Stony Brook University

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

Dec 3, 2025

Shang-Jui Ray Kuo

FAR BEYOND

Stony Brook University

Hardware-Aware Efficient Training (HAET) Workshop at ICML 2022

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

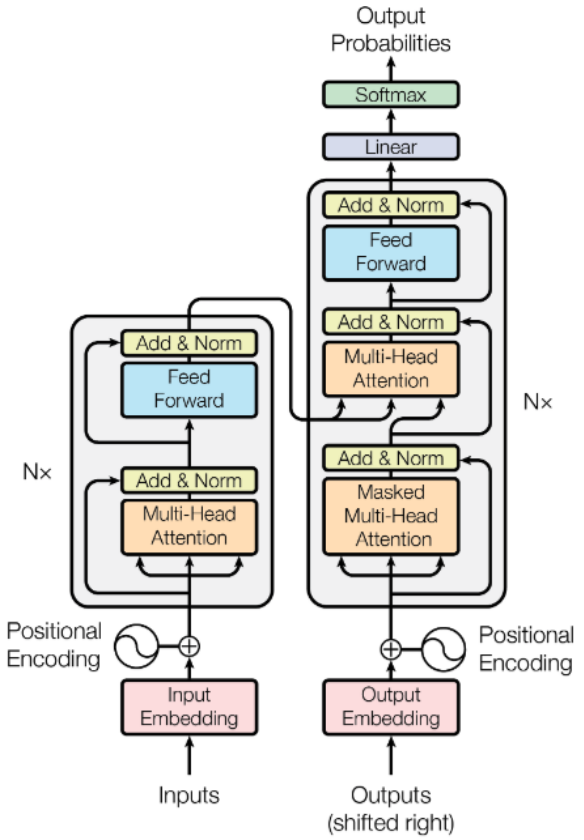
June 24, 2022

FAR BEYOND

Stony Brook University

What Kind of Work Is This?

- Accelerates **attention kernels** on GPUs
- Focuses on the **GPU memory hierarchy** and **HBM traffic**, not just FLOPs
- Impactful because:
  - Self-attention is a core operator in almost all modern Transformers.
  - Wall-clock time and memory are real bottlenecks for long sequences.



Attention Is All You Need

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

FAR BEYOND

Stony Brook University

Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

Stony Brook University

Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

FAR BEYOND

Before the FlashAttention Algorithm

- We need:
  - A basic picture of the **GPU memory hierarchy**.
  - How a GPU **kernel** executes with this hierarchy.
  - The idea of **compute-bound vs memory-bound** kernels.
  - How **kernel fusion** reduces unnecessary memory traffic.
  - How the **standard attention kernel** is implemented before this work.

Stony Brook University

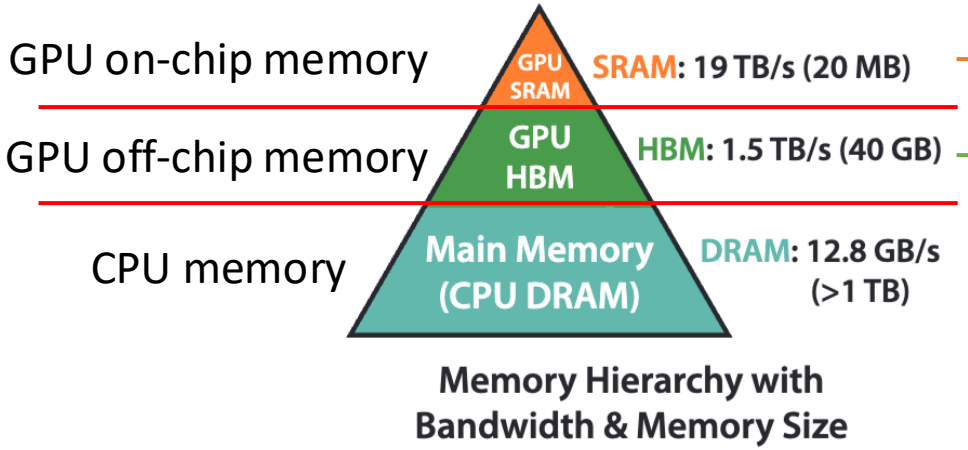
Before the FlashAttention Algorithm

- We need:
  - A basic picture of the **GPU memory hierarchy**.
  - How a GPU **kernel** executes with this hierarchy.
  - The idea of **compute-bound vs memory-bound** kernels.
  - How **kernel fusion** reduces unnecessary memory traffic.
  - How the standard attention kernel is implemented before this work.

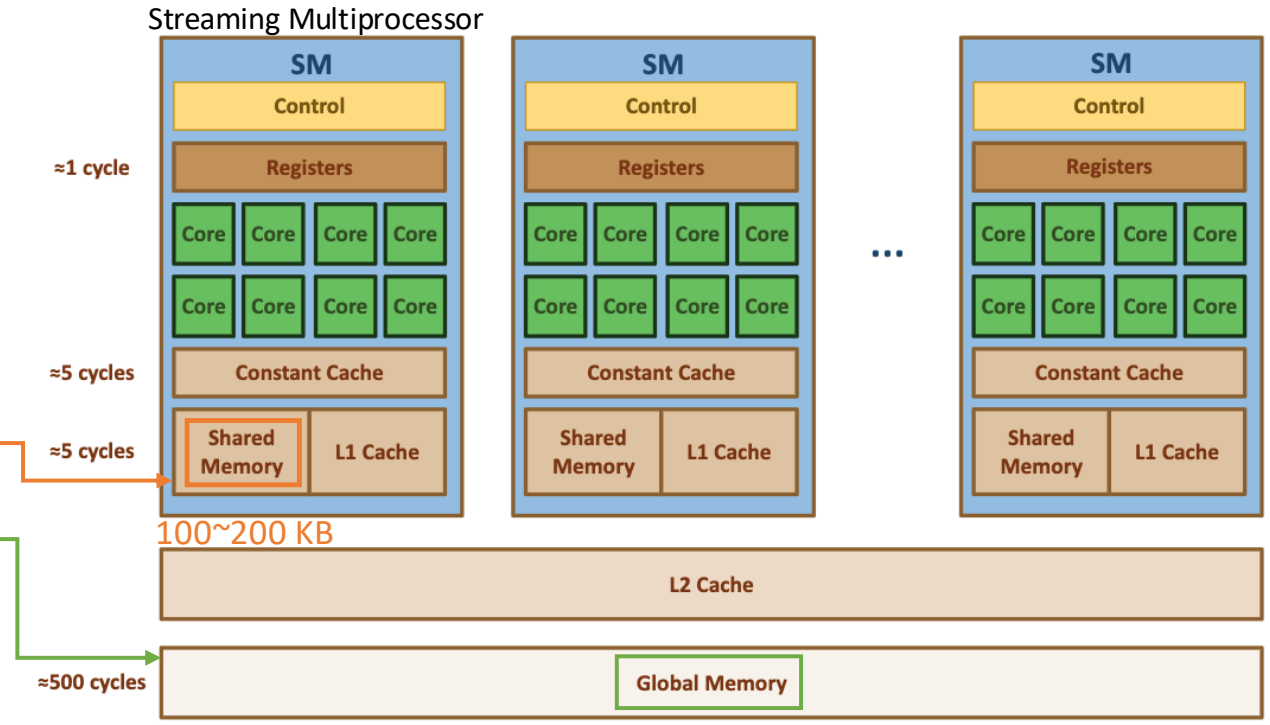
Stony Brook University

GPU Memory Hierarchy

- Multiple levels, **smaller = faster**
  - CPU DRAM
  - GPU **HBM** (DRAM on the GPU)
  - Registers, on-chip **SRAM**



Memory Hierarchy with Bandwidth & Memory Size



Slide credit: Prof. Izzat El Hajj

FAR BEYOND

Stony Brook University

NVIDIA Data-Center GPU Spec Comparison

Model	Architecture	CUDA Cores	Tensor Cores	SM Count	Shared Mem / SM	Registers / SM	Memory	Memory Bandwidth	HBM Type	NVLink Generation
Tesla P100	Pascal	3,584	–	56	64 KB	65,536 32-bit registers	16 GB HBM2	~732 GB/s	HBM2	NVLink 1
Tesla V100	Volta	5,120	640	80	96 KB	65,536 32-bit registers	16–32 GB HBM2	~900 GB/s	HBM2	NVLink 2
A100	Ampere	6,912	432	108	164 KB	65,536 32-bit registers	40–80 GB HBM2e	~1.6–2.0 TB/s	HBM2e	NVLink 3
H100 (SXM)	Hopper	14,592	456	132	228 KB	64,000 32-bit registers (architectural)	80 GB HBM3	~3.35 TB/s	HBM3	NVLink 4
B200 (highest tier)	Blackwell	N/A (not public)	5th-gen Tensor Cores	N/A (not public)	N/A	N/A	192 GB HBM3e	~8 TB/s	HBM3e	NVLink 5

FAR BEYOND

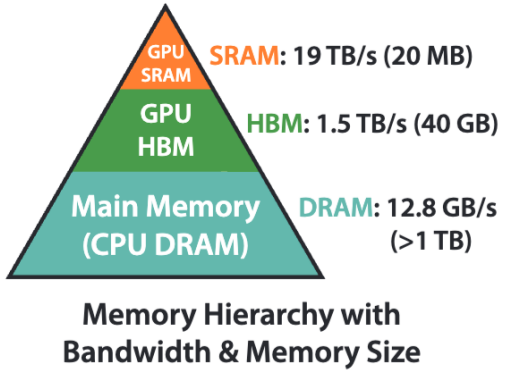
Stony Brook University

Before the FlashAttention Algorithm

- We need:
  - A basic picture of the **GPU memory hierarchy**.
  - How a GPU **kernel** executes with this hierarchy.
  - The idea of **compute-bound vs memory-bound** kernels.
  - How **kernel fusion** reduces unnecessary memory traffic.
  - How the standard attention kernel is implemented before this work.

