FLASH attention

Or, why you should migrate to PyTorch 2.0

Desh Raj

April 14, 2023

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

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

[†]Department of Computer Science, Stanford University
[‡]Department of Computer Science and Engineering, University at Buffalo, SUNY

{trid,danfu}@cs.stanford.edu, ermon@stanford.edu, atri@buffalo.edu,
chrismre@cs.stanford.edu

Same group that developed S4 architecture

Read: https://hazyresearch.stanford.edu/blog/2023-03-27-long-learning

Questionnaire

Before we begin…

- Do you use Transformer-based models?
- Do you use PyTorch for training your models?
- Why are Transformers better at modeling sequence (compared to RNNs or convolutional layers)?
- What is the **time** complexity for self-attention?
- What is the **space** complexity for self-attention? (Other than inputs, output)

What about “efficient” transformers?

Long line of research to approximate self-attention

- Low-rank approximation of attention matrix
  - Linformer, Nystromformer
- Local-global attention
  - Longformer, Big Bird, Long-short transformer
- Softmax as a kernel
  - Transformers are RNNs, Performers

https://desh2608.github.io/2021-07-11-linear-transformers/

What about "efficient" transformers?

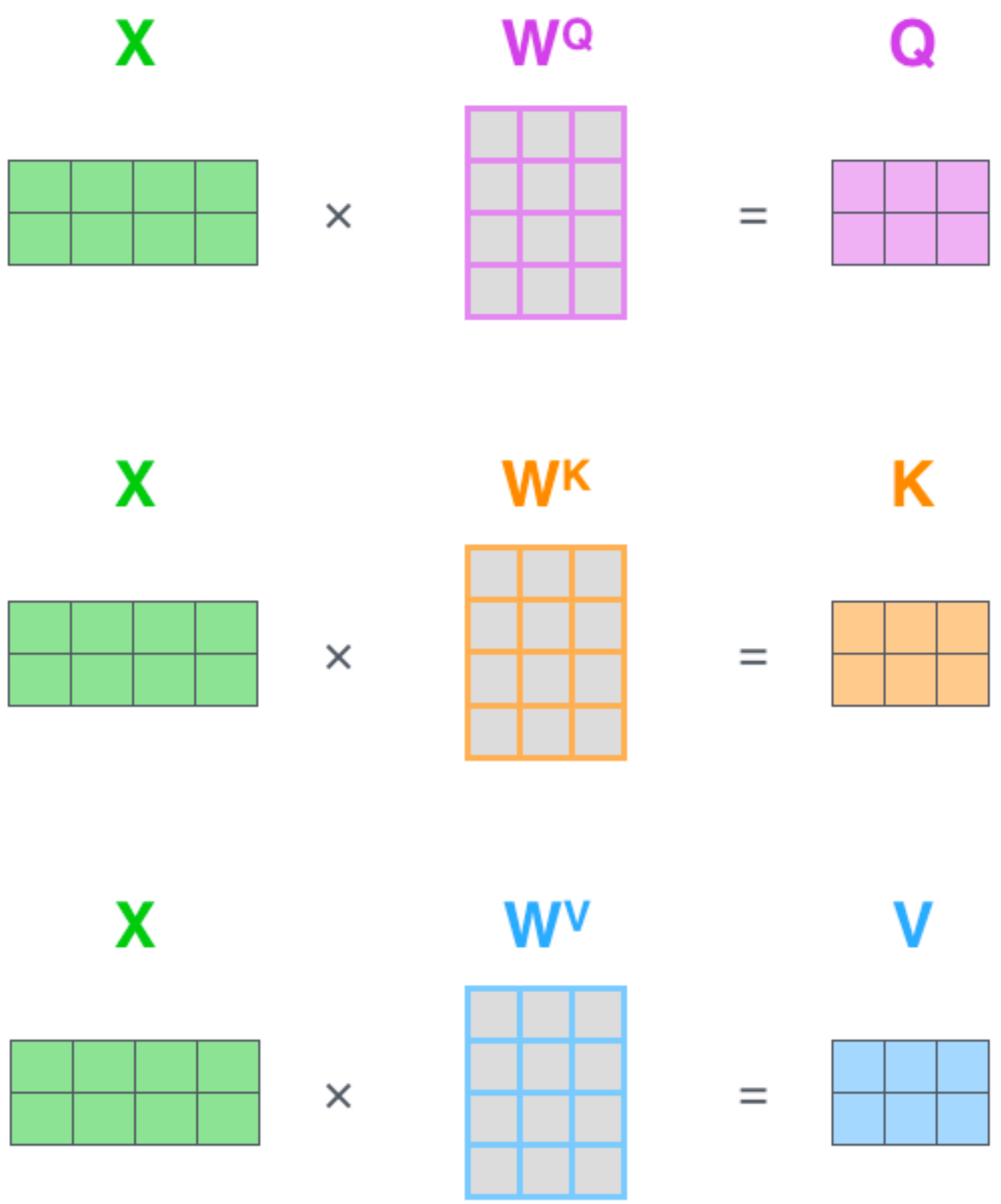
Long line of research to approximate self-attention

