


Machine Learning Department
School of Computer Science
Carnegie Mellon University

# Efficient Attention (FlashAttention)

Matt Gormley & Pat Virtue
Lecture 18
Mar. 24, 2025

# Reminders

- Homework 4: Visual Language Models
  - Out: Thu, Mar 13
  - Due: Mon, Mar 24 at 11:59pm
- Exam
  - Date: In-class, Monday, Mar 31
  - Time: 75 minutes, taking up the whole class time
  - Covered Material: Lectures 1 – 15 (same as Quiz 1 – Quiz 4)
  - You may bring one sheet of notes (front and back)
  - Format of questions: Unlike the Quiz questions, which were all multiple choice, Exam questions will include open-ended questions as well
  - Check Piazza for seat assignment

# Why do we care about FlashAttention?

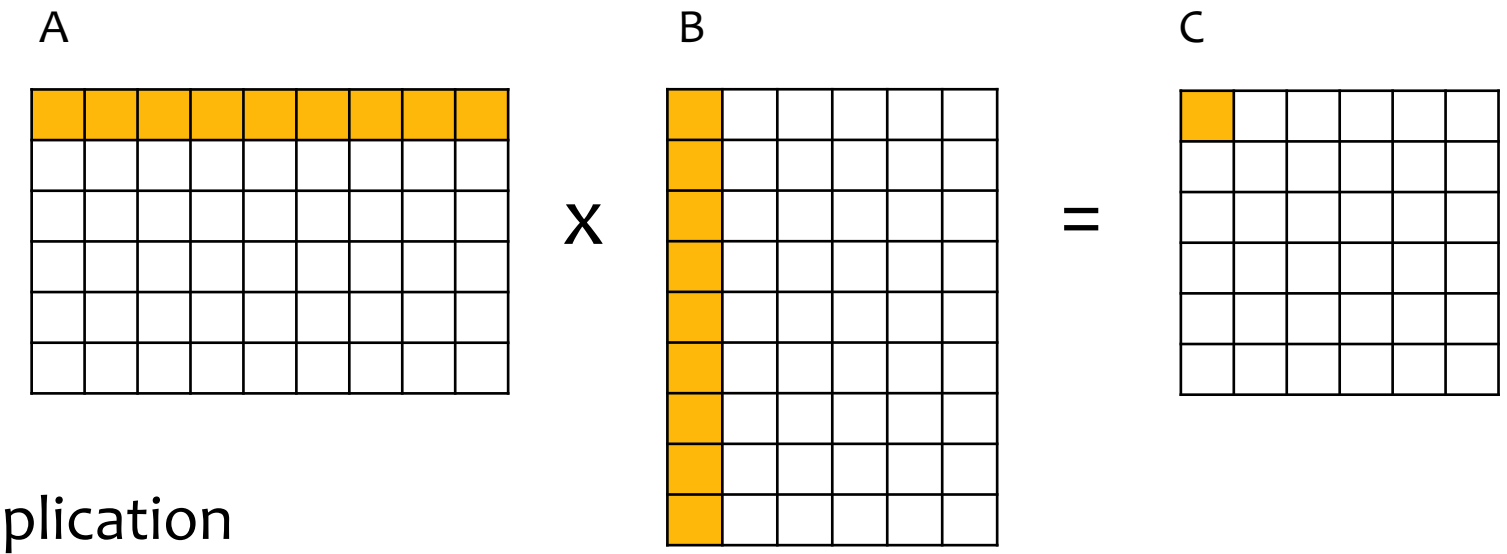
- The algorithm is performing exact attention, so we see no reduction in perplexity or quality of the model
- The key metric is runtime

| 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×)** |

Background

# TILING FOR MATRIX MULTIPLICATION

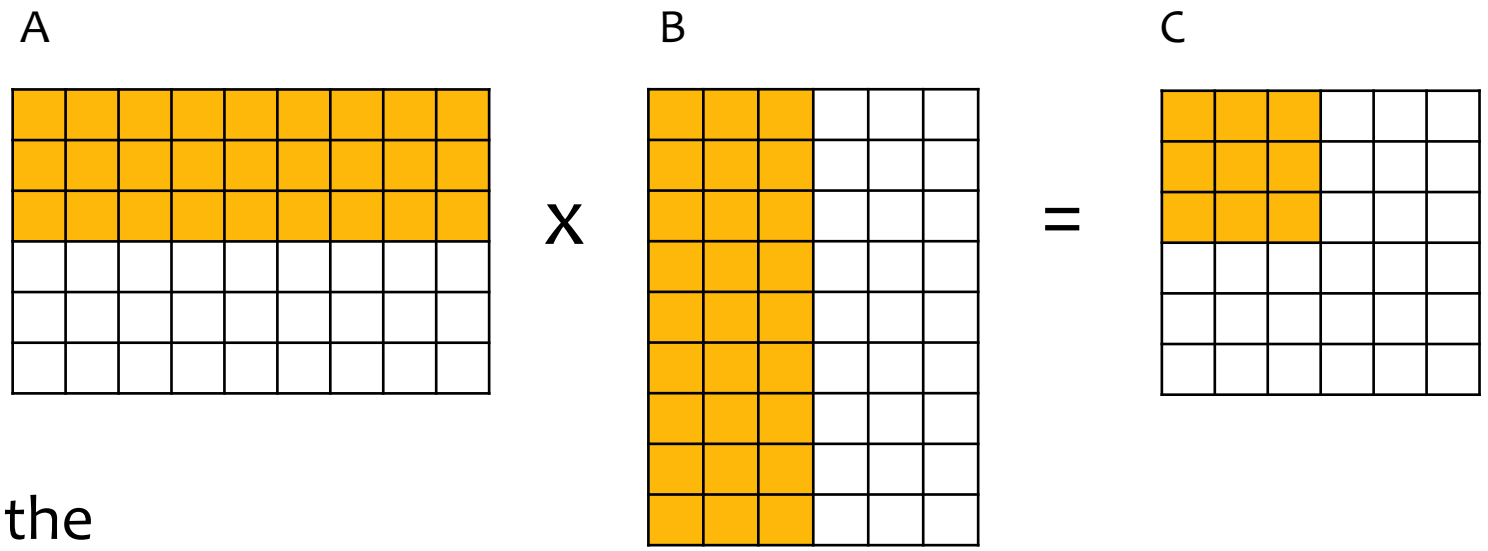
# Tiling for Matrix Multiplication




- Matrix multiplication computes each output value as a dot-product of a row/column pair from the input matrices

