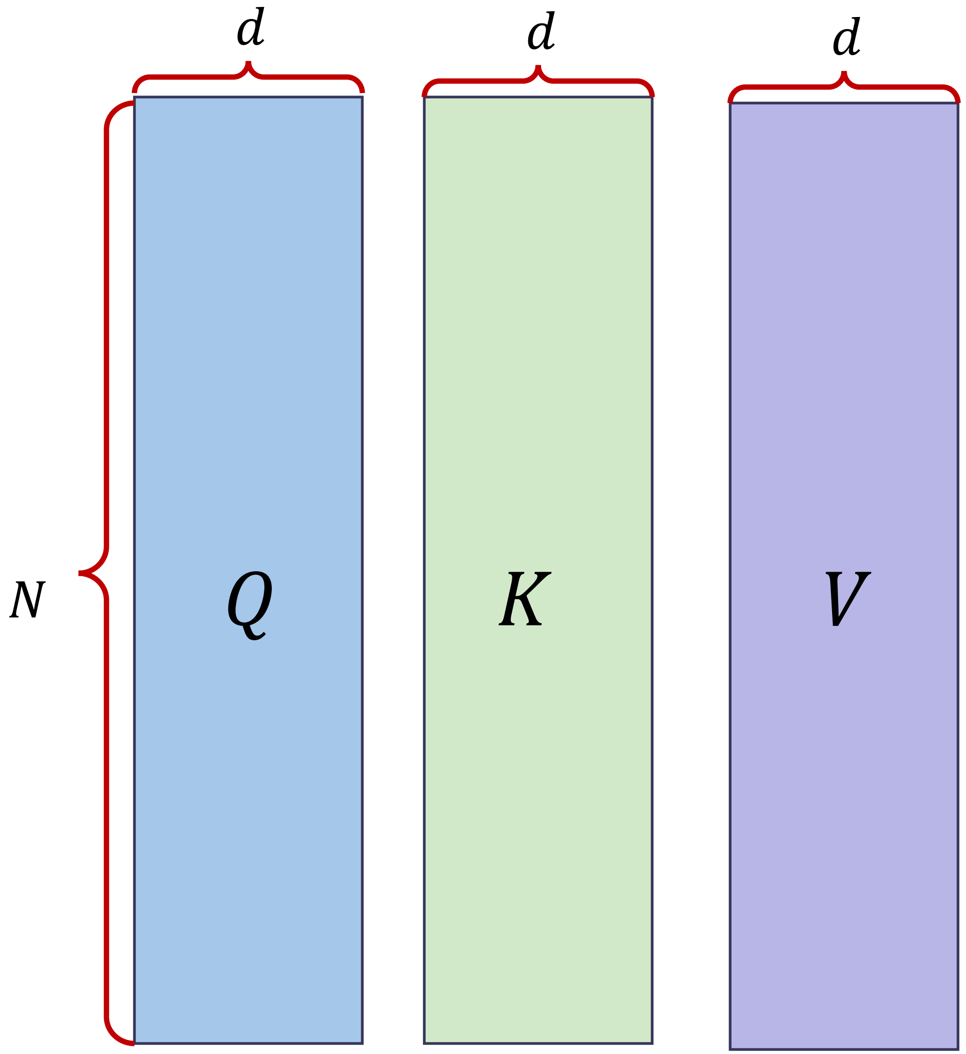
$$\vec{f}\,(x) = \left[e^{m_1 - m(x)} f_1, e^{m_2 - m(x)} f_2\right]$$