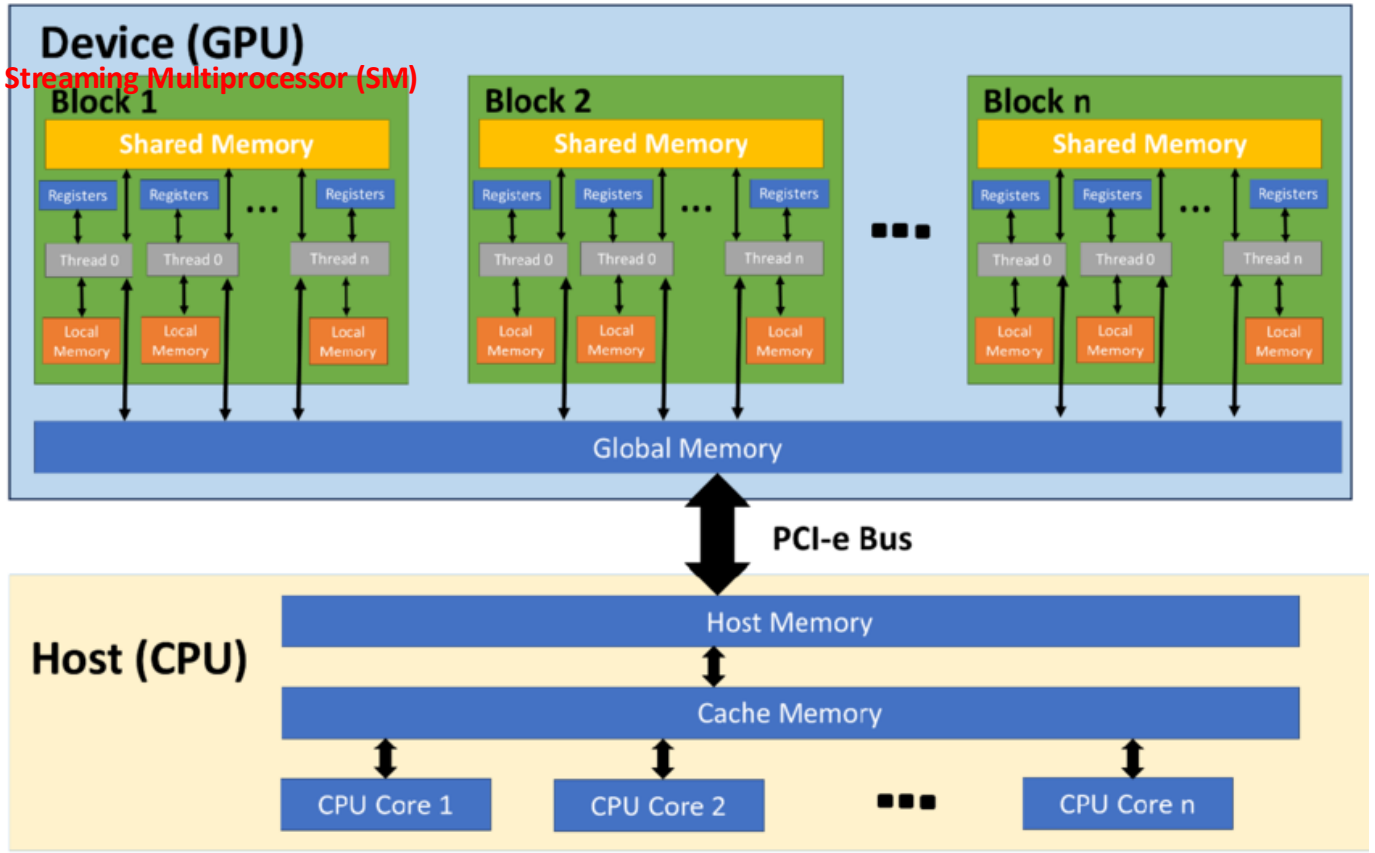
Stony Brook University

GPU Execution Model and Kernels



Memory Hierarchy with Bandwidth & Memory Size

- A **kernel** = one operation launched on the GPU
- Each kernel:
  1. Loads its inputs from HBM to registers/SRAM
  2. Computes
  3. Writes outputs back to HBM
- Runtime of a kernel:
  - Td: Bytes moved between SRAM and HBM (IO)
  - Tc: computation time
  - Ideally: T = max(Tc, Td)



Optimizing MRI Data Processing by exploiting GPU Acceleration for Efficient Image Analysis and Reconstruction

FAR BEYOND

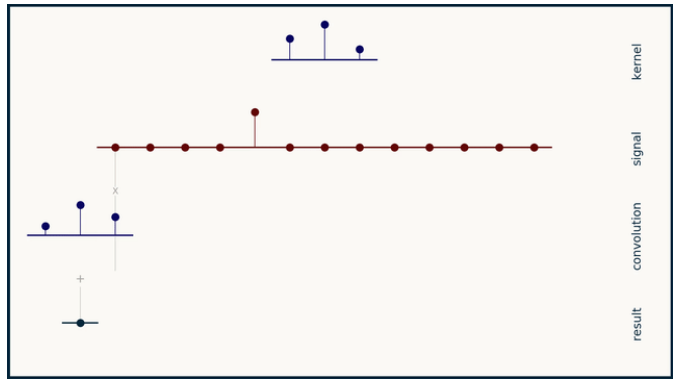
Stony Brook University

Before the FlashAttention Algorithm

- We need:
  - A basic picture of the **GPU memory hierarchy**.
  - How a GPU **kernel** executes with this hierarchy.
  - The idea of **compute-bound vs memory-bound** kernels.
  - How **kernel fusion** reduces unnecessary memory traffic.
  - How the standard attention kernel is implemented before this work.

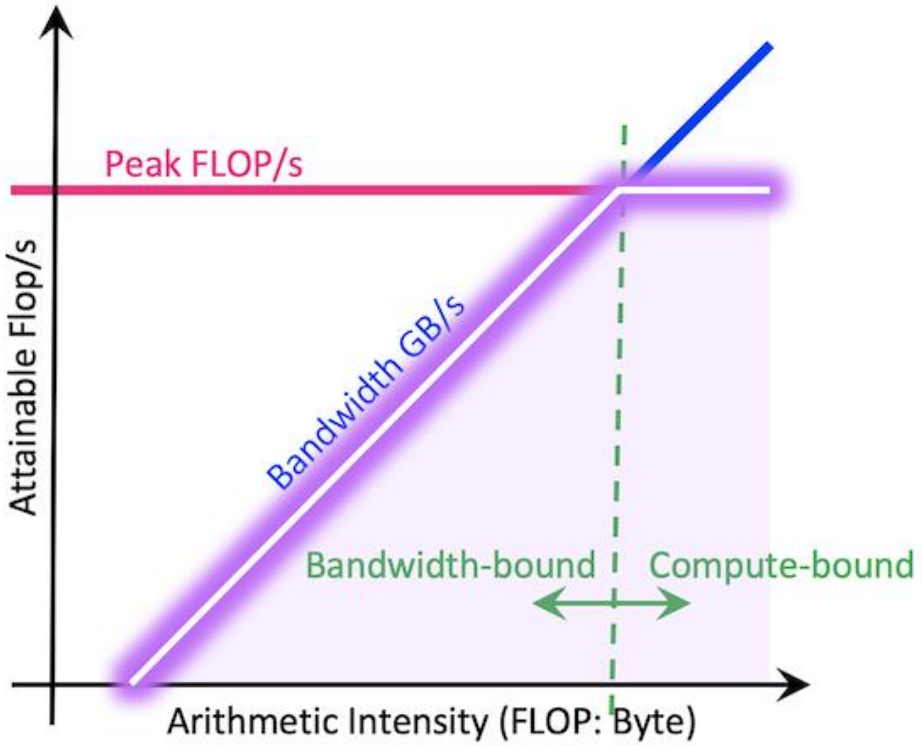
Stony Brook University

Compute-Bound vs Memory-Bound



https://brandonrohrer.com/convolution_one_d.html

- **Compute-bound** kernels (Tc > Td):
  - Time dominated by arithmetic.
  - Example: large dense matmul, large convolution.
- **Memory-bound** kernels (Td > Tc):
  - Time dominated by HBM traffic.
  - Examples: softmax, layernorm, dropout, most elementwise ops.
- **Arithmetic intensity** = FLOPs/bytes moved
- Many Transformer ops, including parts of attention, are **memory-bound**



https://docs.nersc.gov/tools/performance/roofline/

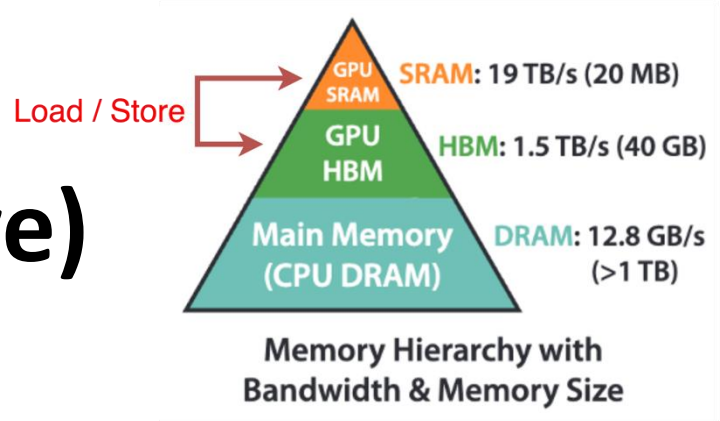
FAR BEYOND

Stony Brook University

Before the FlashAttention Algorithm

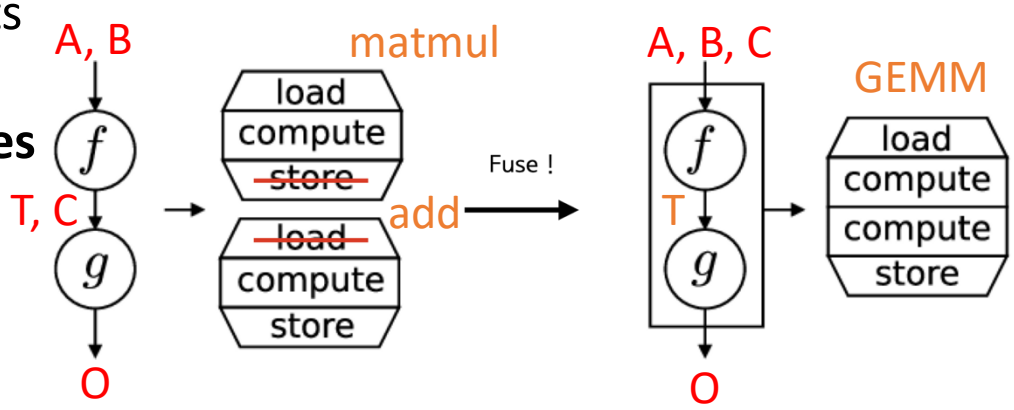
- We need:
  - A basic picture of the **GPU memory hierarchy**.
  - How a GPU **kernel** executes with this hierarchy.
  - The idea of **compute-bound vs memory-bound** kernels.
  - How **kernel fusion** reduces unnecessary memory traffic.
  - How the standard attention kernel is implemented before this work.

Kernel Fusion (PyTorch Perspective)



- Naïve execution:
  - Each op (matmul, add, softmax, dropout, …) is its own kernel.
  - Each kernel **reads** its inputs from HBM and **writes** results back.
- **Fusion:** combine operations so a tensor is
  - Read once from HBM
  - Processed through several ops on-chip
  - Written once back to HBM
- Good for inference, limited in training

Example: O = A x B + C



https://github.com/huggingface/transformers/issues/13845

Load A, B
Compute T = A x B
Store T
Load T, C
Compute O = T + C
Store O

Load A, B, C
Compute O = A x B + C
Store O

Stony Brook University

Before the FlashAttention Algorithm

- We need:
  - A basic picture of the **GPU memory hierarchy**.
  - How a GPU **kernel** executes with this hierarchy.
  - The idea of **compute-bound vs memory-bound** kernels.
  - How **kernel fusion** reduces unnecessary memory traffic.
  - How the **standard attention kernels** are implemented before this work.