$$C_{ij} = \sum_{m=1}^{M} \sum_{n=1}^{N} A_{im} B_{nj}$$

# Tiling for Matrix Multiplication

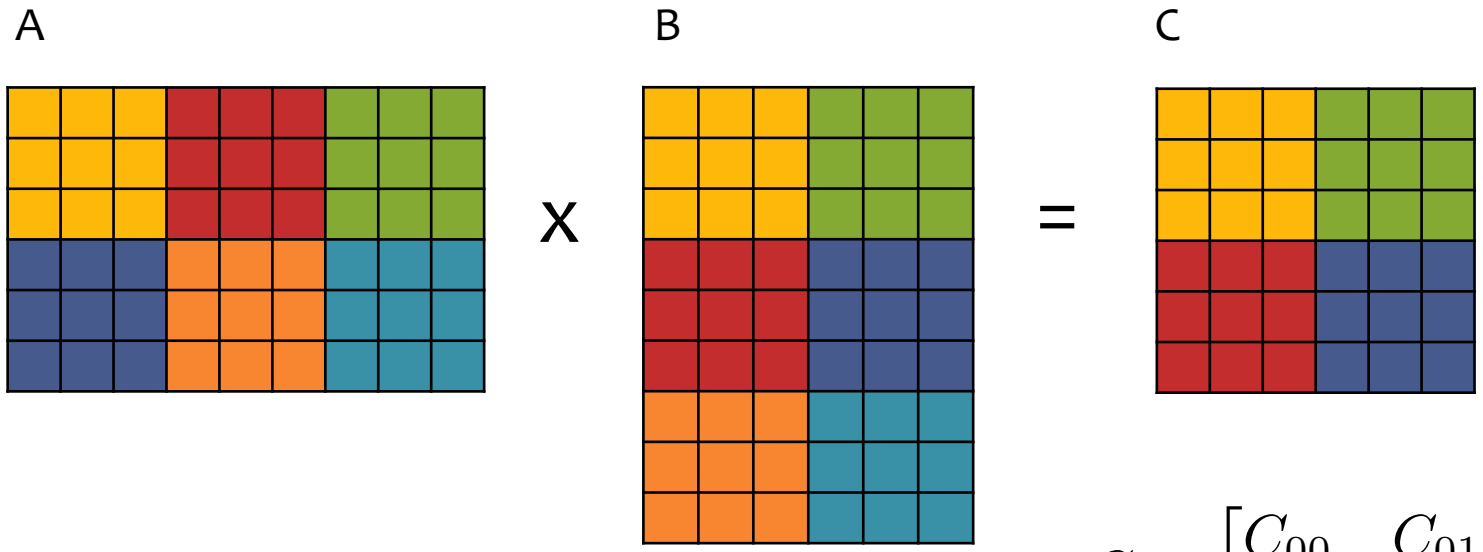



- We can view the computation as decomposing if we consider subsets of rows/columns

$$C_{(1,1):(3,3)} = A_{(1,1):(3,9)} \times B_{(1,1):(9,3)}$$

# Tiling for Matrix Multiplication

- Tiling capitalizes on this decomposition
- Each output tile is computed by multiplying a pair of input tiles and adding it to the appropriate output tile




$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

with each $A_{ij} \in \mathbb{R}^{3\times3}$

$$B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \\ B_{20} & B_{21} \end{bmatrix}$$

with each $B_{ij} \in \mathbb{R}^{3\times3}$

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

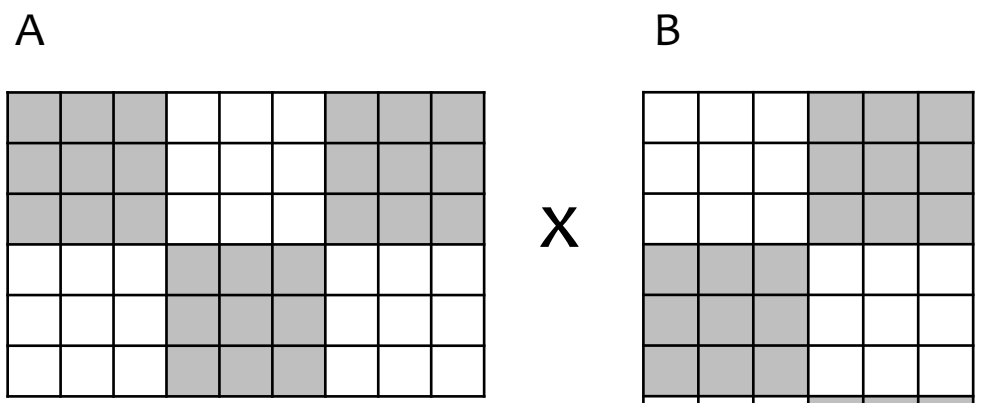
with each $C_{ij} \in \mathbb{R}^{3\times3}$

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

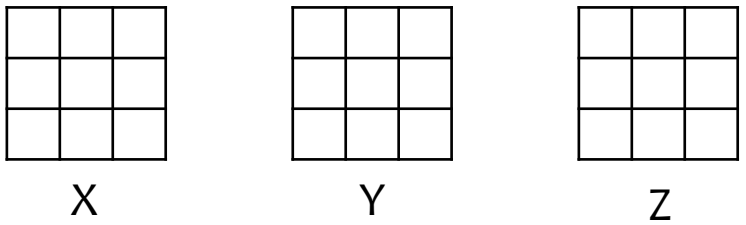
# Tiling for Matrix Multiplication

large/slow memory

- Tiling enables matrix multiplication of two **very** large matrices to capitalize on the small amount of fast memory on a device (e.g. GPU)
- Start by putting the input matrices and storage for the output matrix into large/slow memory
- Do the primary computation in slow/fast memory