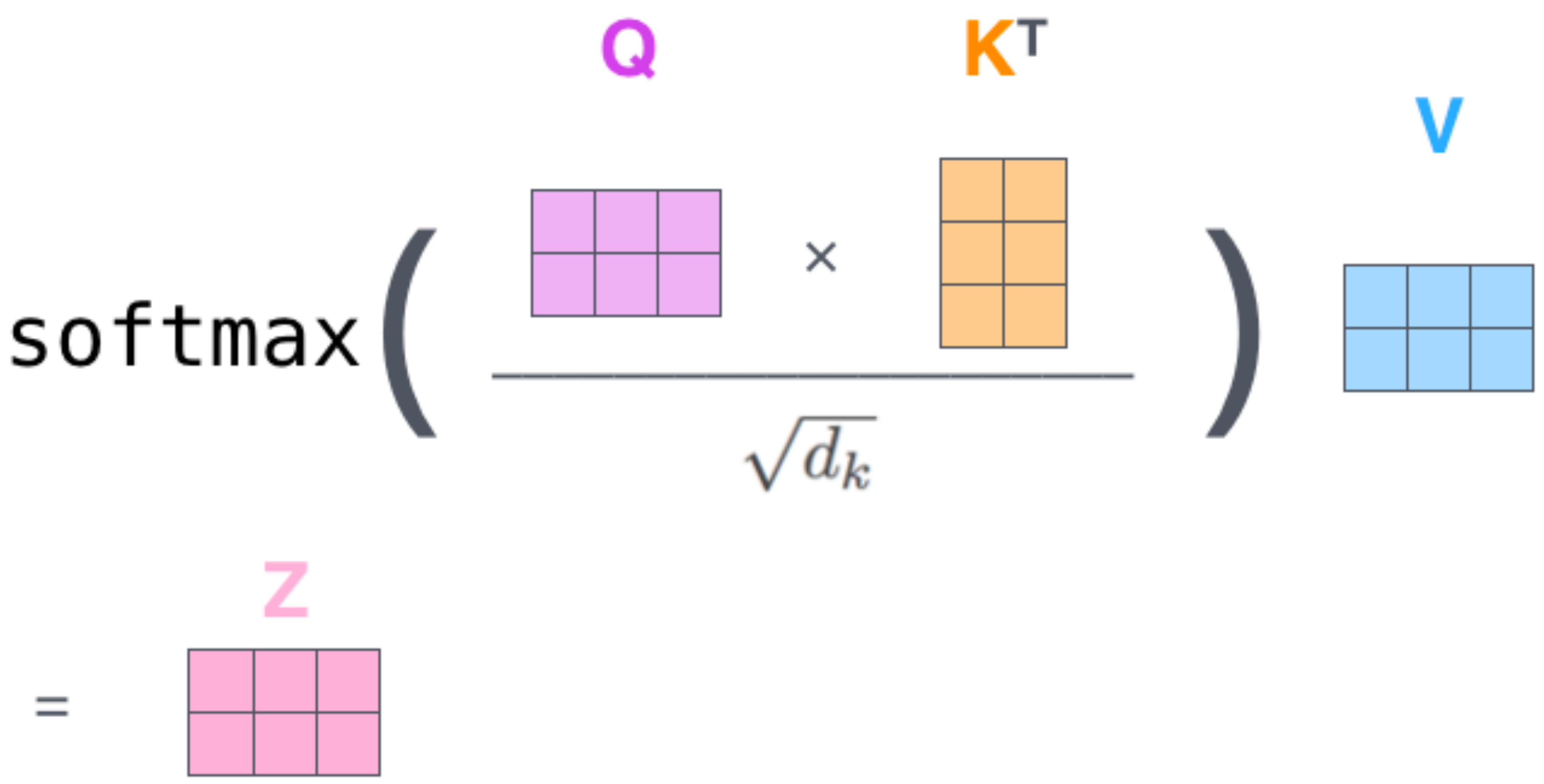
- All of these methods try to reduce *time-complexity* of self-attention
- But lower time-complexity does not really result in faster training/inference wall clock time on GPUs
- Approximations also lead to worse performance
- This is why these methods are not widely used

- What we want: FAST + EXACT self-attention

Multi-head self-attention

The “work-horse” of transformers





Usually also: Masking, Dropout

https://jalammar.github.io/illustrated-transformer/

Self-attention on the GPU

How is it implemented

$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}$	$\mathbf{S}' = \text{mask}(\mathbf{S})$	$\mathbf{P} = \text{softmax}(\mathbf{S}')$	$\mathbf{P}' = \text{dropout}(\mathbf{P})$	$\mathbf{z} = \mathbf{P}'\mathbf{V} \in \mathbb{R}^{N \times d}$
Matmul	**Masking**	**Softmax**	**Dropout**	**Matmul**

- ***Which of these operations are the most time-consuming on the GPU?***

Self-attention on the GPU

Which operations are the slowest?