Standard Attention and Its IO

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Given input sequences $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ where N is the sequence length and d is the head dimension, we want to compute the attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$:

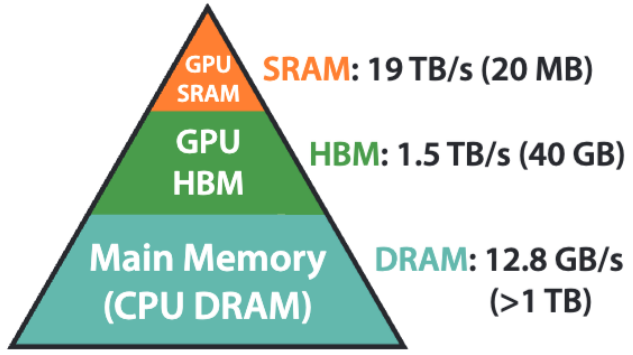
$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

where softmax is applied row-wise. Often N >> d (e.g., for GPT2, N = 1024 and d = 64)

Algorithm 0 Standard Attention Implementation

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

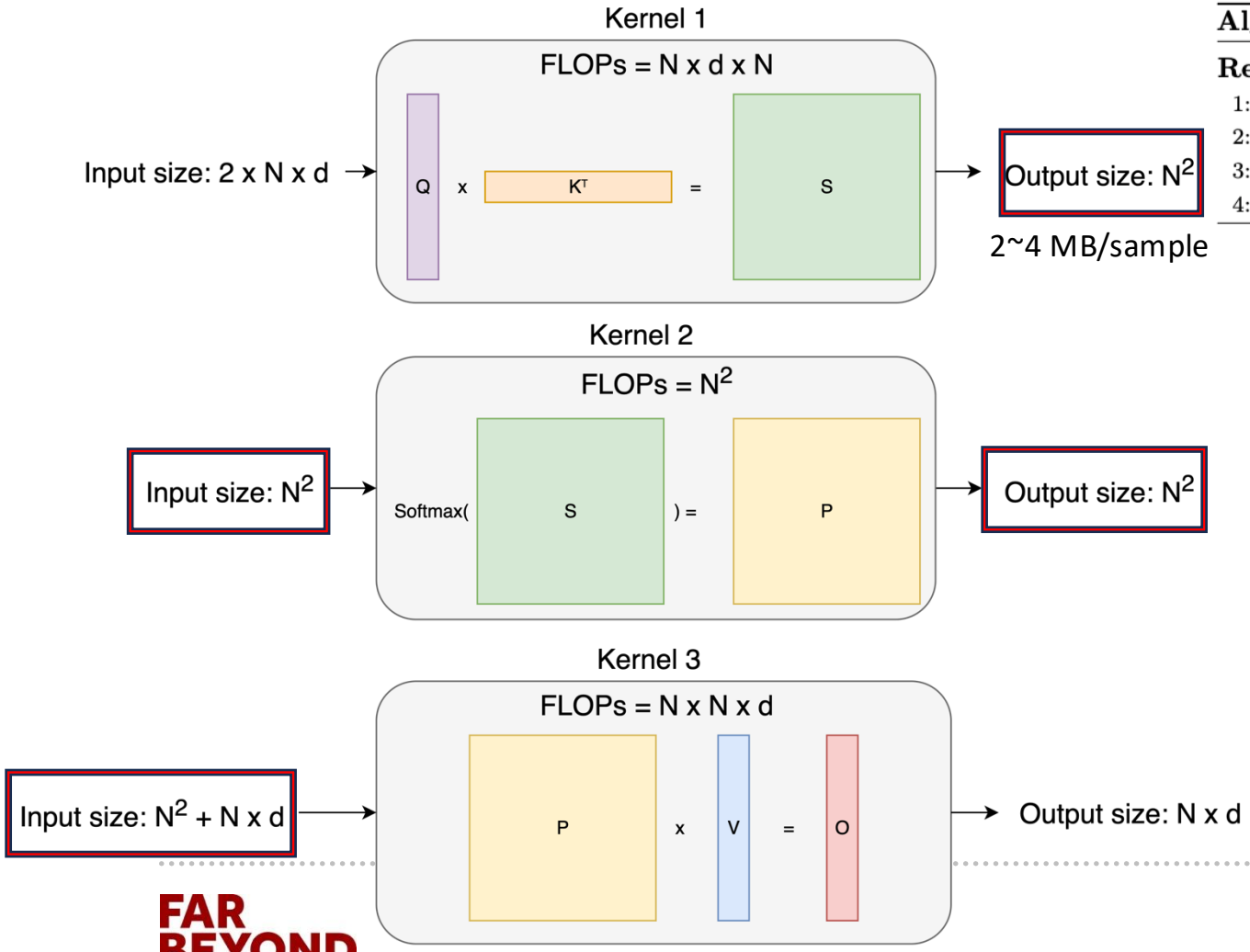
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.



Memory Hierarchy with Bandwidth & Memory Size

Stony Brook University

Standard Attention Kernel Analysis

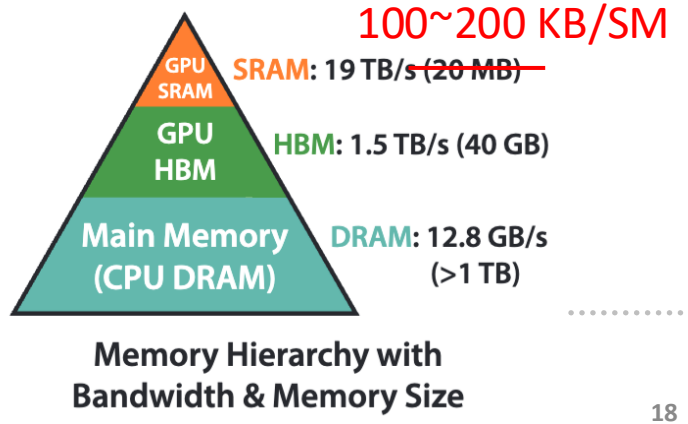


Algorithm 0 Standard Attention Implementation

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 $\mathbf{O}$.

- Memory-bound kernels
- IO Complexity: $\Omega(Nd+N^2)$
- Can we fuse these kernels?

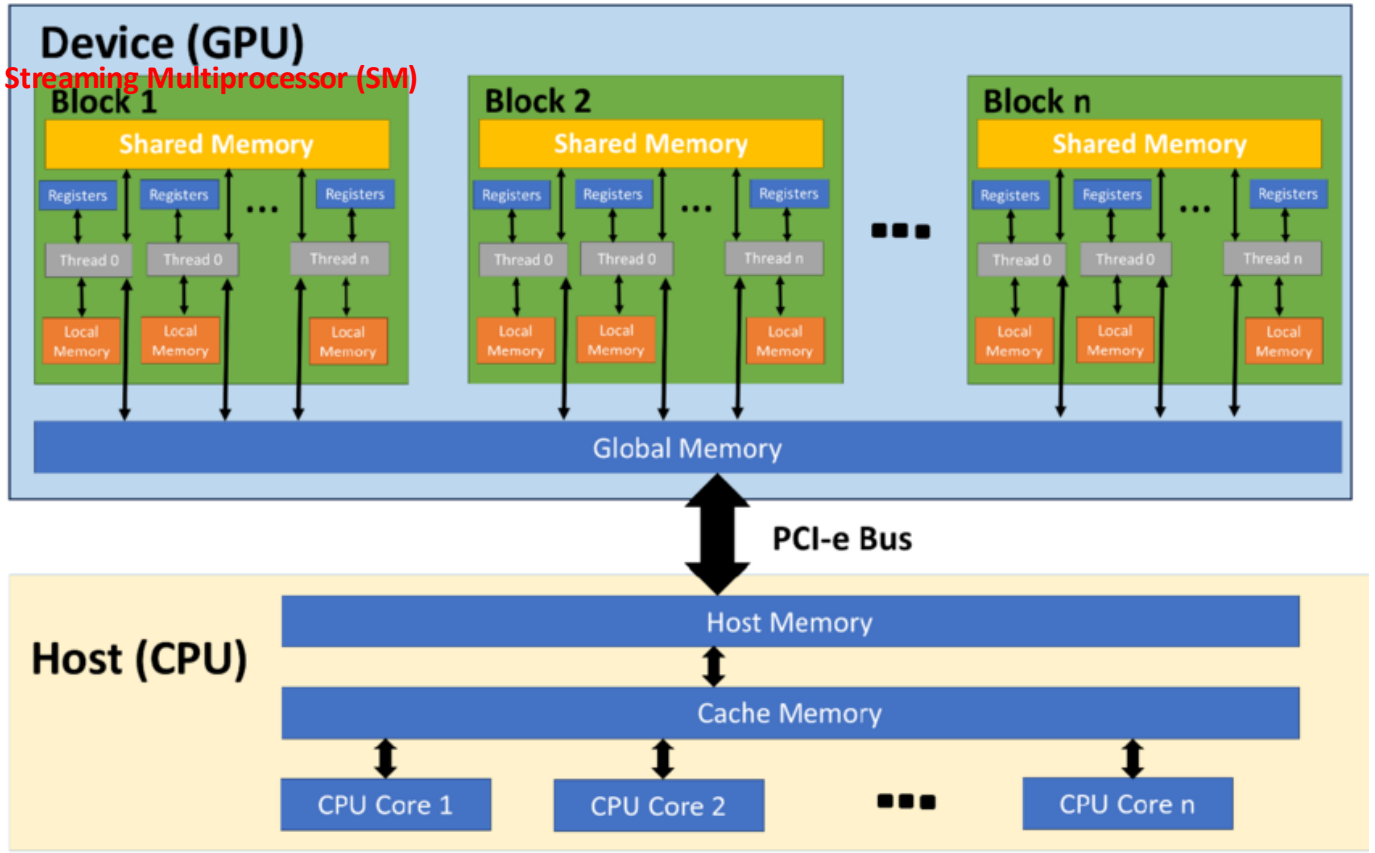


Memory Hierarchy with Bandwidth & Memory Size

FAR BEYOND

GPU Execution Model and Kernels (Review)

- A **kernel** = one operation launched on the GPU
- Each kernel:
  1. Loads its inputs from HBM to registers/SRAM
  2. Computes
  3. Writes outputs back to HBM
- Runtime of a kernel:
  - Tc: FLOPs (compute)
  - Td: Bytes moved between SRAM and HBM (IO)
  - Ideally: T = max(Tc, Td)



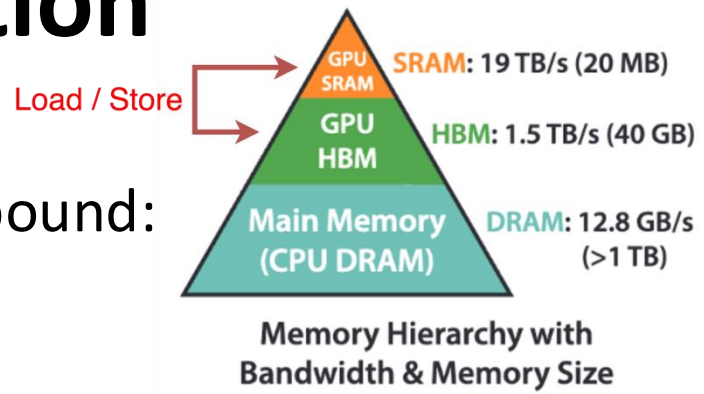
Optimizing MRI Data Processing by exploiting GPU Acceleration for Efficient Image Analysis and Reconstruction

Stony Brook University

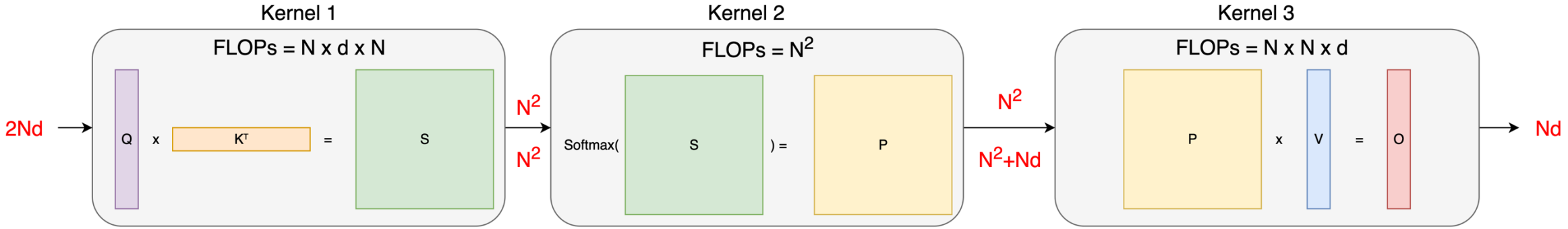
Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