$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$




small/fast memory

$$X = A_{00}$$
$$Y = B_{00}$$
$$Z = XY$$
$$X = A_{01}$$
$$Y = B_{10}$$
$$Z = Z + XY$$

$$X = A_{02}$$
$$Y = B_{20}$$
$$Z = Z + XY$$
$$C_{00} = Z$$

# Tiling for Self-Attention?

- It would be great if we could directly use tiling for self-attention
- Unfortunately, whereas the addition in matrix multiplication is associative, the softmax in self-attention is not!

$$\mathbf{X}' = \text{softmax}(\mathbf{Q}\mathbf{K}^T / \sqrt{d_k})\mathbf{V}$$

Background

# ONLINE SOFTMAX

# Regular Softmax

- The standard softmax computation is used heavily throughout deep learning
- Yet, often we need to compute softmax on very large logits
- To avoid issues of overflow when raising e to some large power, we can use the safe softmax instead
- Every deep learning library implements this

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$$

**Algorithm 1** Naive softmax

1: $d_0 \leftarrow 0$
2: **for** $j \leftarrow 1, V$ **do**
3: $\quad d_j \leftarrow d_{j-1} + e^{x_j}$
4: **end for**
5: **for** $i \leftarrow 1, V$ **do**
6: $\quad y_i \leftarrow \frac{e^{x_i}}{d_V}$
7: **end for**

Figure from https://arxiv.org/pdf/1805.02867

# Safe Softmax

- The standard softmax computation is used heavily throughout deep learning
- Yet, often we need to compute softmax on very large logits
- To avoid issues of overflow when raising e to some large power, we can use the safe softmax instead
- Every deep learning library implements this

$$y_i = \frac{e^{x_i - \max_{k=1}^{V} x_k}}{\sum_{j=1}^{V} e^{x_j - \max_{k=1}^{V} x_k}}$$

**Algorithm 2** Safe softmax

$m_0 \leftarrow -\infty$
**for** $k \leftarrow 1, V$ **do**
    $m_k \leftarrow \max(m_{k-1}, x_k)$
**end for**
$d_0 \leftarrow 0$
**for** $j \leftarrow 1, V$ **do**
    $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
**end for**
**for** $i \leftarrow 1, V$ **do**
    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
**end for**

Figure from https://arxiv.org/pdf/1805.02867

# Online Softmax

- The problem with the usual safe softmax is that it requires three iterations, with each one accessing memory
- Online softmax reduces this to only two iterations through the data!
- This results in not only a 1.33x apparent speedup, but also a 1.3x speedup in practice because of reduced memory bandwidth requirements

**Algorithm 3** Safe softmax with online normalizer calculation

$m_0 \leftarrow -\infty$
$d_0 \leftarrow 0$
**for** $j \leftarrow 1, V$ **do**
    $m_j \leftarrow \max\left(m_{j-1}, x_j\right)$
    $d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j-m_j}$
**end for**
**for** $i \leftarrow 1, V$ **do**
    $y_i \leftarrow \frac{e^{x_i-m_V}}{d_V}$
**end for**

Figure from https://arxiv.org/pdf/1805.02867

# Online Softmax

- The problem with the usual safe softmax is that it requires three iterations, with each one accessing memory
- Online softmax reduces this to only two iterations through the data!
- This results in not only a 1.33x apparent speedup, but also a 1.3x speedup in practice because of reduced memory bandwidth requirements