Matmul $\mathbf{z} = \mathbf{P}'\mathbf{V} \in \mathbb{R}^{N \times d}$

↑

Dropout $\mathbf{P}' = \mathrm{dropout}(\mathbf{P})$

↑

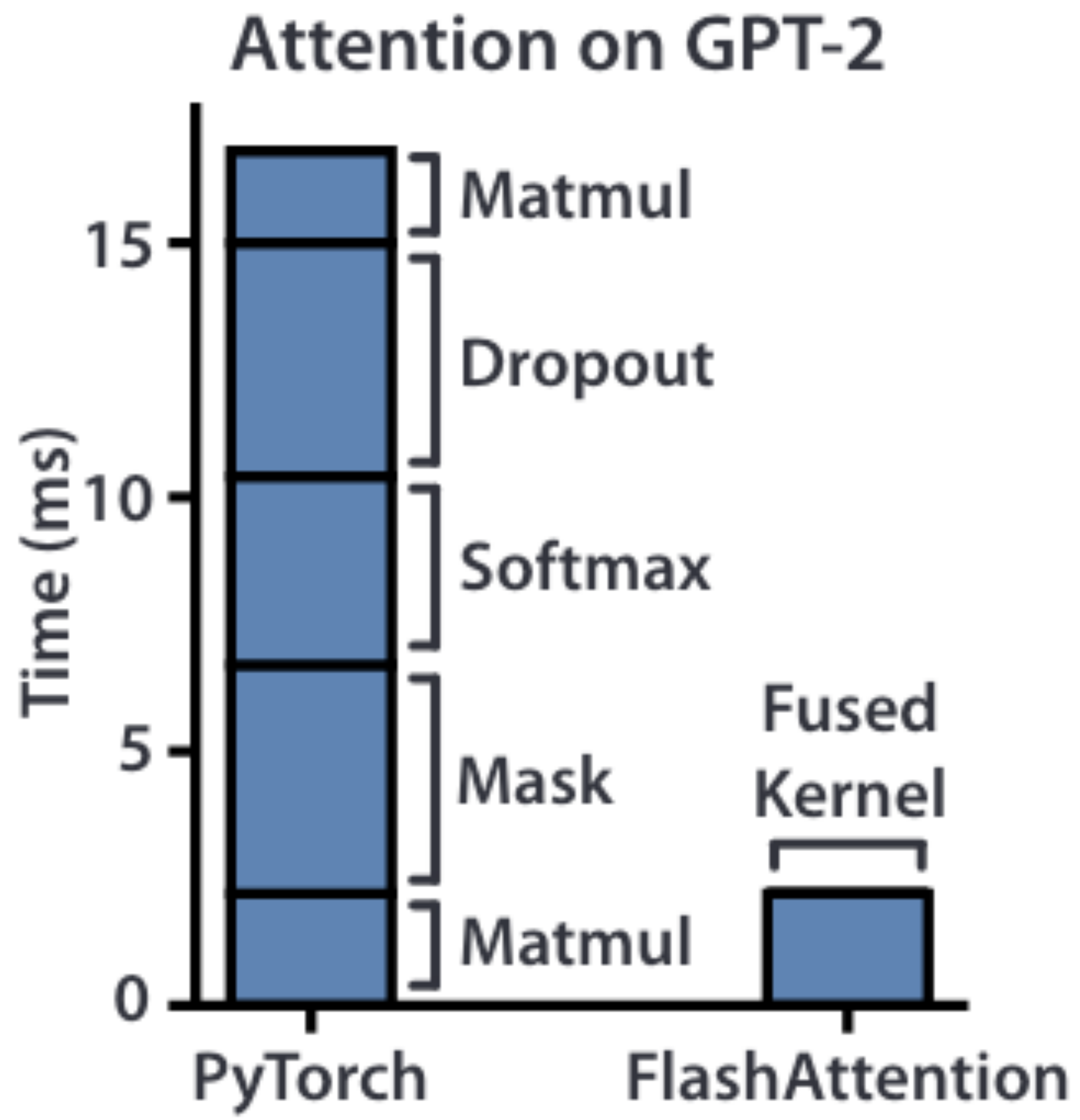
Softmax $\mathbf{P} = \mathrm{softmax}(\mathbf{S}')$