Main Bottleneck of Standard Attention

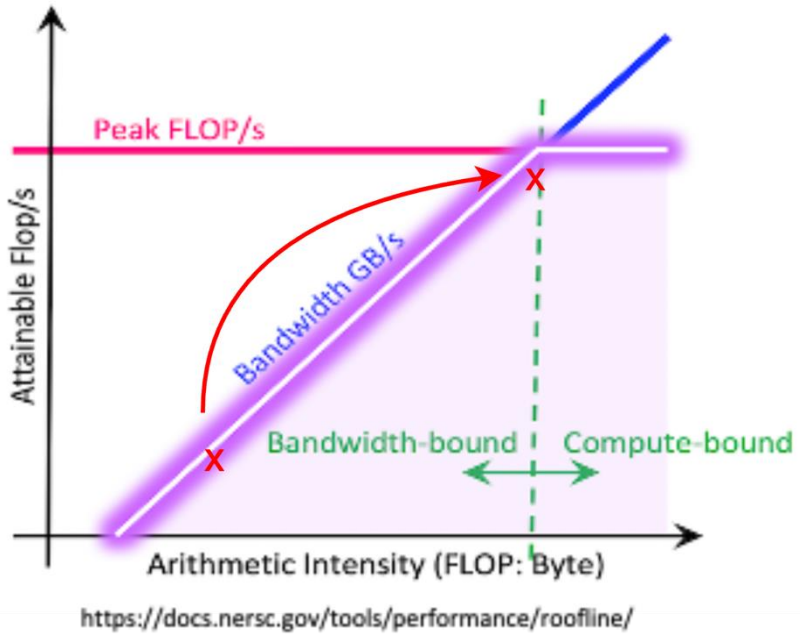


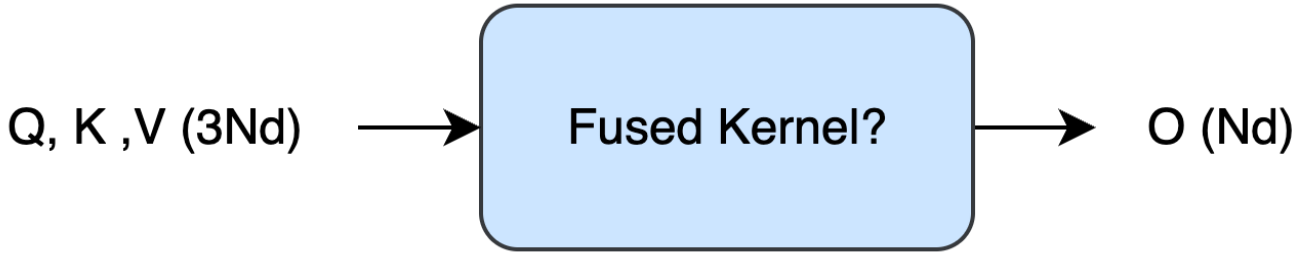
- Many attention steps are **memory-bound**, not compute-bound:
  - S and P are size N×N, much larger than Q, K, and V (N×d).
- The attention kernel's wall-clock time is dominated by:
  - **Reading and writing N^2 elements** (S and P) between HBM and on-chip memory.
- This is the core bottleneck FlashAttention targets.



FlashAttention Goal

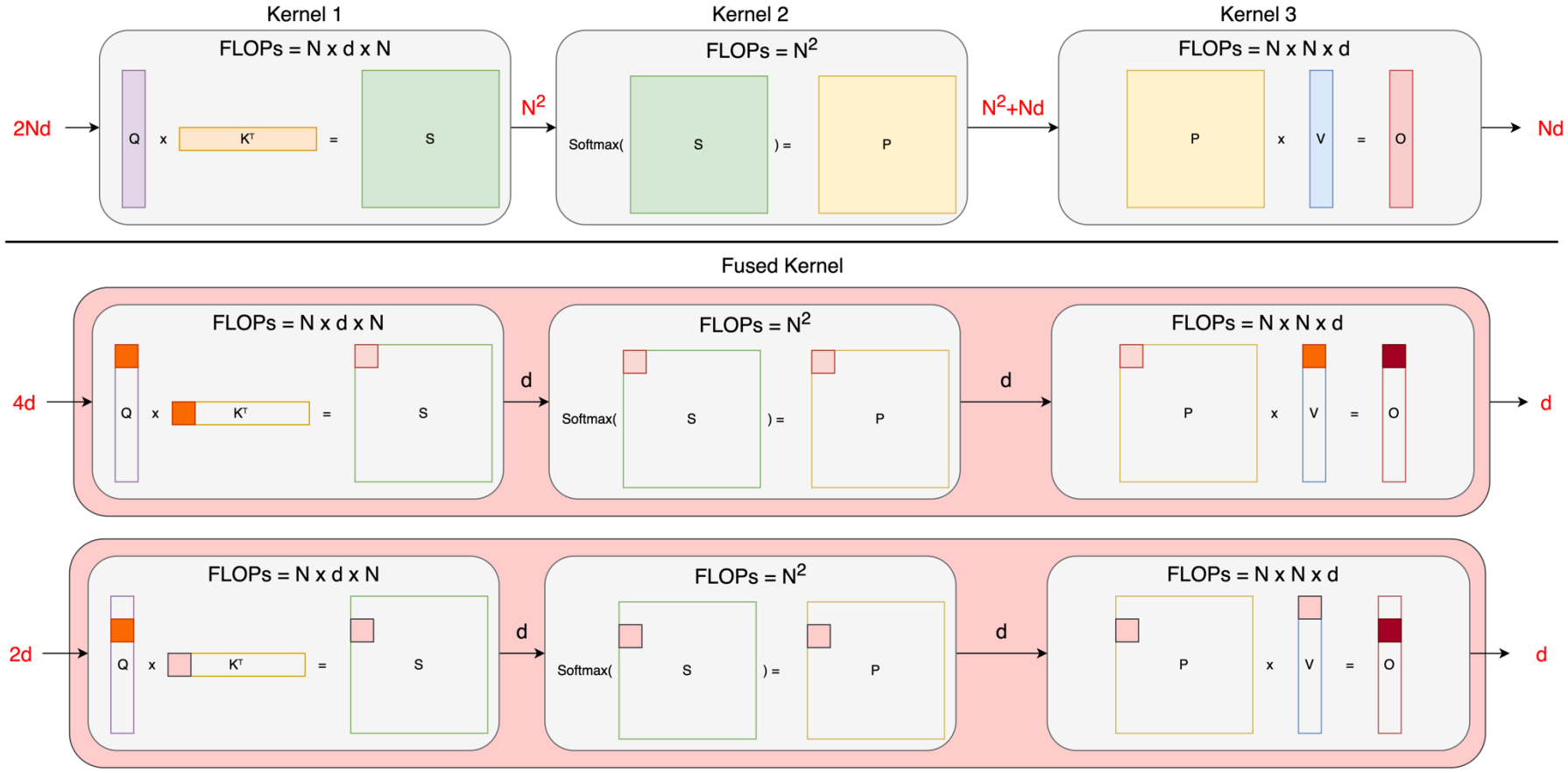
- Design a fused kernel that
  - Has the same numerical result as the standard implementation
  - Much **less HBM traffic**.
- High-level question:
  - Can we **reorder** the computation to keep most work on-chip, without changing the mathematical result?