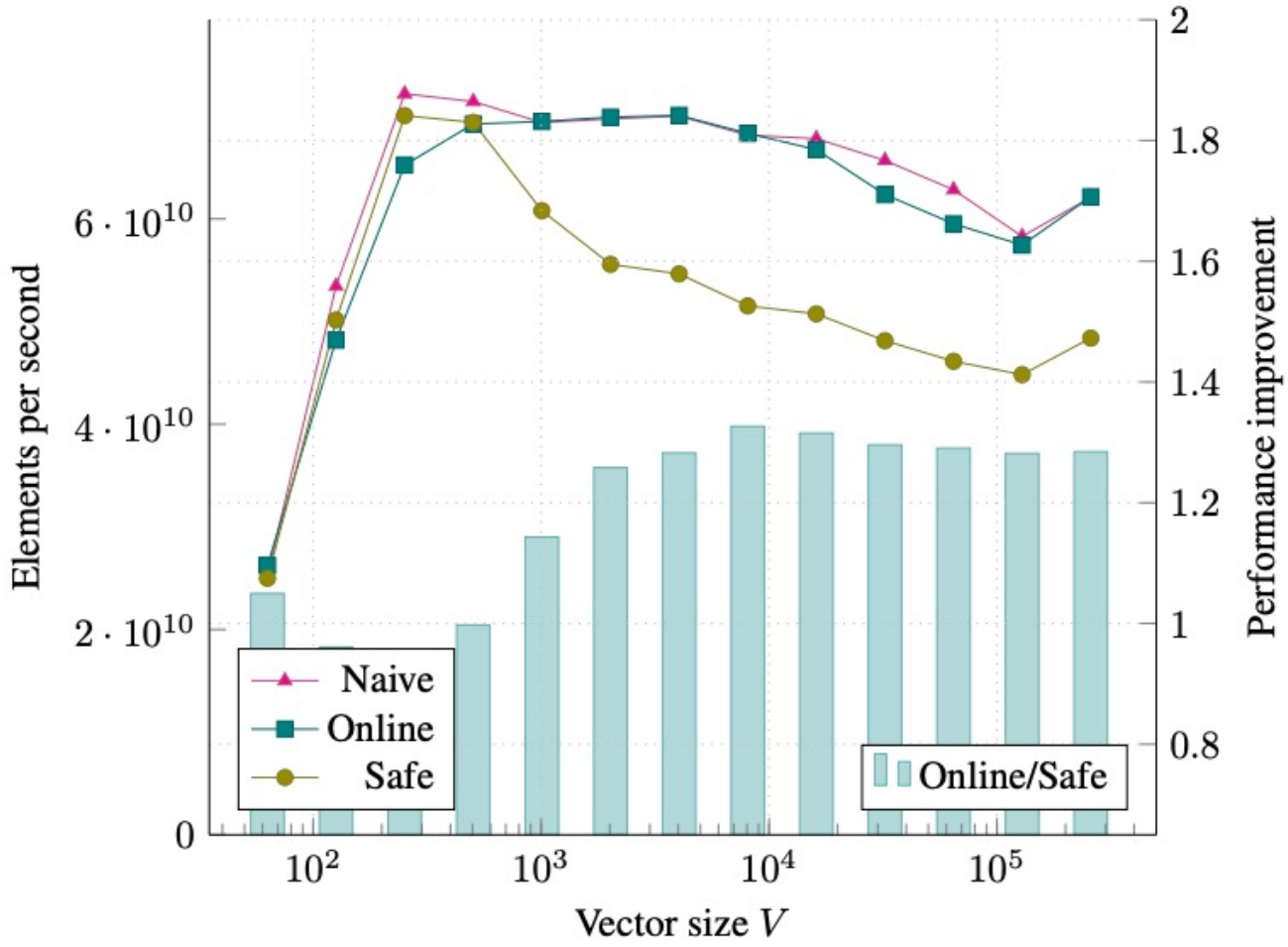



Figure 1: Benchmarking softmax, Tesla V100, fp32, batch size 4000 vectors

Figure from https://arxiv.org/pdf/1805.02867

# Online Softmax

**Theorem 1.** *The lines 1-6 of the algorithm 3 compute* $m_V = \max_{k=1}^{V} x_k$ *and* $d_V = \sum_{j=1}^{V} e^{x_j - m_V}$

*Proof.* We will use a proof by induction.

◊ *Base case*: $V = 1$

$m_1 \leftarrow x_1$ by line 4 of the algorithm 3

$= \max_{k=1}^{1} x_k$

$d_1 \leftarrow e^{x_1 - m_1}$ by line 5 of the algorithm 3

$= \sum_{j=1}^{1} e^{x_j - m_1}$

The theorem holds for $V = 1$.

Figure from https://arxiv.org/pdf/1805.02867

# Online Softmax

**Theorem 1.** *The lines 1-6 of the algorithm 3 compute* $m_V = \max_{k=1}^{V} x_k$ *and* $d_V = \sum_{j=1}^{V} e^{x_j - m_V}$

*Proof.* We will use a proof by induction.

◊ *Inductive step*: We assume the theorem statement holds for $V = S - 1$, that is the lines 1-6 of the algorithm 3 compute $m_{S-1} = \max_{k=1}^{S-1} x_k$ and $d_{S-1} = \sum_{j=1}^{S-1} e^{x_j - m_{S-1}}$. Let's see what the algorithm computes for $V = S$

$$
\begin{aligned}
m_S &\leftarrow \max\left(m_{S-1}, x_S\right) && \text{by line 4 of the algorithm 3} \\
&= \max(\max_{k=1}^{S-1} x_k, x_S) && \text{by the inductive hypothesis} \\
&= \max_{k=1}^{S} x_k \\
d_S &\leftarrow d_{S-1} \times e^{m_{S-1} - m_S} + e^{x_S - m_S} && \text{by line 5 of the algorithm 3} \\
&= \left(\sum_{j=1}^{S-1} e^{x_j - m_{S-1}}\right) \times e^{m_{S-1} - m_S} + e^{x_S - m_S} && \text{by the inductive hypothesis} \\
&= \sum_{j=1}^{S-1} e^{x_j - m_S} + e^{x_S - m_S} \\
&= \sum_{j=1}^{S} e^{x_j - m_S}
\end{aligned}
$$

The inductive step holds as well. □

Figure from https://arxiv.org/pdf/1805.02867

# FLASHATTENTION

Figure from https://slideslive.com/38988230/flashattention-fast-and-memoryefficient-exact-attention-with-ioawareness

# FlashAttention

- One of the most impactful ideas in ML recently
- Even though many people probably don't even know they are using it!
- Introduced at HAET Workshop @ ICML July 2022
- Published @ NeurIPS Dec 2022


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

Tri Dao, Dan Fu ({trid, danfu}@cs.stanford.edu)
7/23/22 HAET Workshop @ ICML 2022

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Ruda, Christopher Ré. Flash Attention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv preprint arXiv:2205.14135.*
https://github.com/HazyResearch/flash-attention.

Figure from https://slideslive.com/38988230/flashattention-fast-and-memoryefficient-exact-attention-with-ioawareness

# FlashAttention

- One of the most impactful ideas in ML recently
- Even though many people probably don't even know they are using it!
- Introduced at HAET Workshop @ ICML July 2022
- Published @ NeurIPS Dec 2022