↑

Masking $\mathbf{S}' = \mathrm{mask}(\mathbf{S})$

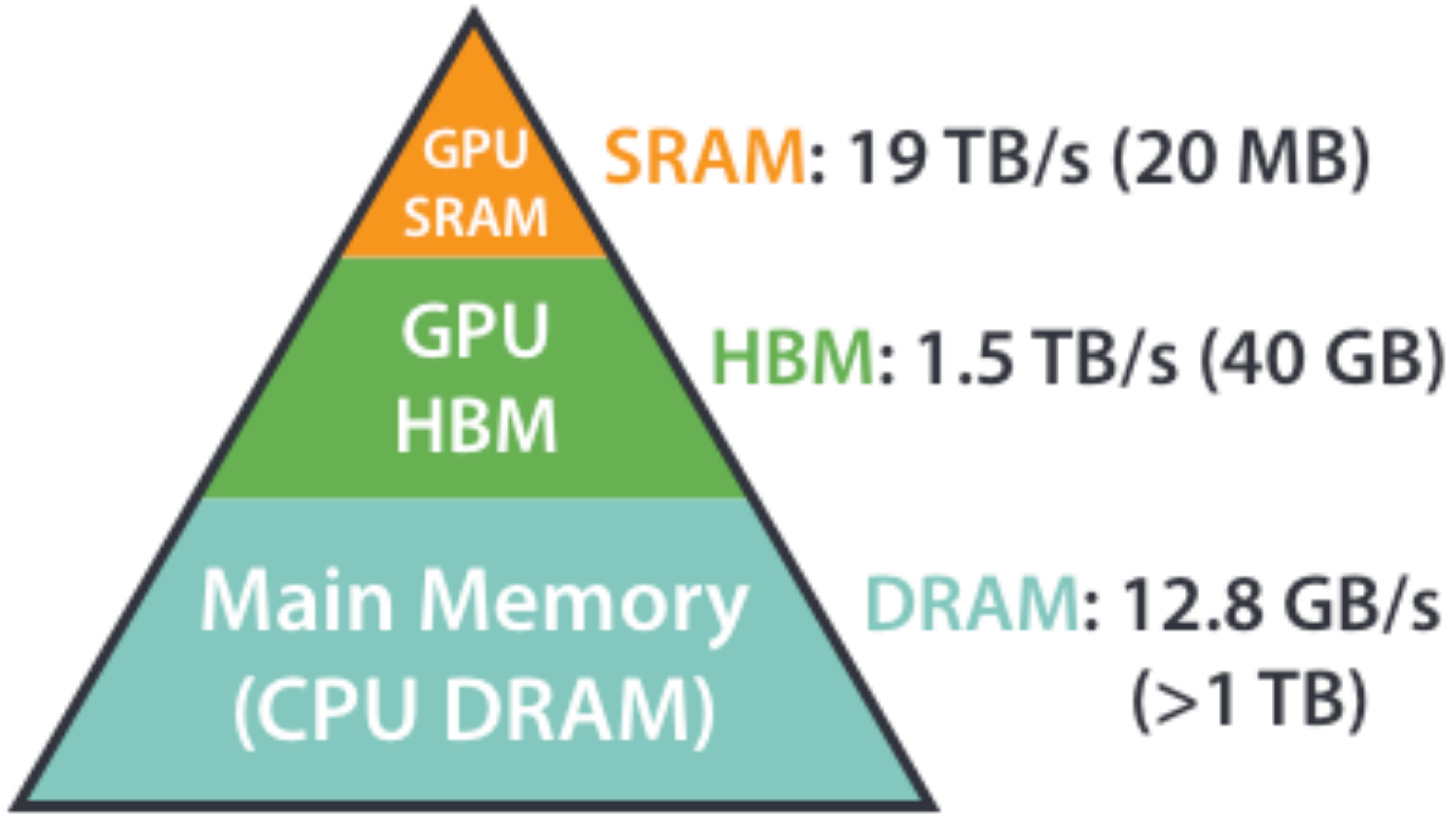
↑

Matmul $\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}$

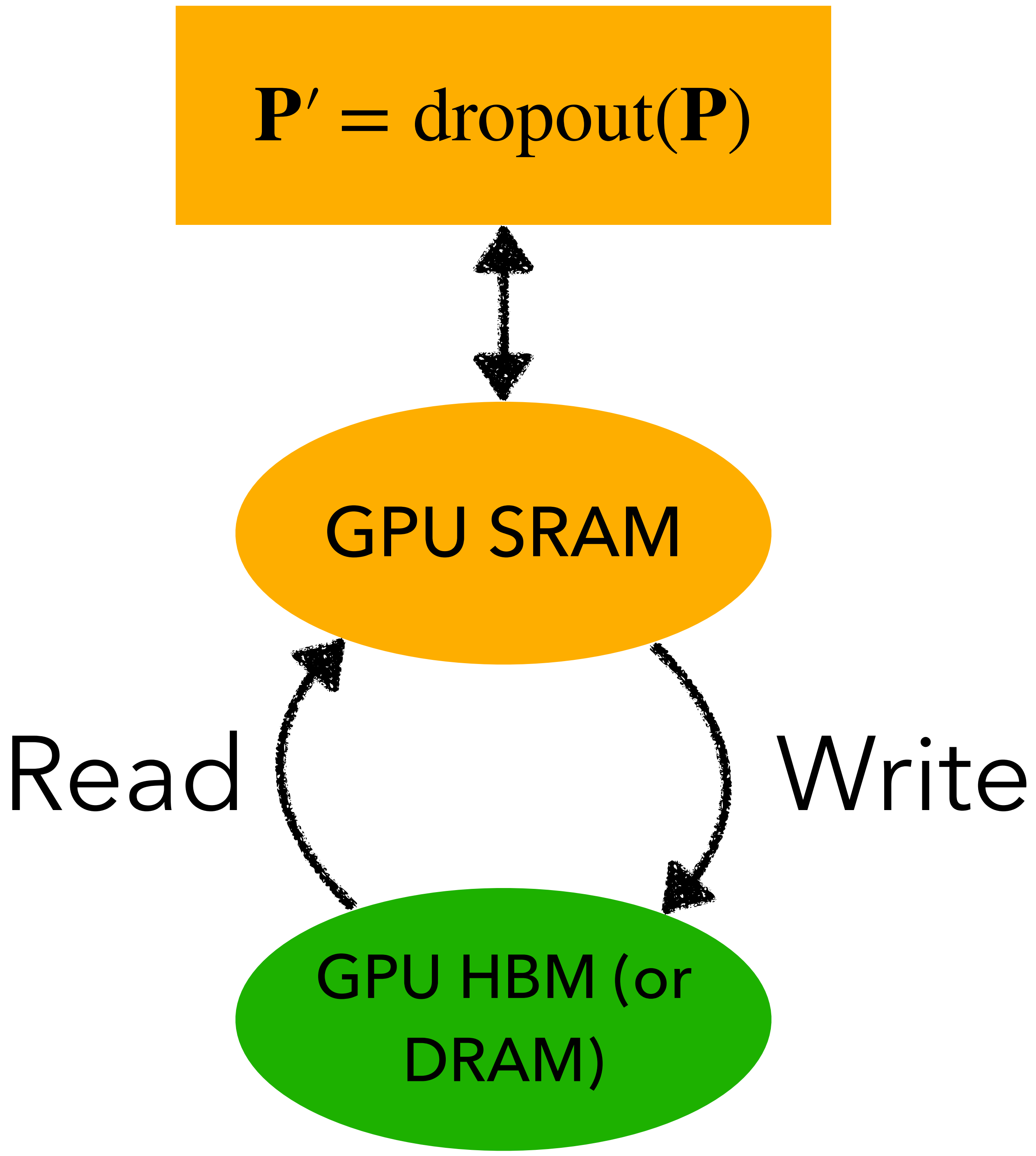


This seems counter-intuitive

GPU memory hierarchy

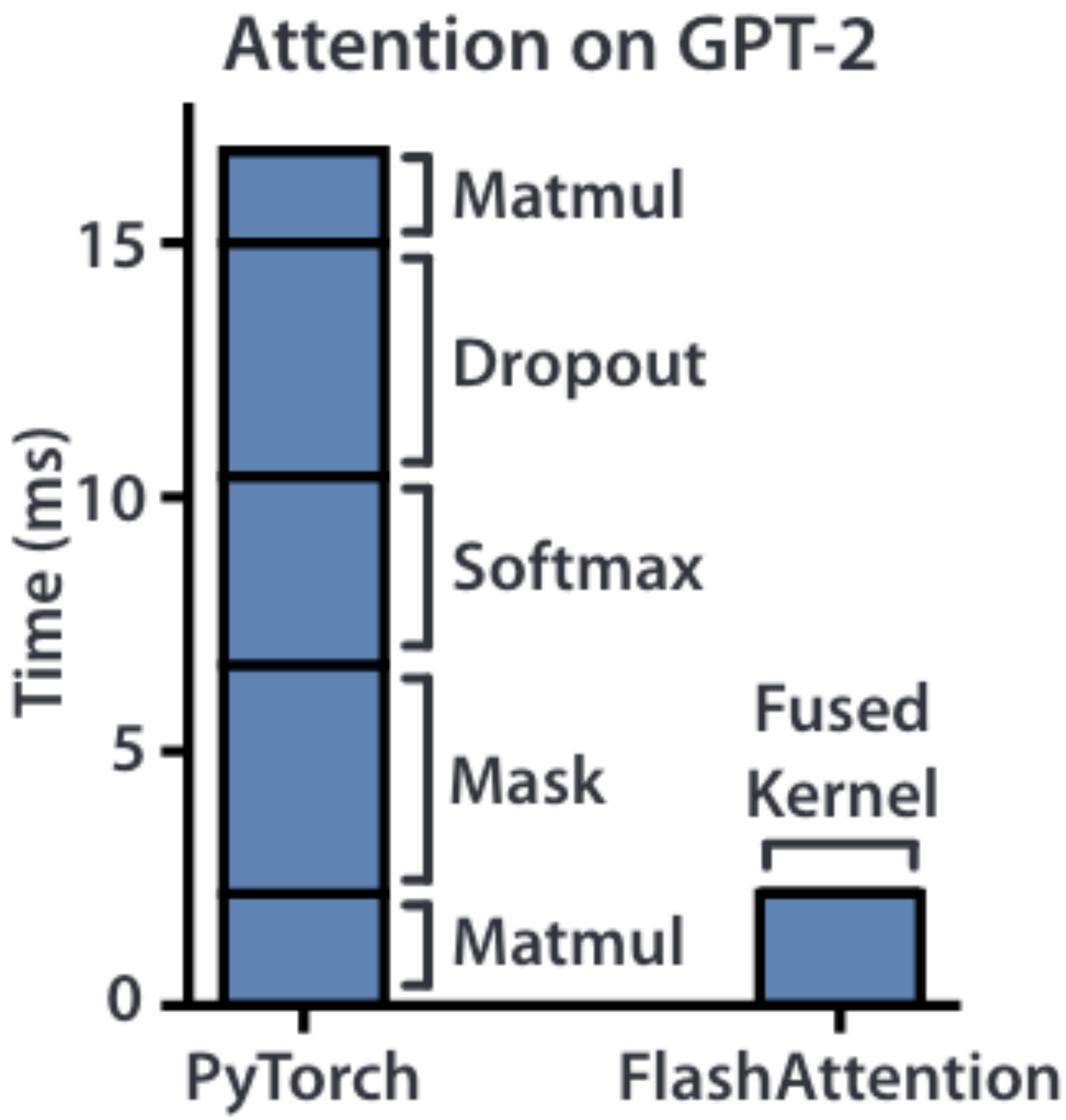


Memory Hierarchy with Bandwidth & Memory Size



Self-attention on the GPU

		Read	Write
Matmul	$\mathbf{z} = \mathbf{P}'\mathbf{V} \in \mathbb{R}^{N \times d}$	N*(N+d)	N*d
Dropout	$\mathbf{P}' = \mathrm{dropout}(\mathbf{P})$	N*N	N*N
Softmax	$\mathbf{P} = \mathrm{softmax}(\mathbf{S}')$	N*N	N*N
Masking	$\mathbf{S}' = \mathrm{mask}(\mathbf{S})$	N*N	N*N
Matmul	$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}$	2*N*d	N*N



How can we speed-up self-attention?

Reduce number of read/write from HBM to SRAM

- Basically we want to avoid creating the big attention matrix which is of N^2 size.
- But there are 2 problems?
  - **Problem 1:** How to compute softmax without the entire sequence?
  - **Problem 2:** How to compute gradients for back-propagation?