Stony Brook University

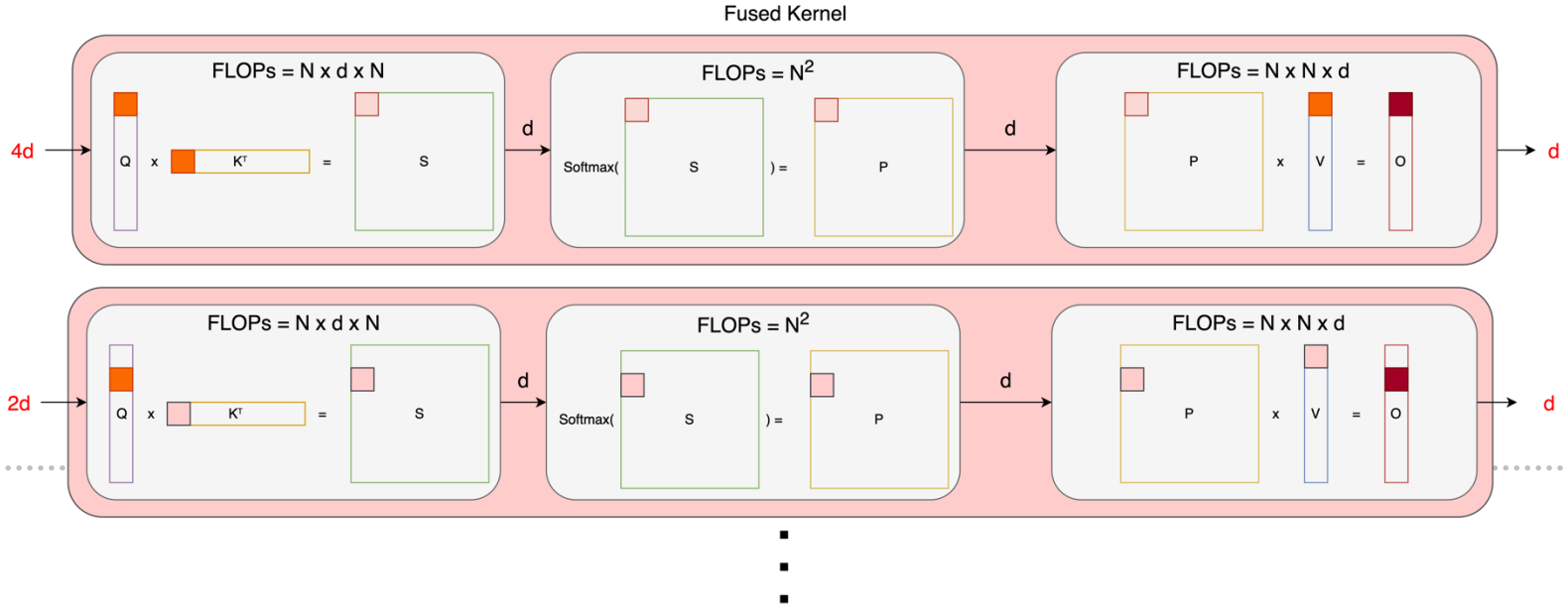
Natural Idea: Tiling



FAR BEYOND

Why Naive Tiling Does Not Work

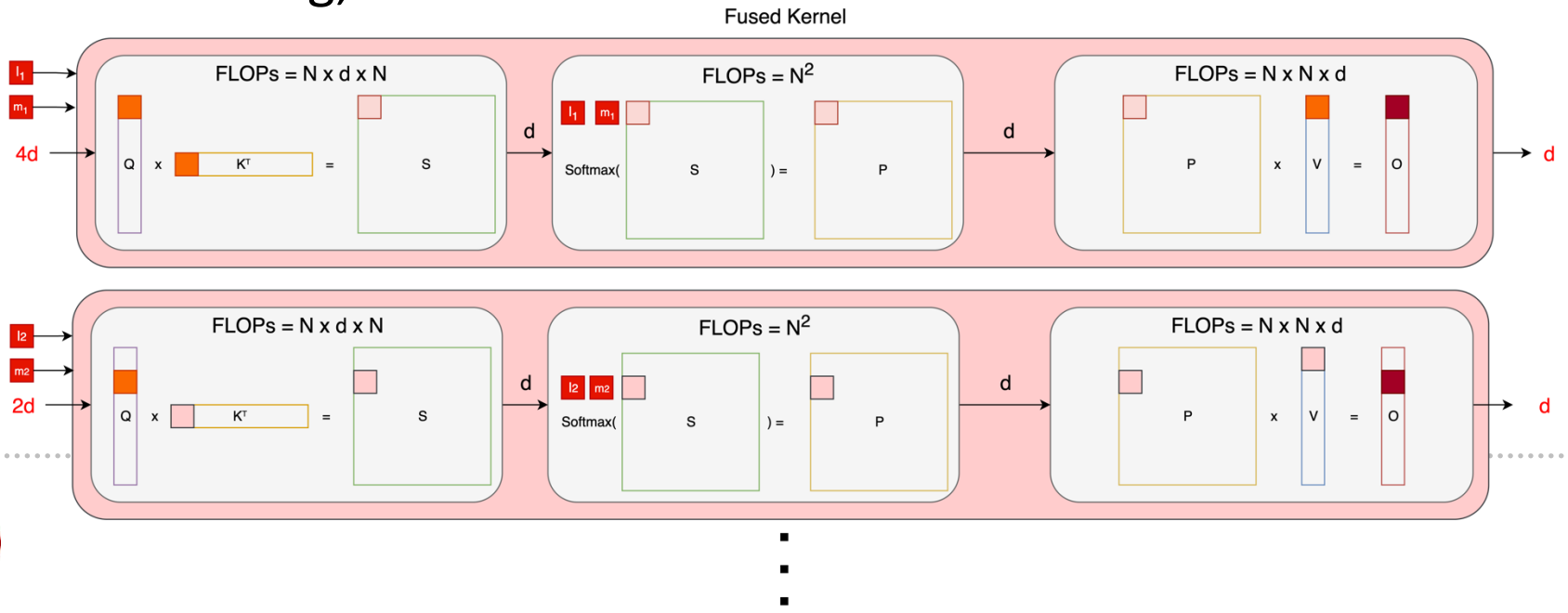
- **Issue 1: Softmax needs the whole row**
- **Issue 2: Backward needs S or P**
  - Gradients dQ, dK, dV depend on S/P for the whole N×N matrix.
  - If we do not store S or P during forward, naive backward has no information.



Stony Brook University

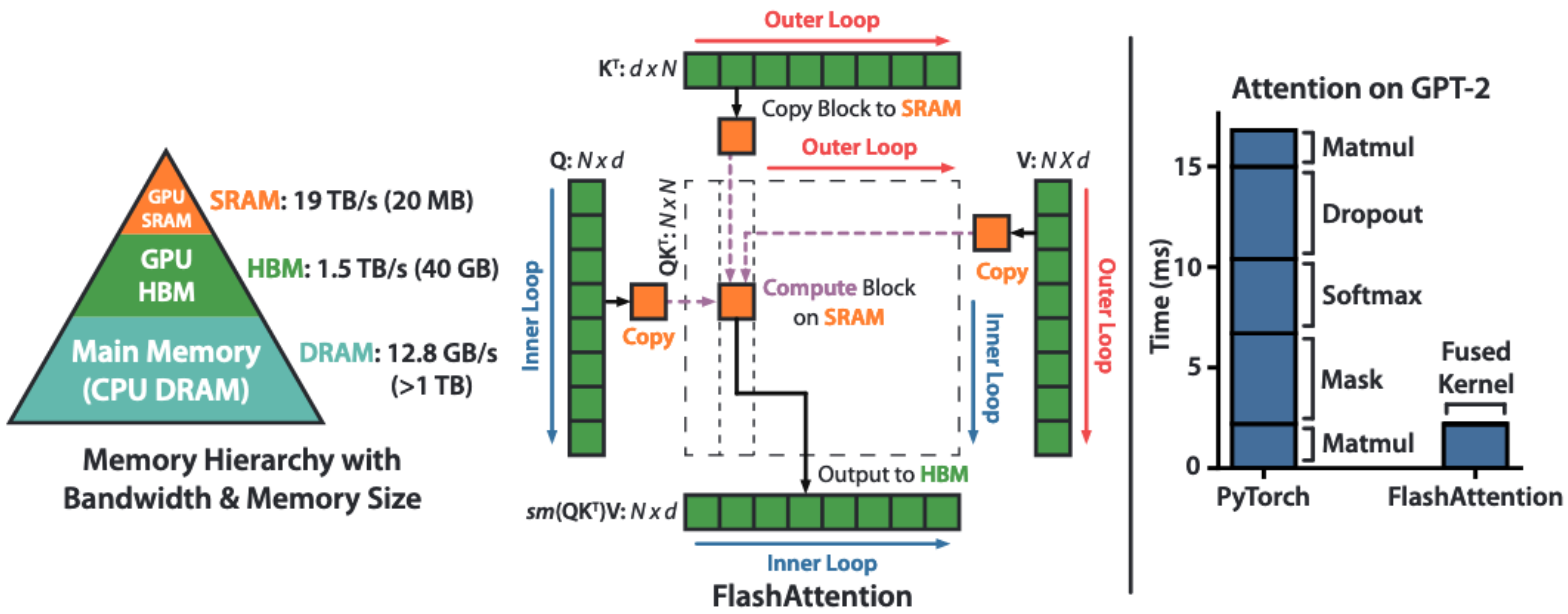
Two Key Ideas to Fix Tiling

- FlashAttention introduces **two additional ideas** on top of tiling
  - **Online softmax**: compute softmax row-by-row in blocks
  - **Recomputation** for backward: recompute S and P on-the-fly
- Together with tiling, these ideas make an **IO-efficient fused attention kernel**.



FAR BEYOND

FlashAttention Core Idea



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

Forward + backward runtime of standard attention and FlashAttention for GPT-2 on A100 GPU. HBM access is the primary factor affecting runtime.

FAR BEYOND

Stony Brook University

Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

Stony Brook University

Method Details

- Tiling
- Online Softmax
- Recomputation
- Block-Sparse Extension

Stony Brook University

Method Details

- Tiling
- Online Softmax
- Recomputation
- Block-Sparse Extension

Tiling

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 $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

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 $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

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

for $1 \le j \le T_c$ **do** Outer loop

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

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

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

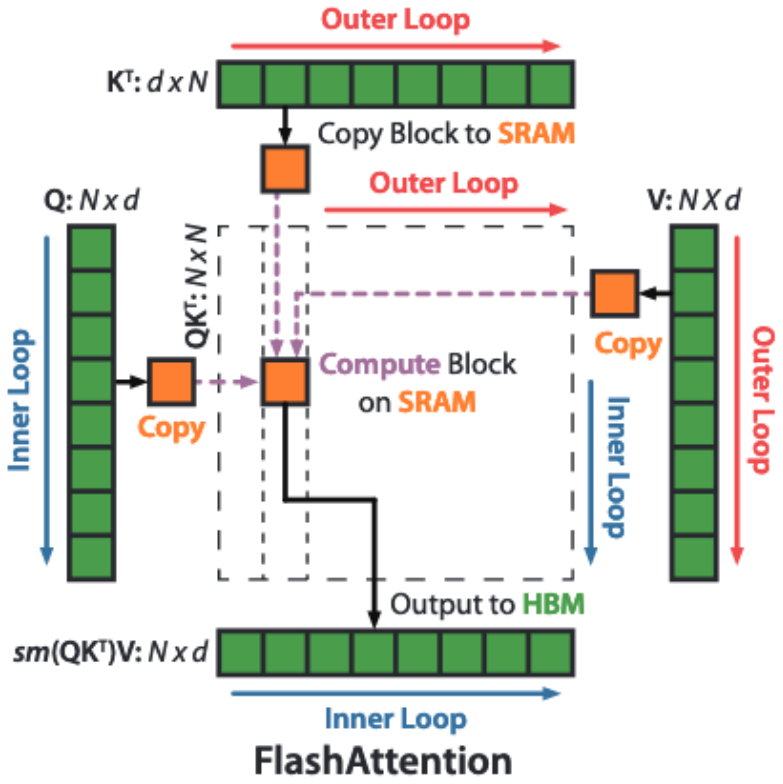
    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

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

  end for

end for

Return $\mathbf{O}$.

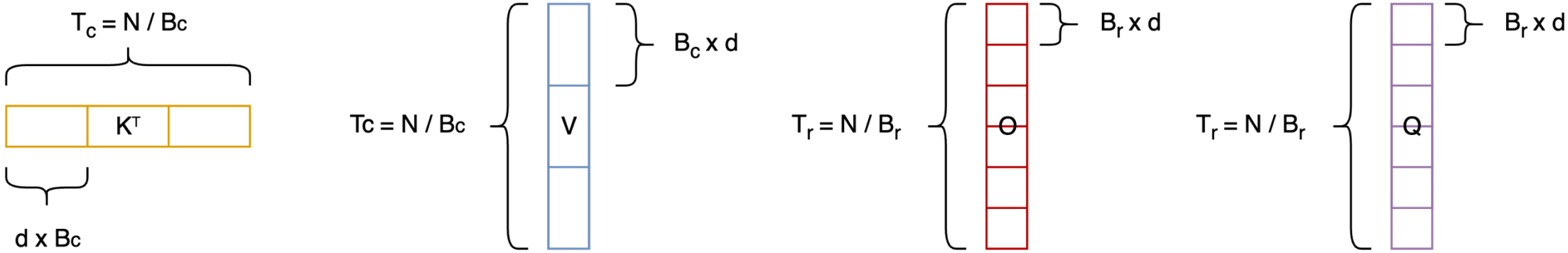


Tiling – Define Block Size

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

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



Stony Brook University

Tiling Visualization – Outer Loop Iteration 1

- Inner loop: repeat T_r times

FAR BEYOND

Stony Brook University

Tiling Visualization – Outer Loop Iteration 2

- Inner loop: repeat T_r times

FAR BEYOND

Stony Brook University

FlashAttention IO Complexity Analysis

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

$T_c = \lceil \frac{N}{B_c} \rceil \qquad T_r = \lceil \frac{N}{B_r} \rceil$

$T_c = 4dN / M$

- Each inner loop iteration: $3dB_r + 4B_r$
- T_r iterations for inner loop: $T_r \times (3dB_r + 4B_r) + 3dB_c$
  - $3dN + 4N + 3dB_c$
- T_c outer loop iterations: $T_c \times (3dN + 4N + 3dB_c)$
  - $12N^2d^2 / M + 16N^2d / M + 3Nd$
- HBM access complexity: $O(N^2d^2M^{-1})$

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.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: 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.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: 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), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: 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.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