Figure from https://awaisrauf.github.io/deepCuriosity/Attending-NeurIPS2023

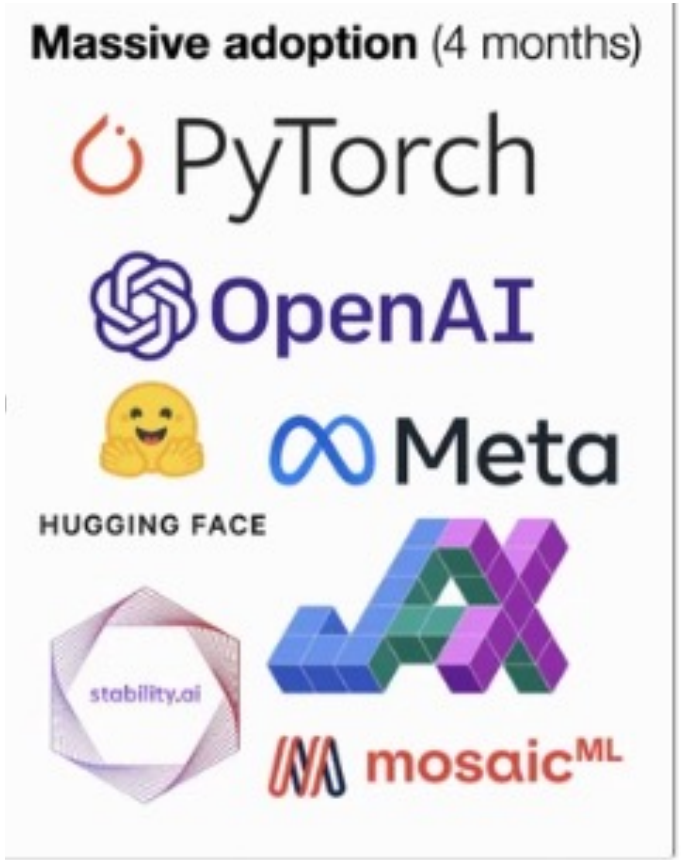
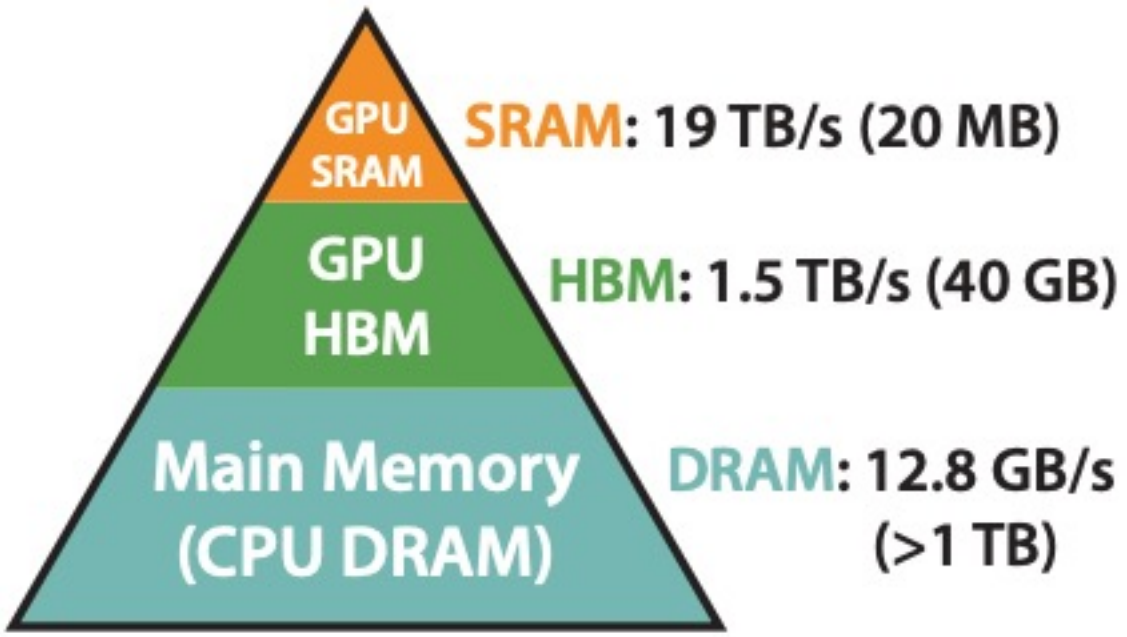



Figure from https://neurips.cc/virtual/2022/poster/54008

# GPU Memory

Memory is arranged hierarchicaly

- GPU SRAM is small, and supports the fastest access
- GPU HBM is larger but with much slower access
- CPU DRAM is huge, but the slowest of all




Memory Hierarchy with Bandwidth & Memory Size

Figure from https://arxiv.org/pdf/2205.14135

# GPU Memory and Transformers

Transformer training is usually memory-bound

- Matrix multiplication takes up 99% of the FLOPS
- But only takes up 61% of the runtime
- Lots of time is wasted moving data around on the GPU
- Instead of doing computation

*Table 1.* Proportions for operator classes in PyTorch.

| Operator class | % flop | % Runtime |
|---|---|---|
| △ Tensor contraction | 99.80 | 61.0 |
| □ Stat. normalization | 0.17 | 25.5 |
| ○ Element-wise | 0.03 | 13.5 |