Adjusted normalization factor

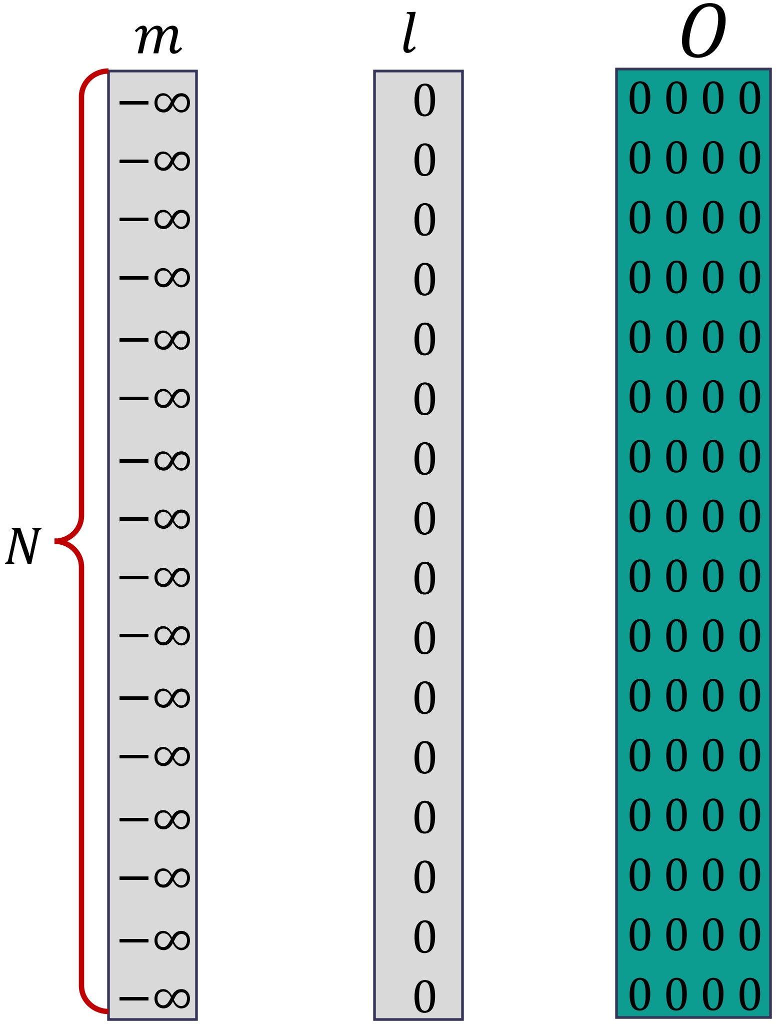
$$l(x) = e^{m_1 - m(x)} l_1 + e^{m_2 - m(x)} l_2$$

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

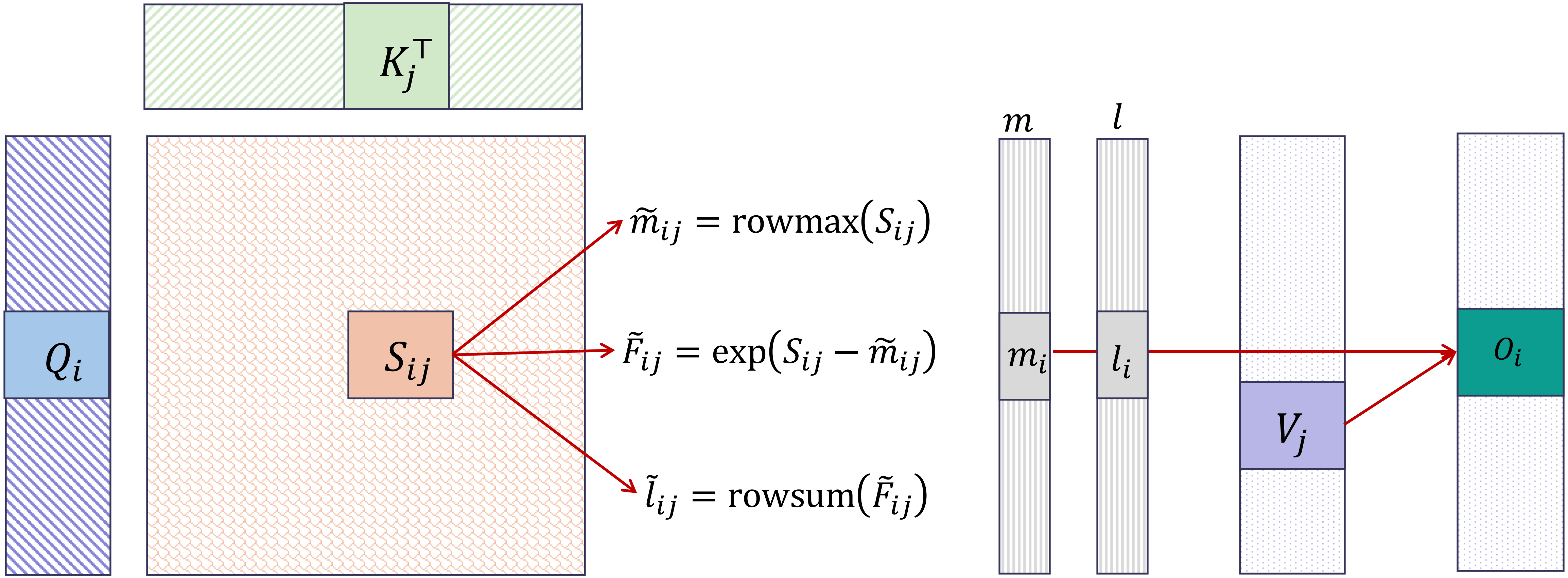
FlashAttention – Input and Initialization