FAR BEYOND

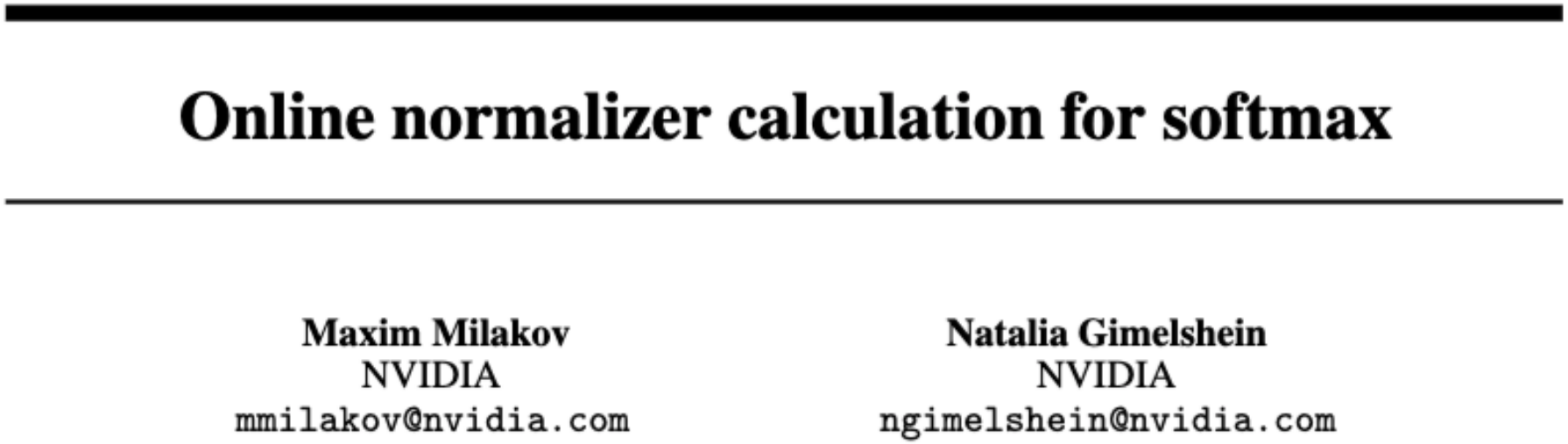
Stony Brook University

Method Details

- Tiling
- Online Softmax
- Recomputation
- Block-Sparse Extension

Online Softmax

- Jun 2018



Online normalizer calculation for softmax

Maxim Milakov
NVIDIA
mmilakov@nvidia.com

Natalia Gimelshein
NVIDIA
ngimelshein@nvidia.com

Original Softmax

- On real hardware, where the range of numbers represented is limited, line 3 of the Algorithm 1 can overflow or underflow due to the exponent
- Three memory accesses per vector element: two loads and one store

$$y = Softmax(x)$$

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

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**

Safe Softmax

- All major DL frameworks are using this safe version for the Softmax computation (PyTorch, TensorFlow, etc.)
- This results in 4 memory access per vector element overall

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

Algorithm 2 Safe softmax

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do**
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: **end for**

Safe Softmax with Online Normalizer Calculation

- Reduce memory accesses from 4 down to 3 per vector element

Algorithm 2 Safe softmax

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do**
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: **end for**

Algorithm 3 Safe softmax with online normalizer calculation

1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: **for** $j \leftarrow 1, V$ **do**
4: $\quad m_j \leftarrow \max(m_{j-1}, x_j)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: **end for**

Stony Brook University

Parallel Online Normalizer Calculation

- Modern computing devices allow running multiple threads concurrently; We need to have a parallel version of the algorithm to fully utilize devices.
- The operation $\bigoplus$ is associative and commutative, which provides the flexibility needed to make parallel implementations more efficient.

Algorithm 3 Safe softmax with online normalizer calculation

1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: **for** $j \leftarrow 1, V$ **do**
4: $\quad m_j \leftarrow \max(m_{j-1}, x_j)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j-m_j}$
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i-m_V}}{d_V}$
9: **end for**

$$\begin{bmatrix} m_V \\ d_V \end{bmatrix} = \begin{bmatrix} x_1 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} x_2 \\ 1 \end{bmatrix} \oplus \ldots \oplus \begin{bmatrix} x_V \\ 1 \end{bmatrix}$$

where $x_i, m_V, d_V \in \mathbb{R}$. The binary operation $\oplus : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$ is defined as:

$$\begin{bmatrix} m_i \\ d_i \end{bmatrix} \oplus \begin{bmatrix} m_j \\ d_j \end{bmatrix} = \begin{bmatrix} \max(m_i, m_j) \\ d_i \times e^{m_i - \max(m_i, m_j)} + d_j \times e^{m_j - \max(m_i, m_j)} \end{bmatrix}$$

Stony Brook University

In 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 $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

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 $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

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

Algorithm 3 Safe softmax with online normalizer calculation

$m_0 \leftarrow -\infty$

$d_0 \leftarrow 0$

for $j \leftarrow 1, V$ **do**

  $m_j \leftarrow \max(m_{j-1}, x_j)$

  $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$

end for

for $i \leftarrow 1, V$ **do**

  $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$

end for

$$\begin{bmatrix} m_V \\ d_V \end{bmatrix} = \begin{bmatrix} x_1 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} x_2 \\ 1 \end{bmatrix} \oplus \dots \oplus \begin{bmatrix} x_V \\ 1 \end{bmatrix}$$

where $x_i, m_V, d_V \in \mathbb{R}$. The binary operation $\oplus : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ is defined as:

$$\begin{bmatrix} m_i \\ d_i \end{bmatrix} \oplus \begin{bmatrix} m_j \\ d_j \end{bmatrix} = \begin{bmatrix} \max(m_i, m_j) \\ d_i \times e^{m_i - \max(m_i, m_j)} + d_j \times e^{m_j - \max(m_i, m_j)} \end{bmatrix}$$

FAR BEYOND

Stony Brook University

Tiling Visualization – Outer Loop Iteration 1 (Review)

- Inner loop: repeat T_r times

FAR BEYOND

Stony Brook University

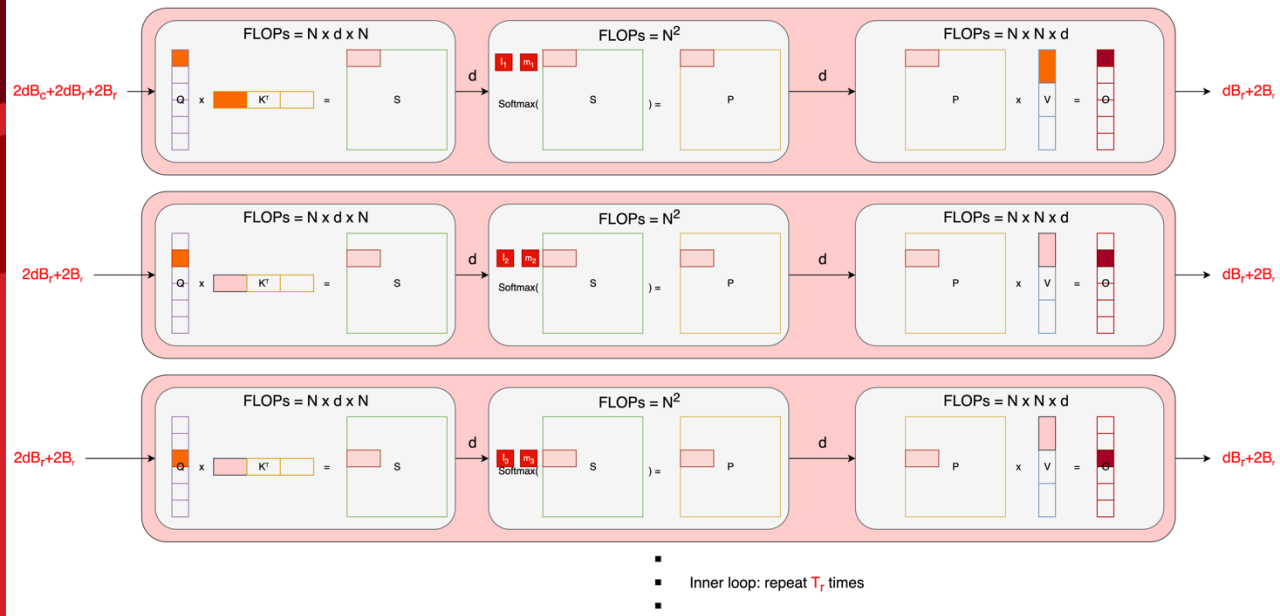
Method Details

- Tiling
- Online Softmax
- Recomputation
- Block-Sparse Extension

FAR BEYOND

Stony Brook University

Recomputation for Backward Pass



Algorithm 4 FLASHATTENTION Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ in HBM, vectors $\ell, m \in \mathbb{R}^N$ in HBM, on-chip SRAM of size M, softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability p_{drop}, pseudo-random number generator state $\mathcal{R}$ from the forward pass.

1: Set the pseudo-random number generator state to $\mathcal{R}$.
2: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ 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 $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_i, \dots, \mathbf{dO}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: Initialize $\mathbf{dQ} = (0)_{N \times d}$ in HBM and divide it into T_r blocks $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Initialize $\mathbf{dK} = (0)_{N \times d}, \mathbf{dV} = (0)_{N \times d}$ in HBM and divide $\mathbf{dK}, \mathbf{dV}$ in to T_c blocks $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$ and $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$, of size $B_c \times d$ each.
6: **for** $1 \le j \le T_c$ **do**
7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8: Initialize $\tilde{\mathbf{dK}}_j = (0)_{B_c \times d}, \tilde{\mathbf{dV}}_j = (0)_{B_c \times d}$ on SRAM.
9: **for** $1 \le i \le T_r$ **do**
10: Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
11: On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
12: On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
13: On chip, compute $\mathbf{P}_{ij} = \text{diag}(l_i)^{-1} \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{B_r \times B_c}$.
14: On chip, compute dropout mask $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r \times B_c}$ where each entry has value $\frac{1}{1-p_{\text{drop}}}$ with probability $1 - p_{\text{drop}}$ and value 0 with probability p_{drop}.
15: On chip, compute $\mathbf{P}_{ij}^{\text{dropped}} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
16: On chip, compute $\tilde{\mathbf{dV}}_j \leftarrow \tilde{\mathbf{dV}}_j + (\mathbf{P}_{ij}^{\text{dropped}})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
17: On chip, compute $\mathbf{dP}_{ij}^{\text{dropped}} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$.
18: On chip, compute $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{dropped}} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
19: On chip, compute $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{B_r}$.
20: On chip, compute $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
21: Write $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$ to HBM.
22: On chip, compute $\tilde{\mathbf{dK}}_j \leftarrow \tilde{\mathbf{dK}}_j + \tau \mathbf{dS}_{ij}^\top \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
23: **end for**
24: Write $\mathbf{dK}_j \leftarrow \tilde{\mathbf{dK}}_j, \mathbf{dV}_j \leftarrow \tilde{\mathbf{dV}}_j$ to HBM.
25: **end for**
26: Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