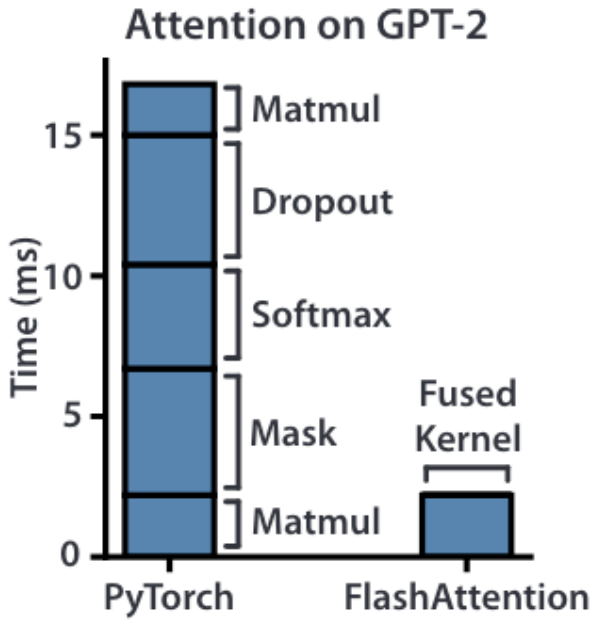
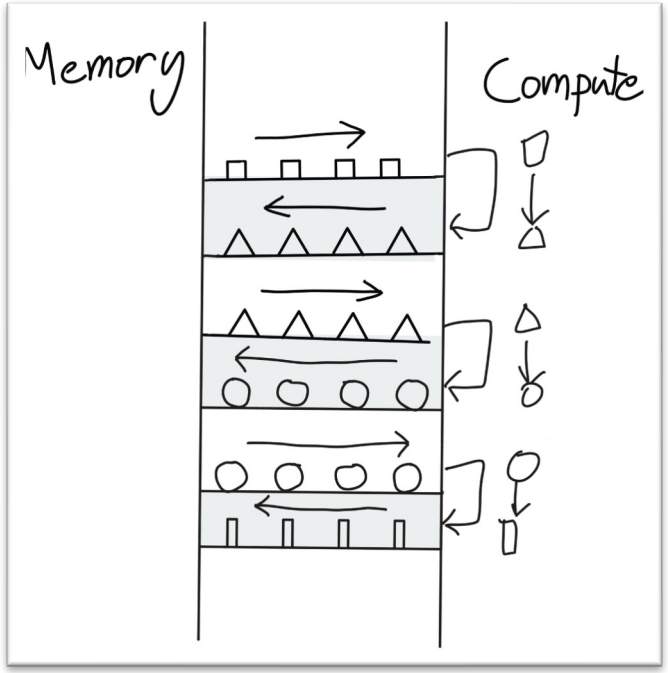



Figure from https://arxiv.org/pdf/2007.00072

Figure from https://arxiv.org/pdf/2205.14135

# Operator Fusion

**Version A:** Usually, we compute a neural network one layer one at a time by moving the layer input to GPU SRAM (fast/small), doing some computation, then returning the output to GPU HBM (slow/large)




**Version B:** Operator fusion instead moves the original input to GPU SRAM (fast/small), does a whole sequence of layer computations without ever touching HBM, and then returns the final layer output to GPU HBM (slow/large)

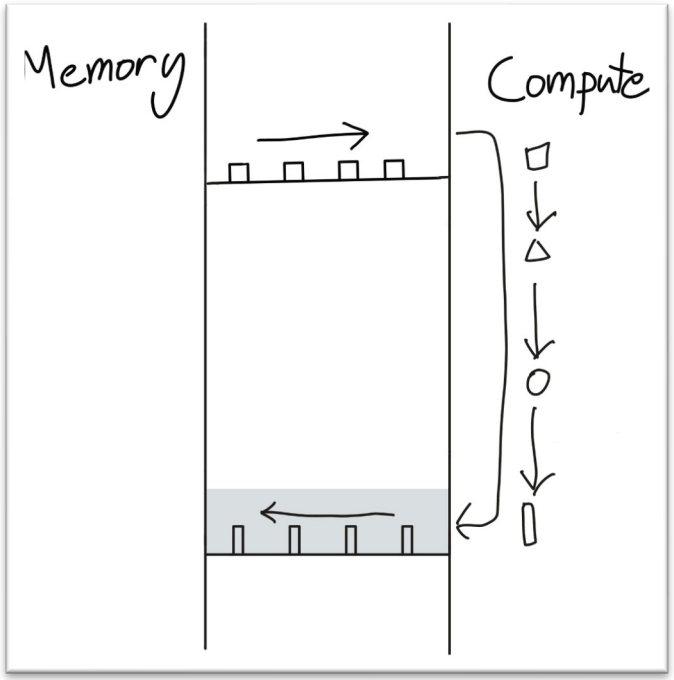



Figure from https://horace.io/brrr_intro.html

# Operator Fusion

**Version A:** Usually, we compute a neural network one layer one at a time by moving the layer input to GPU SRAM (fast/small), doing some computation, then returning the output to GPU HBM (slow/large)

**Version B:** Operator fusion instead moves the original input to GPU SRAM (fast/small), does a whole sequence of layer computations without ever touching HBM, and then returns the final layer output to GPU HBM (slow/large)

Version A is exactly how standard attention is implemented

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

**Algorithm 0** Standard Attention Implementation

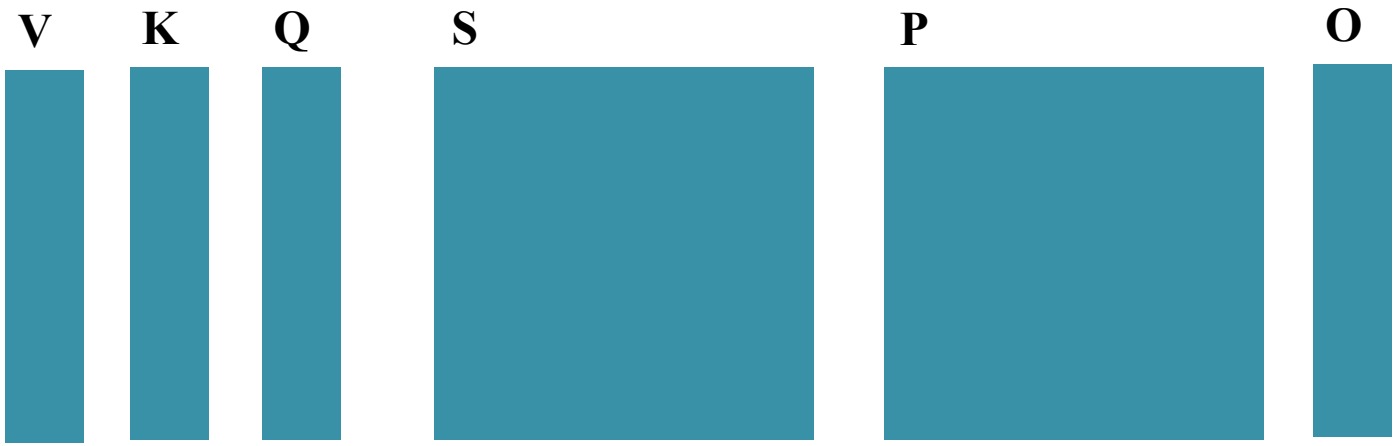
**Require:** Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

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 $\mathbf{O}$.