Initialize intermediate vectors and the output:



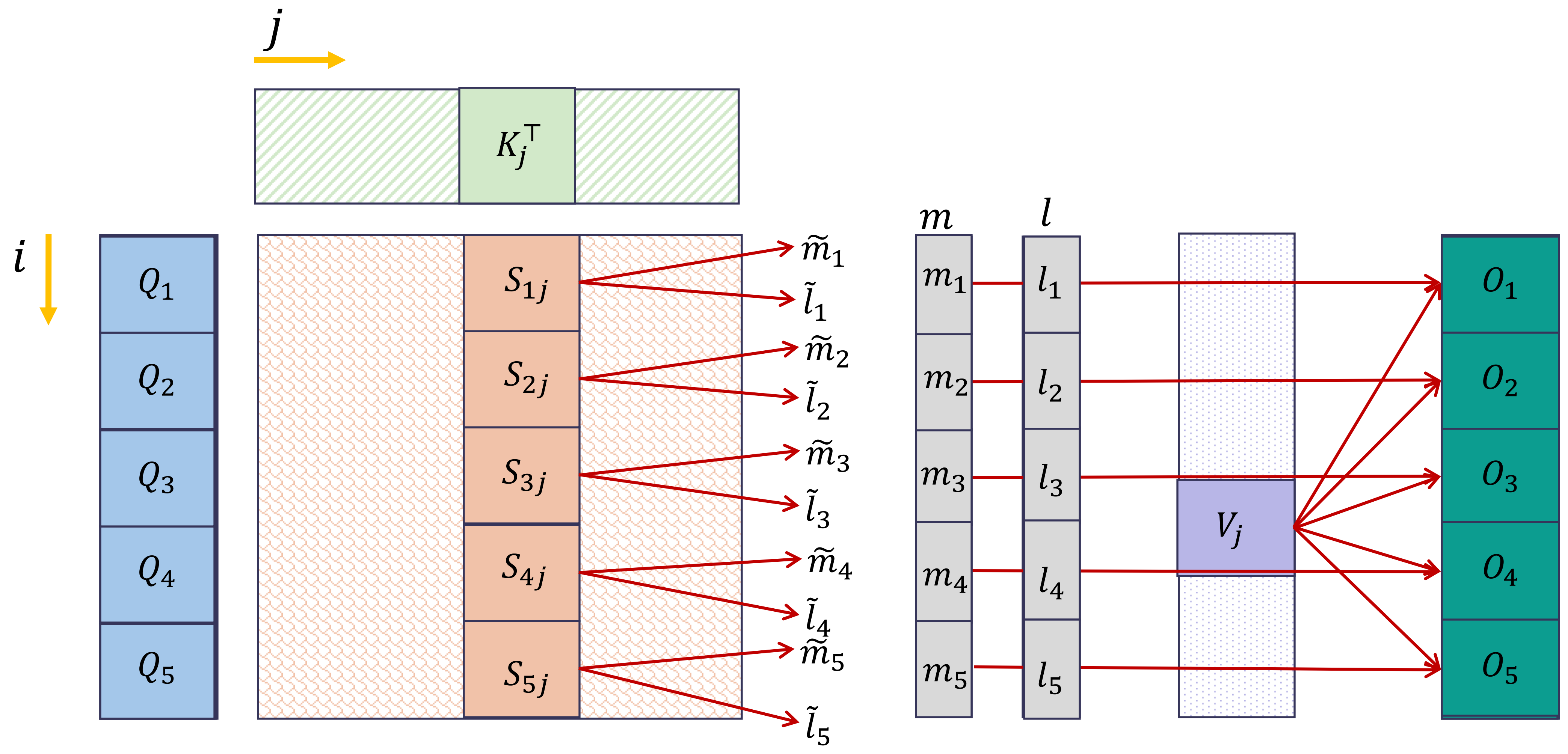
FlashAttention – Partial Updates



Partial update to block matrix O_i

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

FlashAttention – Partial Updates



HBM Access Comparison

Naïve Algorithm		FlashAttention
$O(Nd + N^2)$	$\gg$	$O(N^2 d^2 M^{-1})$

M: Size of SRAM

e.g. M =100KB → $M \ll d^2$

➡ FlashAttention reduces number of HBM accesses

Reduced training time of LLMs

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0×)
GPT-2 small - FlashAttention	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8×)
GPT-2 medium - FlashAttention	14.3	**6.9 days (3.0×)**

Increasing context length

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×)

Ref: https://arxiv.org/abs/2205.14135