FAR BEYOND

Stony Brook University

Method Details

- Tiling
- Online Softmax
- Recomputation
- Block-Sparse Extension

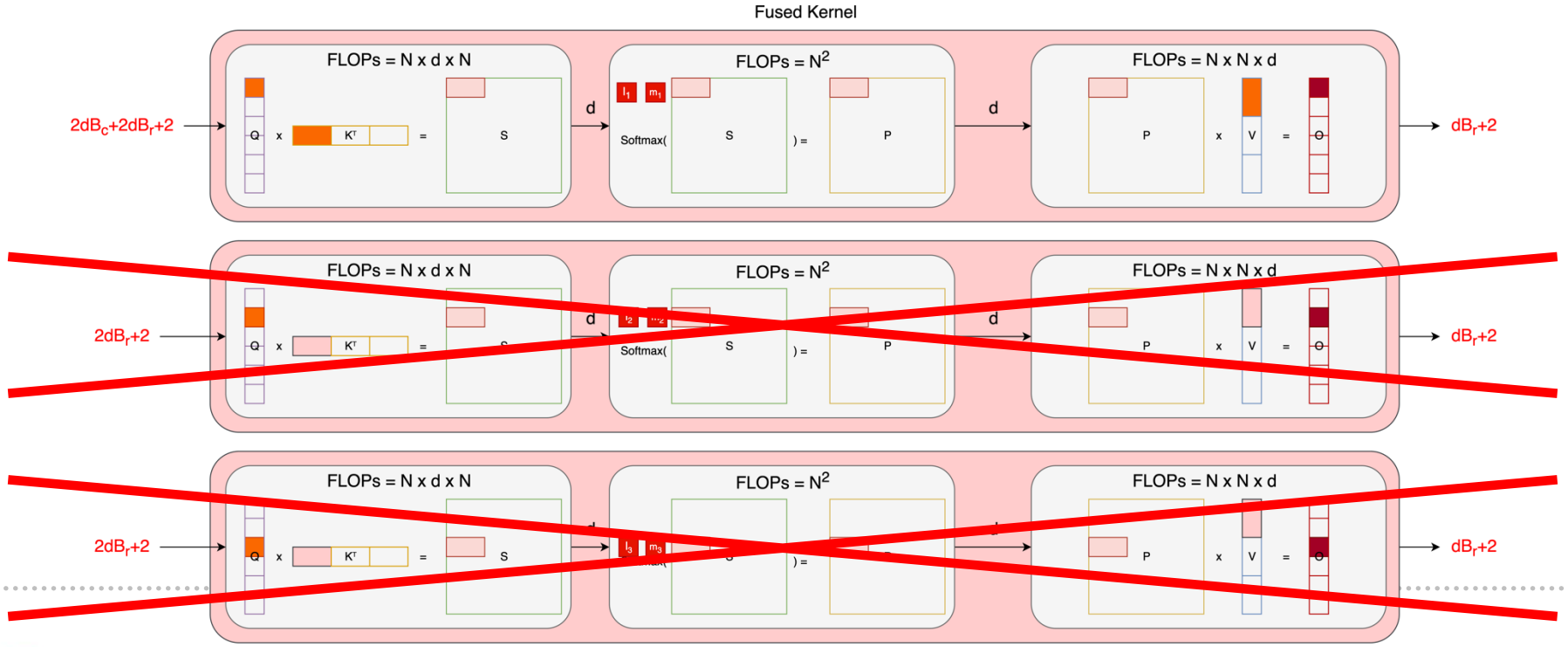
FAR BEYOND

Stony Brook University

Block-Sparse Extension

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S} \odot \mathbb{1}_{\tilde{\mathbf{M}}}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$

$$\tilde{\mathbf{M}} \in \{0,1\}^{N\times N} \qquad \text{where } (\mathbf{S} \odot \mathbb{1}_{\tilde{\mathbf{M}}})_{kl} = \mathbf{S}_{kl} \text{ if } \tilde{\mathbf{M}}_{kl} = 1 \text{ and } -\infty \text{ if } \mathbf{M}_{kl} = 0.$$



- Inner loop: repeat T_r times

FAR BEYOND

Stony Brook University

Block-Sparse Extension IO Complexity

- IO w/o block sparse extension: $(12N^2d^2 / M) + (16N^2d / M) + (3Nd)$

$$O(N^2d^2M^{-1})$$

- Let s be the fraction of nonzero blocks in the block-sparsity mask:
  IO w/ block sparse extension: $s(12N^2d^2 / M) + s(16N^2d / M) + (3Nd)$

$$\Theta(Nd + N^2d^2M^{-1}s)$$

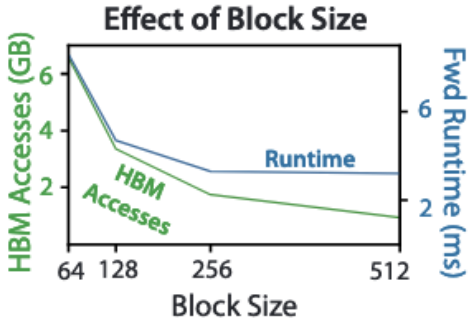
Stony Brook University

Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

Attention Kernel Benchmarks

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



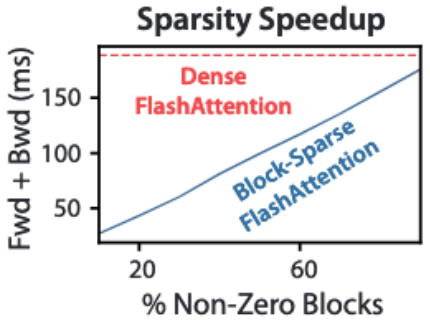


Figure 2: **Left**: Forward + backward runtime of standard attention and FLASHATTENTION for GPT-2 medium (seq. length 1024, head dim. 64, 16 heads, batch size 64) on A100 GPU. HBM access is the primary factor affecting runtime. **Middle**: Forward runtime of FLASHATTENTION (seq. length 1024, head dim. 64, 16 heads, batch size 64) on A100 GPU. Fewer HBM accesses result in faster runtime, up to a point. **Right**: The runtime (for seq. length 4K) of block-sparse FLASHATTENTION is faster than FLASHATTENTION by a factor proportional to the sparsity.

End-to-End Training Speed

Table 1: Training time of BERT-large, starting from the same initialization provided by the MLPerf benchmark, to reach the target accuracy of 72.0% on masked language modeling. Averaged over 10 runs on 8×A100 GPUs.

BERT Implementation	Training time (minutes)
Nvidia MLPerf 1.1 [58]	20.0 ± 1.5
FlashAttention (ours)	**17.4** ± 1.4

Table 2: GPT-2 small and medium using FlashAttention achieve up to 3× speed up compared to Huggingface implementation and up to 1.7× compared to Megatron-LM. Training time reported on 8×A100s GPUs.

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×)**

Stony Brook University

End-to-End Training Speed Baselines

- All the baselines (MLPerf FMHA, HuggingFace, Megatron) try to fuse kernels
- Not IO-aware
  - Still **materialize the full N×N attention matrix in HBM** and perform $O(N^2)$ HBM reads/writes.
- FlashAttention's contribution is to **avoid storing this matrix at all** via
  - Tiling
  - Online softmax
  - Recomputation

Better Models with Longer Sequences

Table 4: GPT-2 small with FLASHATTENTION, with 4× larger context length compared to Megatron-LM, is still 30% faster while achieving 0.7 better perplexity. Training time on 8×A100 GPUs is reported.

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×)

Stony Brook University

Long Document Classification and LRA

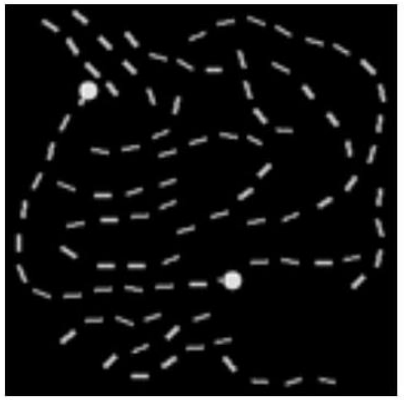
Table 5 shows that sequence length 16K outperforms length 512 by 4.3 points on MIMIC, and that length 8K outperforms length 512 by 8.5 points on ECtHR. The discrepancies may be due to subtle distribution shifts: MIMIC-III contains specialized medical text and thus may be more susceptible to a distribution shift in the document length, whereas ECtHR contains general language.

Table 5: Long Document performance (micro F_1) at different sequence lengths using FlashAttention.

	512	1024	2048	4096	8192	16384
MIMIC-III [47]	52.8	50.7	51.7	54.6	56.4	**57.1**
ECtHR [6]	72.2	74.3	77.1	78.6	**80.7**	79.2

Table 6: We report the first Transformer model that can achieve non-random performance on Path-X and Path-256.

Model	Path-X	Path-256
Transformer	✗	✗
Linformer [84]	✗	✗
Linear Attention [50]	✗	✗
Performer [12]	✗	✗
Local Attention [80]	✗	✗
Reformer [51]	✗	✗
SMYRF [19]	✗	✗
FlashAttention	**61.4**	✗
Block-sparse FlashAttention	56.0	**63.1**



Path-X example

FAR BEYOND

Stony Brook University

Who is Strong on LRA Now

- State Space Models (S4, S5, S5 variants, Mamba, etc.)
  - **S4** (Structured State Spaces) was the first architecture that really "solved" LRA in practice Achieve 96% accuracy on Path-X.
  - **S5 and related variants** improve further. Achieve ~98.6% accuracy on Path-X.
- Other non-SSM architectures
  - **MEGA** (Moving Average Equipped Gated Attention)
  - Convolutional / Hyena-style models
  - RWKV and other recurrent hybrids
- Newer work often reports LRA as a *small part* of a broader evaluation suite (LongBench, long-document QA, code, speech, synthetic algorithmic tasks, etc.)

Memory Footprint and Runtime

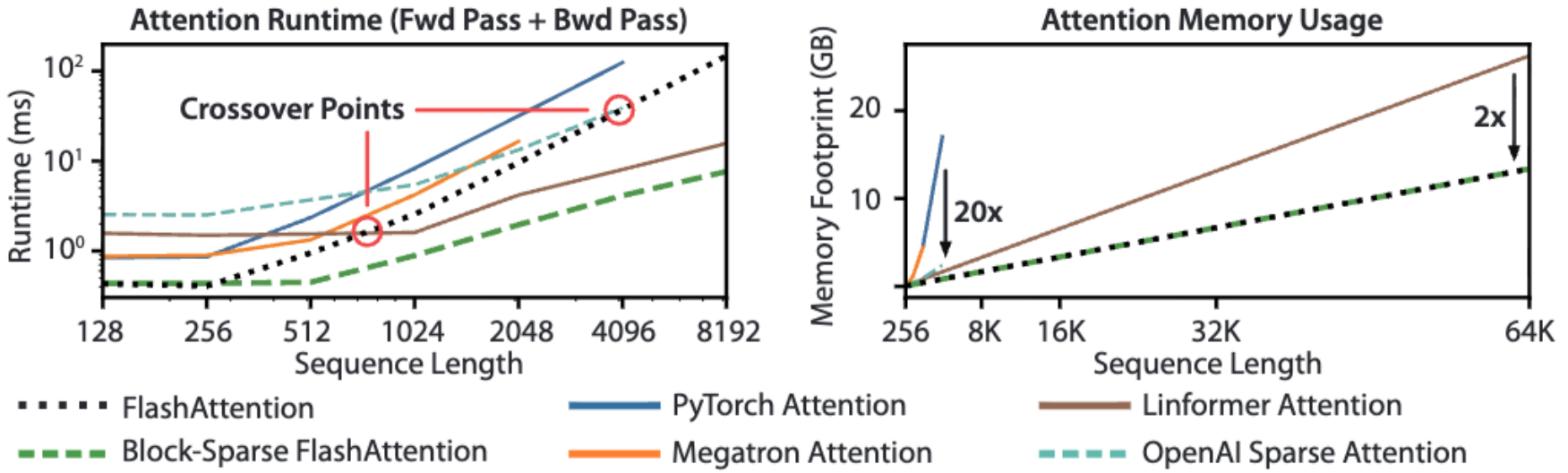


Figure 3: **Left:** runtime of forward pass + backward pass. **Right:** attention memory usage.