Figure from https://arxiv.org/pdf/2205.14135

# Standard Attention




Version A is exactly how standard attention is implemented

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$

**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 $\mathbf{O}$.

Figure from https://arxiv.org/pdf/2205.14135

# FlashAttention

- Two key ideas are combined to obtain FlashAttention
- Both are well-established ideas, so the interesting part is how they are put together for attention
  1. **Tiling**: compute the attention weights block by block so that we don't have to load everything into SRAM at once
  2. **Recomputation**: don't ever store the full attention matrix, but just recompute the parts of it you need during the backward pass

# FlashAttention: Tiling

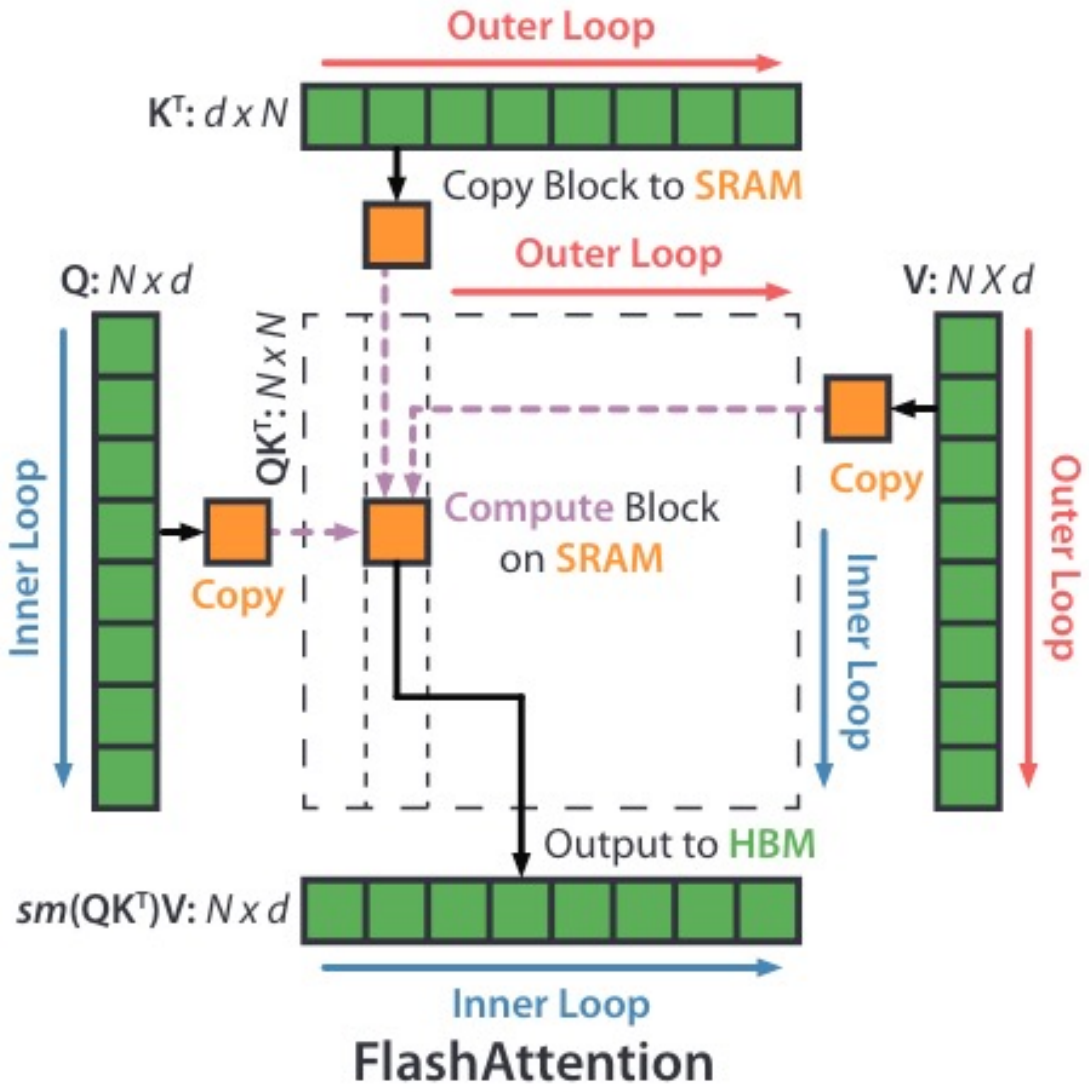



Figure from https://arxiv.org/pdf/2205.14135

# FlashAttention

**Algorithm 1** FlashAttention

**Require:** Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

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

Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_i, \ldots, \ell_{T_r}$ of size $B_r$ each, divide $m$ into $T_r$ blocks $m_1, \ldots, m_{T_r}$ 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.

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

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

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

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  **end for**

**end for**

Return $\mathbf{O}$.