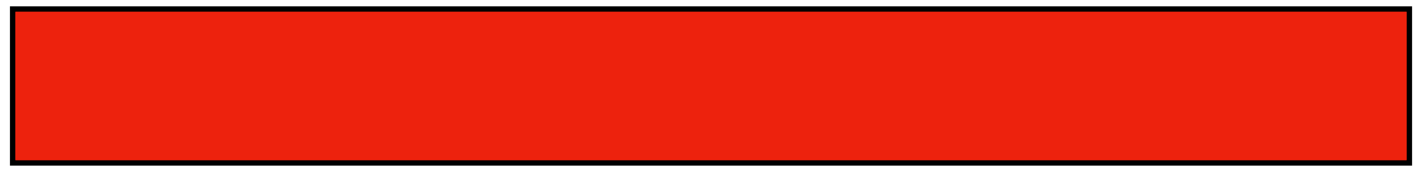
Solution

- **Problem 1:** solved by TILING
- **Problem 2:** solved by RECOMPUTATION

Tiling to solve Problem 1

Block-wise computation of softmax

$$\max(\mathbf{x}) = m$$



$\mathbf{x}$

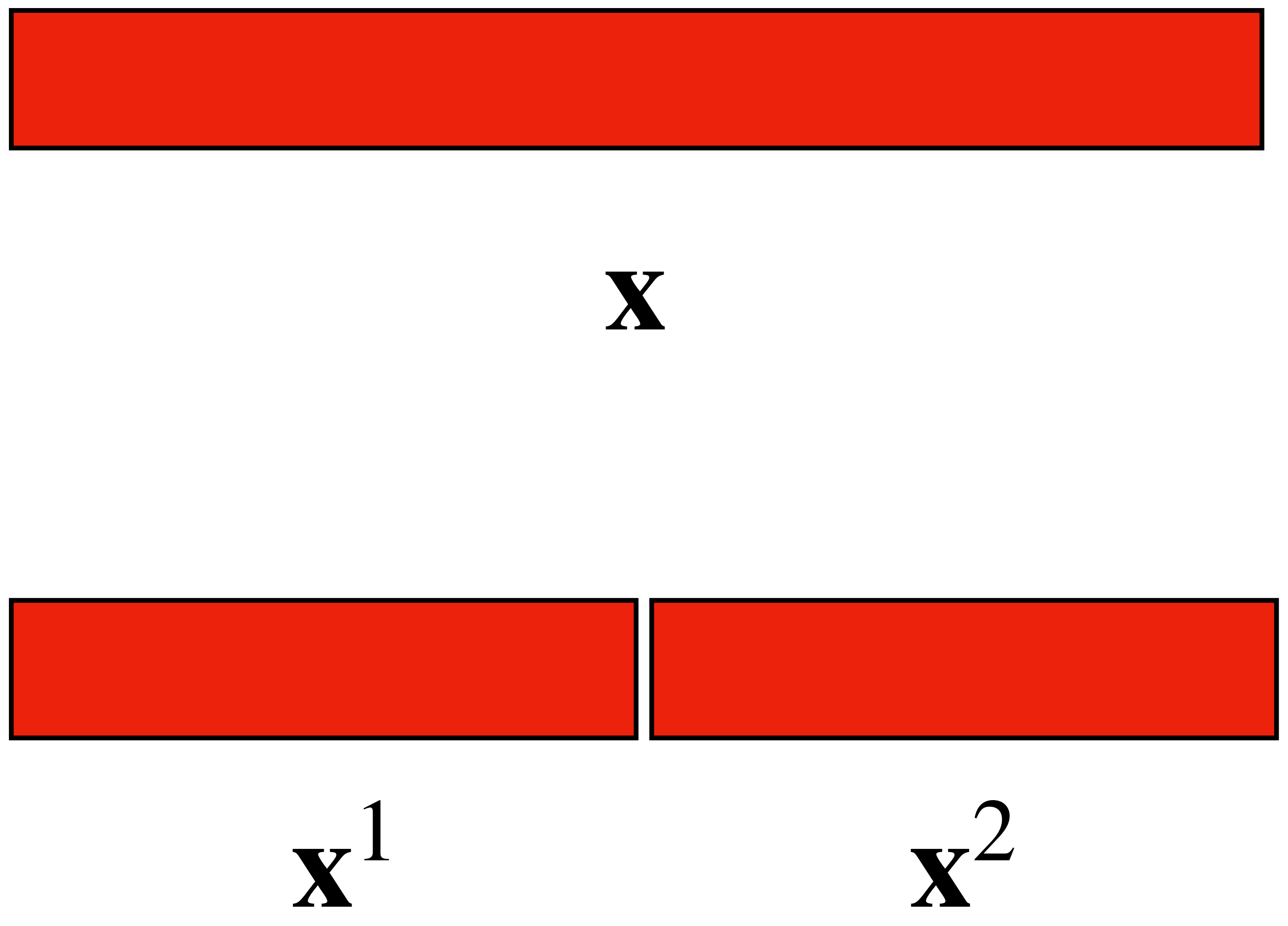
$$f(\mathbf{x}) = [e^{x_1 - m}, \ldots, e^{x_{2N} - m}]$$