# FlashAttention: Tiling

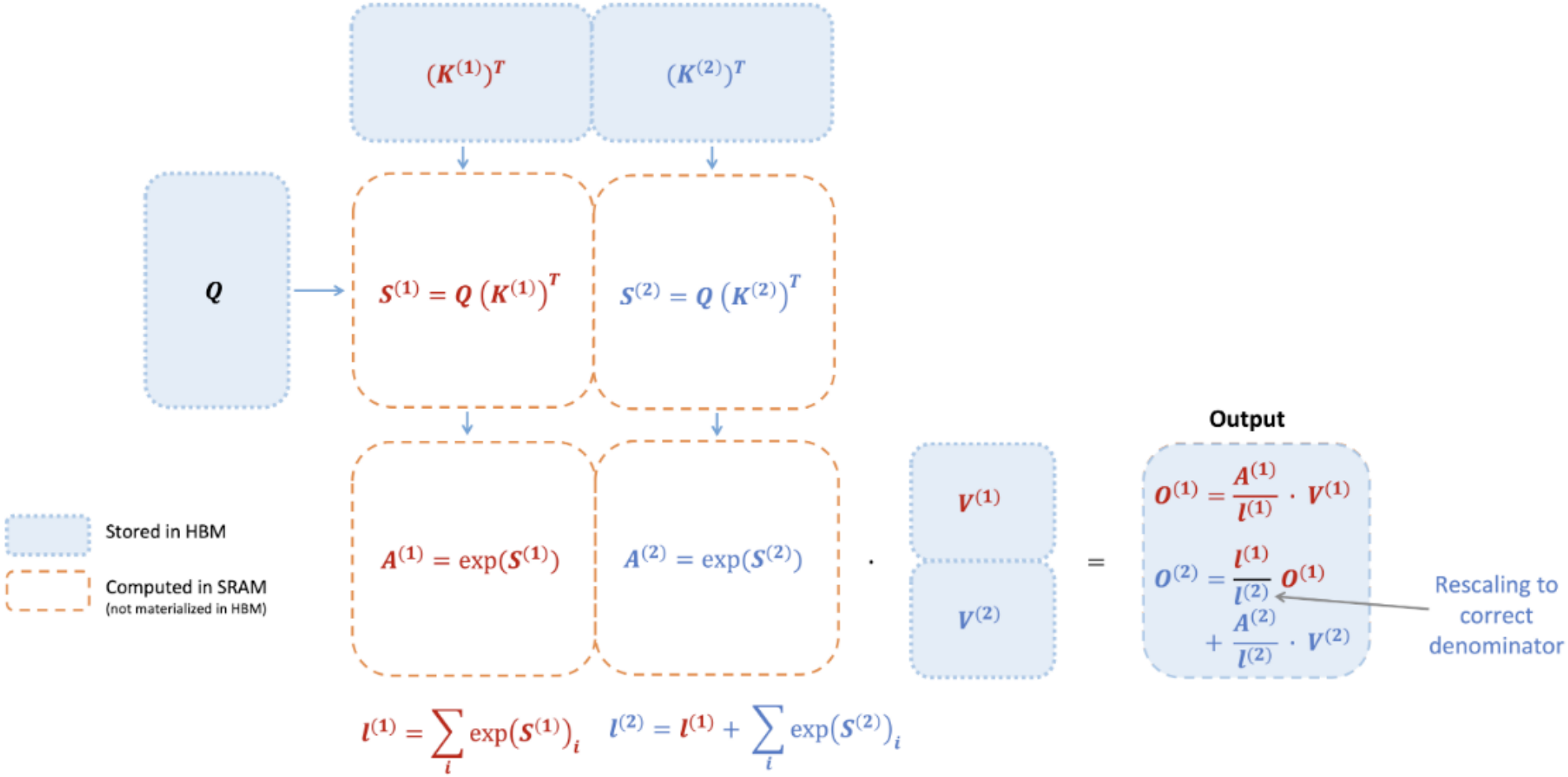



Figure from http://arxiv.org/abs/2307.08691

# FlashAttention: Tiling

One of the key challenges is how to compute the softmax since it is inherently going to require working with multiple blocks

For numerical stability, the softmax of vector $x \in \mathbb{R}^B$ is computed as:

$$m(x) := \max_i \; x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}, \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix} \in \mathbb{R}^{2B}$ as:

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Therefore if we keep track of some extra statistics $(m(x), \ell(x))$, we can compute softmax one block at a time.[2]



Figure from https://arxiv.org/pdf/2205.14135

# Reconstruction for a Feed-Forward MLP

# FlashAttention: Reconstruction

# FlashAttention

**Algorithm 1** FlashAttention

**Require:** Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

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

Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_i, \ldots, \ell_{T_r}$ of size $B_r$ each, divide $m$ into $T_r$ blocks $m_1, \ldots, m_{T_r}$ 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.

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

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

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

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  **end for**

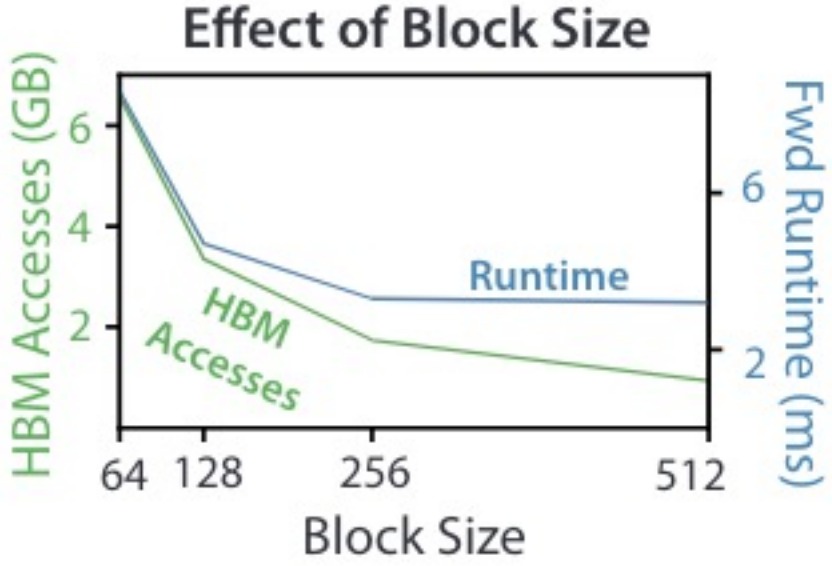
**end for**

Return $\mathbf{O}$.

# FlashAttention: Results

- The algorithm is performing exact attention, so we see no reduction in perplexity or quality of the model
- The key metric is runtime

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



# FlashAttention: Results

- The algorithm is performing exact attention, so we see no reduction in perplexity or quality of the model
- The key metric is runtime

| 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×)** |