$$\mathrm{softmax}(\mathbf{x}) = \frac{f(\mathbf{x})}{l(\mathbf{x})}$$

$$l(\mathbf{x}) = \sum_i f(\mathbf{x})_i$$

Tiling to solve Problem 1

Block-wise computation of softmax



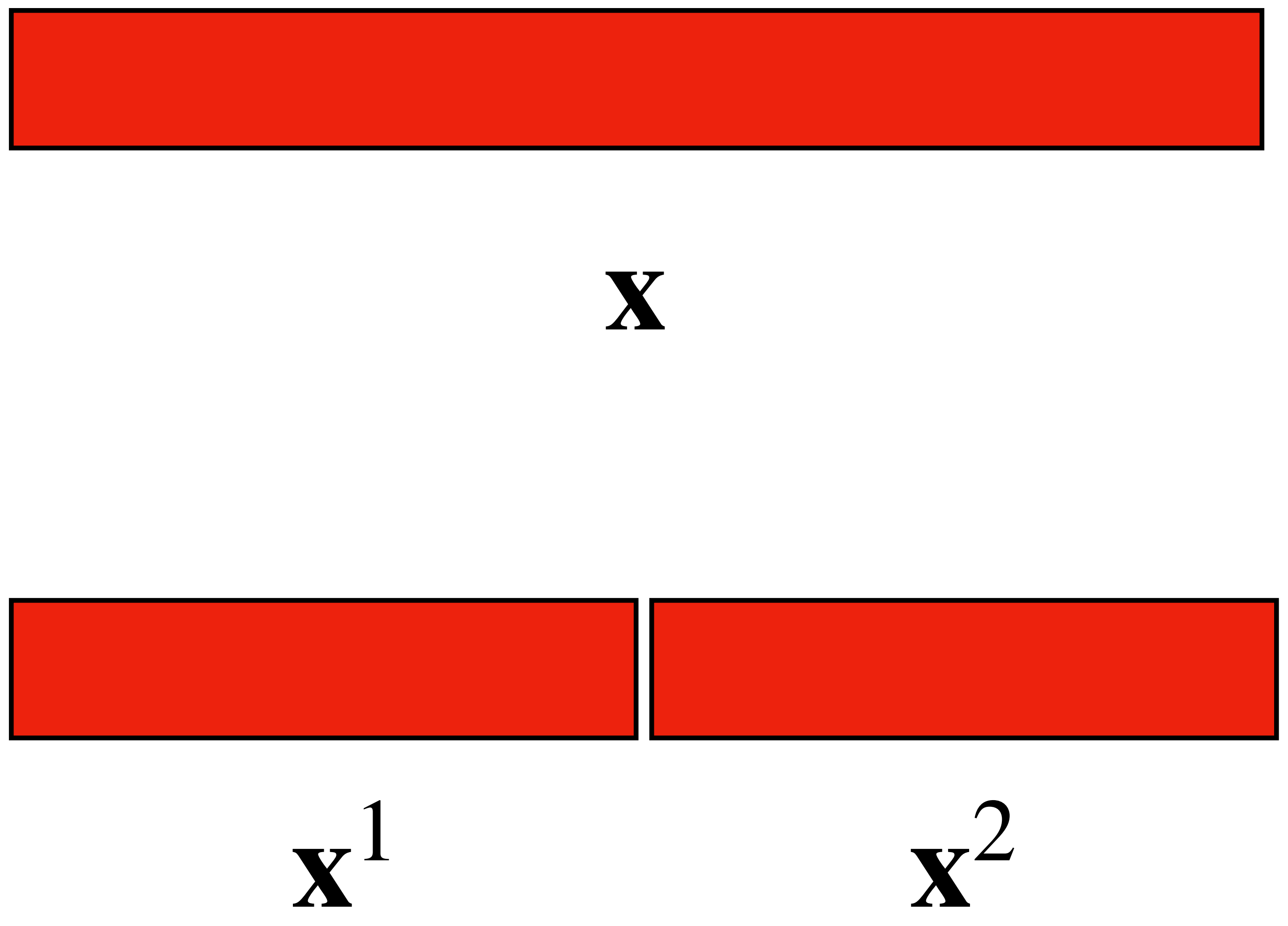
$$\max(\mathbf{x}^1) = m_1 \qquad \max(\mathbf{x}^2) = m_2$$

$$f(\mathbf{x}^1) = [e^{x_1 - m_1}, \ldots, e^{x_N - m_1}] \qquad f(\mathbf{x}^2) = [e^{x_{N+1} - m_2}, \ldots, e^{x_{2N} - m_2}]$$

$$l(\mathbf{x}^1) = \sum_i f(\mathbf{x}^1)_i \qquad l(\mathbf{x}^2) = \sum_i f(\mathbf{x}^2)_i$$

Tiling to solve Problem 1

Block-wise computation of softmax



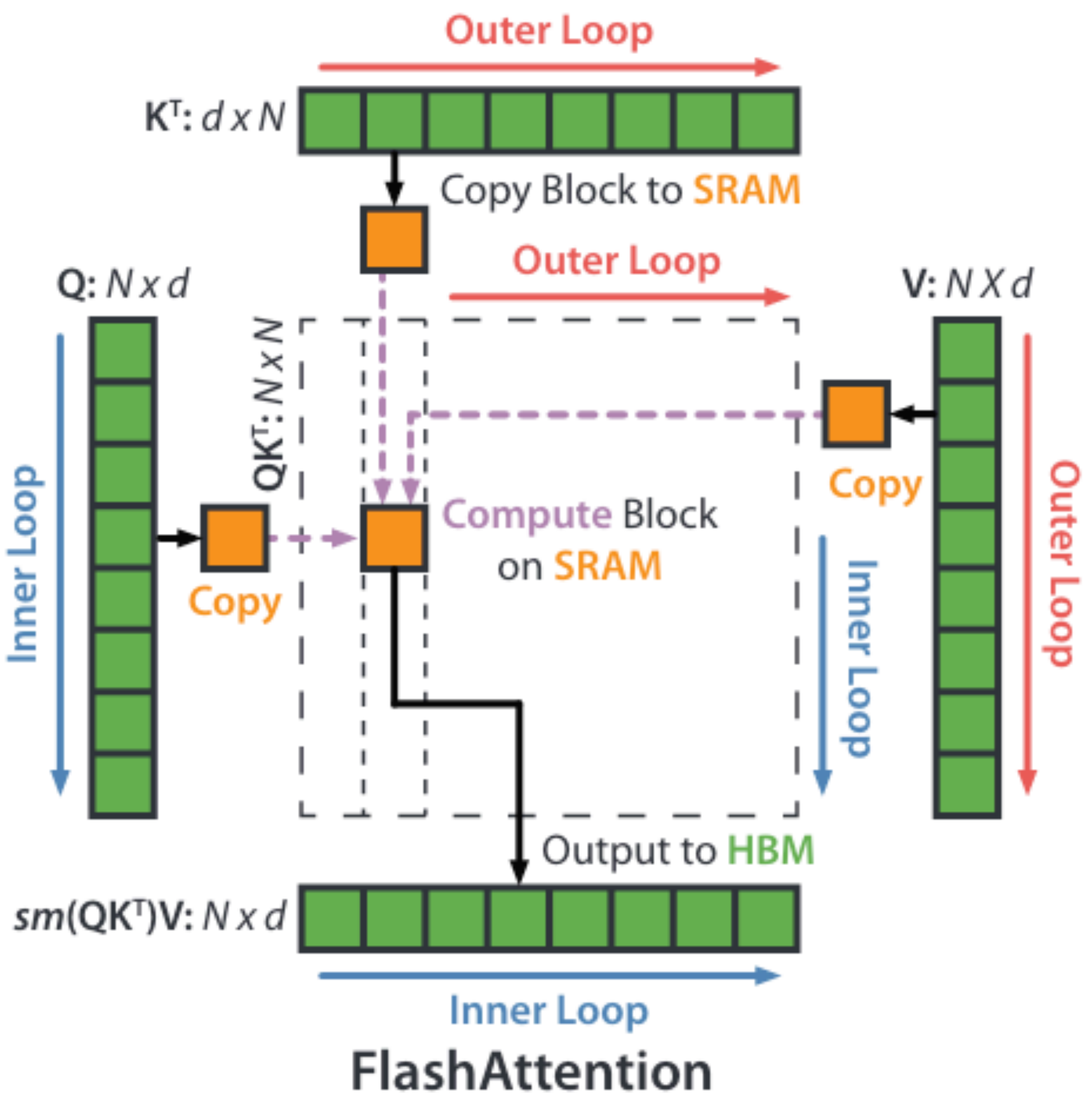
$$\max(\mathbf{x}) = m = \max(m_1, m_2)$$

$$\begin{aligned} f(\mathbf{x}) &= [e^{x_1 - m}, \ldots, e^{x_{2N} - m}] \\ &= [e^{x_1 - m_1 + m_1 - m}, \ldots, e^{x_{2N} - m_2 + m_2 - m}] \\ &= [e^{m_1 - m} \cdot e^{x_1 - m_1}, \ldots, e^{m_2 - m} \cdot e^{x_{2N} - m_2}] \\ &= [e^{m_1 - m} \cdot f(\mathbf{x}^1), e^{m_2 - m} \cdot f(\mathbf{x}^2)] \end{aligned}$$

$$l(\mathbf{x}) = e^{m_1 - m} \cdot l(\mathbf{x}^1) + e^{m_2 - m} \cdot l(\mathbf{x}^2)$$

Tiling to solve Problem 1

Block-wise computation of self-attention output



Recomputation to solve Problem 2

How do we get gradients for back-prop

$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}$ → $\mathbf{S}' = \text{mask}(\mathbf{S})$ → $\mathbf{P} = \text{softmax}(\mathbf{S}')$ → $\mathbf{P}' = \text{dropout}(\mathbf{P})$ → $\mathbf{z} = \mathbf{P}'\mathbf{V} \in \mathbb{R}^{N \times d}$

- Need to compute gradients with respect to **Q, K, V**
- In original implementation, we will need to store all intermediate matrices.
- But in new implementation, we don't have these big intermediate matrices.

Recomputation to solve Problem 2

How do we get gradients for back-prop

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N} \rightarrow \mathbf{S}' = \text{mask}(\mathbf{S}) \rightarrow \mathbf{P} = \text{softmax}(\mathbf{S}') \rightarrow \mathbf{P}' = \text{dropout}(\mathbf{P}) \rightarrow \mathbf{z} = \mathbf{P}'\mathbf{V} \in \mathbb{R}^{N \times d}$$

- It can be shown through some matrix calculus that we can compute derivatives of **Q, K, V** without the intermediates, by using the stored statistics m and l.
- This requires recomputing some things, but it is fast in SRAM.
- Total FLOPs increases, but wall clock time decreases.

Results

Run-time for GPT-2

Attention	Standard	FLASHATTENTION
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

FLASH attention has more FLOPs but much lower runtime due to less HBM access.

Results

Faster training with FLASH attention

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×)**

Performance is same because there is no approximation.

Results

Performance on Long Range Arena (LRA) benchmark

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×

Performance is better because other methods use approximation.

Results

Modeling longer sequences

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×)

We can use longer contexts since we don't require quadratic memory.

So what does it have to do with PyTorch?

- PyTorch 2.0 has native support for FLASH attention!
- Caveat: attention masks are not supported yet.

https://pytorch.org/tutorials/intermediate/scaled_dot_product_attention_tutorial.html