Stony Brook University

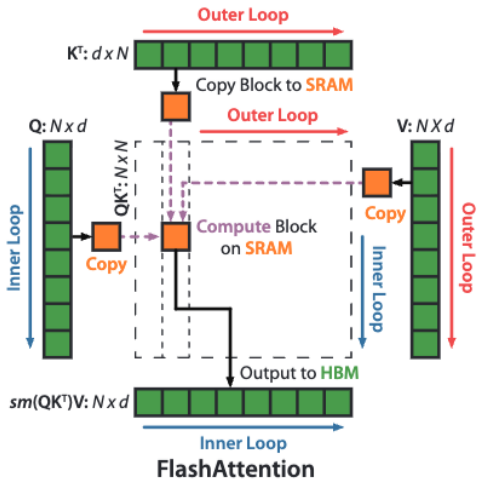
Outlines

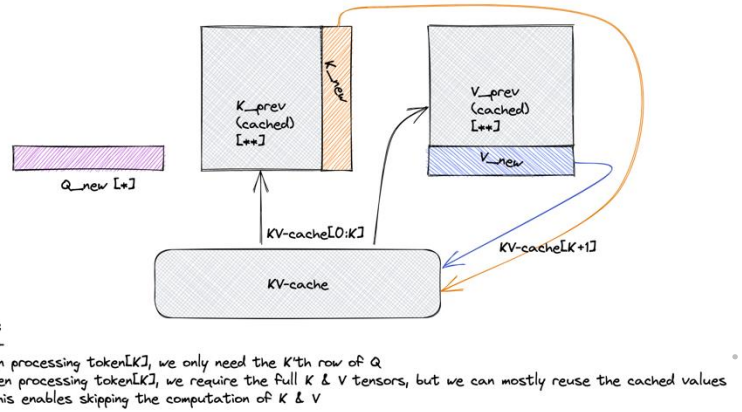
- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

Stony Brook University

One-Sentence Rule

- Rule: **FlashAttention accelerates full-sequence attention (many queries over the whole sequence at once), not token-by-token KV-cached decoding.**
- Think in terms of attention structure:
  - Full N×N attention inside a kernel ⇒ good match for FlashAttention.
  - Per-token 1×t attention with KV cache ⇒ different regime and different bottlenecks.





https://medium.com/@shoa.devs/kv-cache-secrets-boost-llm-inference-efficiency-ae6c53968857

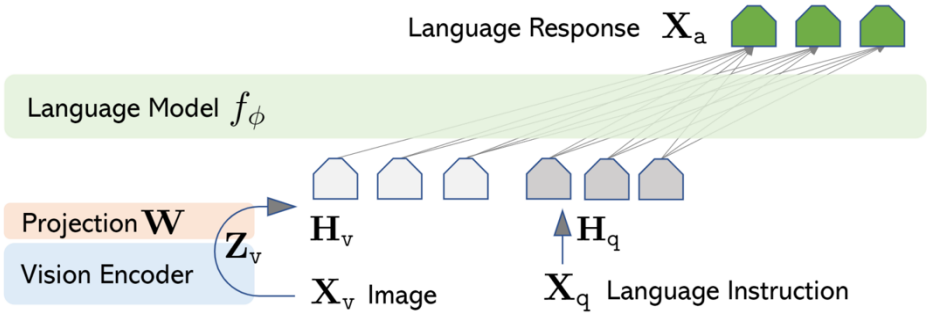
FAR BEYOND

Helpful Scenarios - Training

- NTP (next-token prediction) training
  - Standard LM pretraining/finetuning on full sequences.
  - Example: GPT-2/3–style LM trained on 2K–4K token sequences (books, code, web text).
- Batch RL training on trajectories (PPO / RLHF)
  - After rollout, the training step is structurally the same as NTP training.
  - Example: RLHF fine-tuning a chat model on logged conversations.

Stony Brook University

Helpful Scenarios - Inference



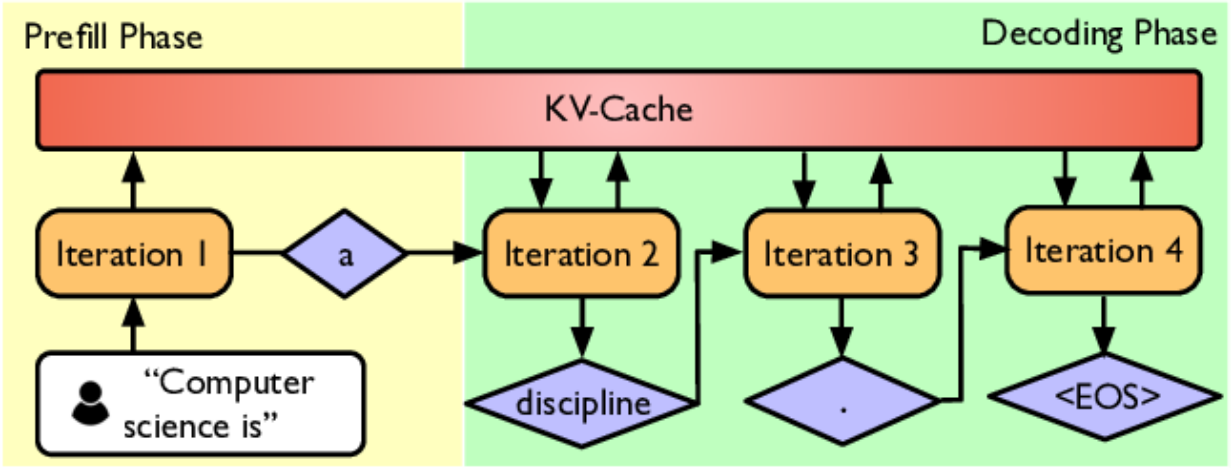
Visual Instruction Tuning (LLaVA)

- Encoder-style inference/scoring
  - Model sees the entire input at once; no KV cache.
  - Example: BERT-style document classification.
- KV cache prefill (long prefix)
  - Encode a long prefix once, then decode autoregressively.
  - Examples:
    - Chat LLM taking a long conversation history as context.
    - **LLaVA-style VLM**: first encode hundreds/thousands of **vision tokens** (image patches) + text prompt as a prefix; FlashAttention accelerates this prefill over the vision+text sequence.

Stony Brook University

Limited-Benefit Scenarios

- Autoregressive decode after prefill
  - Pattern: many small “1 × t” attentions with a KV cache.
  - Cost dominated by repeatedly reading the growing KV cache from HBM.
- Online RL interaction (step-by-step acting)
  - Same structure as autoregressive decoding: generate actions/tokens one step at a time, keep a KV cache across steps.



Introduction to Multimodal Learning — Part 8: Incremental Prefilling as an Inference optimization

FAR BEYOND

Stony Brook University

Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

Stony Brook University

Strengths

- Simple, powerful core ideas: tiling, online softmax, recomputation
- Big real-world speedups at the time
- Theoretical grounding that mathematically proves the IO lower bound
- Foundational impact
  - The FA-1 design (tiling + online softmax + recomputation) became the standard pattern for modern attention kernels.
  - Even though frameworks like PyTorch 2.2+ now default to FlashAttention-2, FA-1 is the original idea that made fused, IO-aware attention the default baseline.

Weaknesses (FlashAttention2 Trys to Solve)

- Limited parallelism and low occupancy
  - FA-1 achieves only about **25–40%** of the theoretical peak FLOPs/s for attention on A100.
- 1 thread-block processes **one head.** Parallelism is only over **batch × heads**.
  - If batch_size × num_heads is smaller than the number of SMs (common for long-context LMs with small batch), many SMs are idle
- Too many non-matmul FLOPs (slow on GPU)
  - Tensor Cores for FP16/BF16 matmuls is ~16× faster than scalar FP32 units
  - FA-1's online softmax update does **extra scalar work**
- Too much shared-memory traffic inside a block

FlashAttention Algorithm (Review)

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 ℓ 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

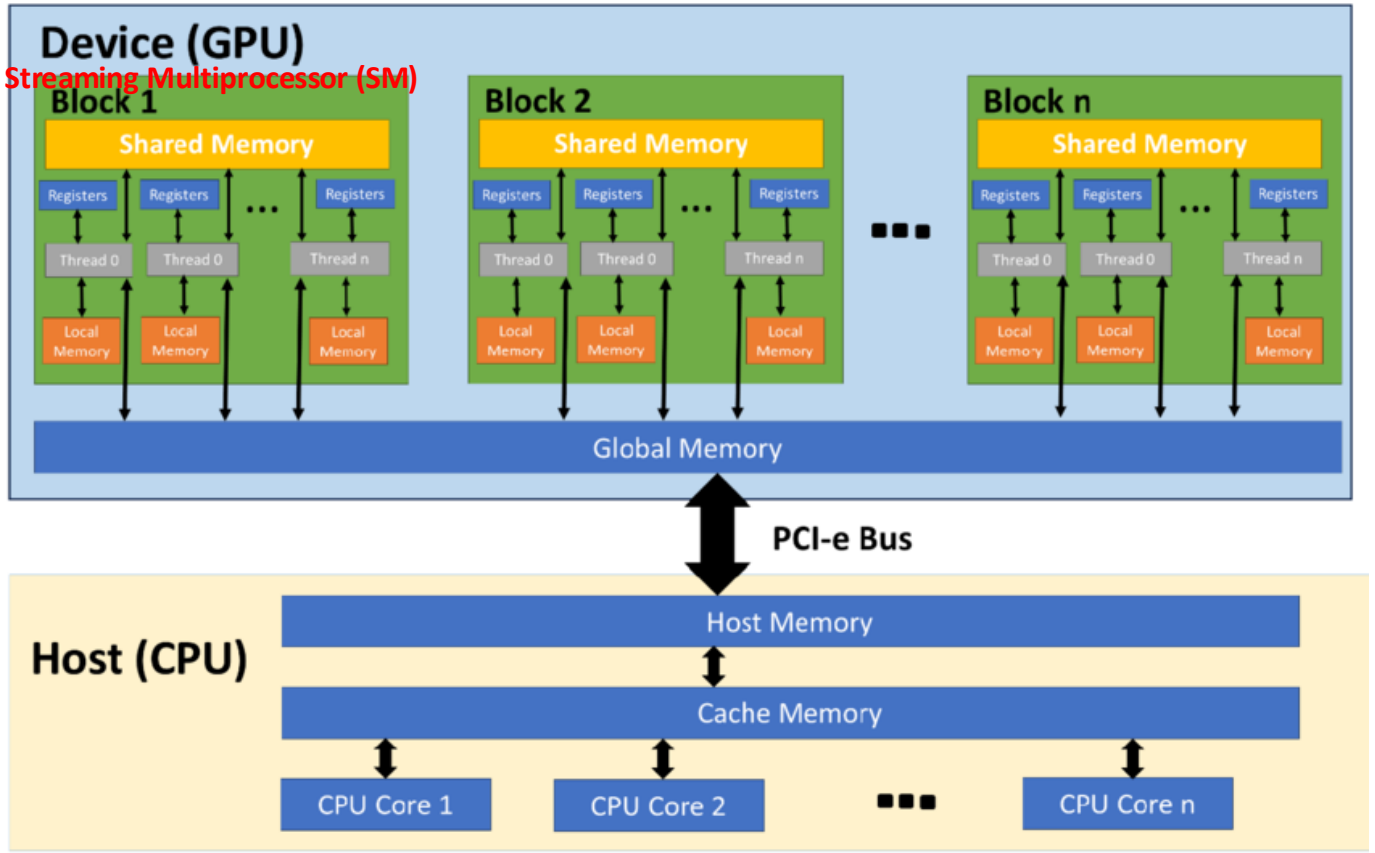
end for

Return $\mathbf{O}$.

FAR BEYOND

GPU Execution Model and Kernels (Review)

- A **kernel** = one operation launched on the GPU
- Each kernel:
  1. Loads its inputs from HBM to registers/SRAM
  2. Computes
  3. Writes outputs back to HBM
- Runtime of a kernel:
  - Td: Bytes moved between SRAM and HBM (IO)
  - Tc: computation time
  - Ideally: T = max(Tc, Td)



Optimizing MRI Data Processing by exploiting GPU Acceleration for Efficient Image Analysis and Reconstruction

Stony Brook University

Outlines

- Prerequisites: Hardware and Kernels
- FlashAttention Core Ideas
- Method Details
- Experiment Results
- Applicability: When FlashAttention Helps
- Strengths and Weaknesses
- Possible Improvement

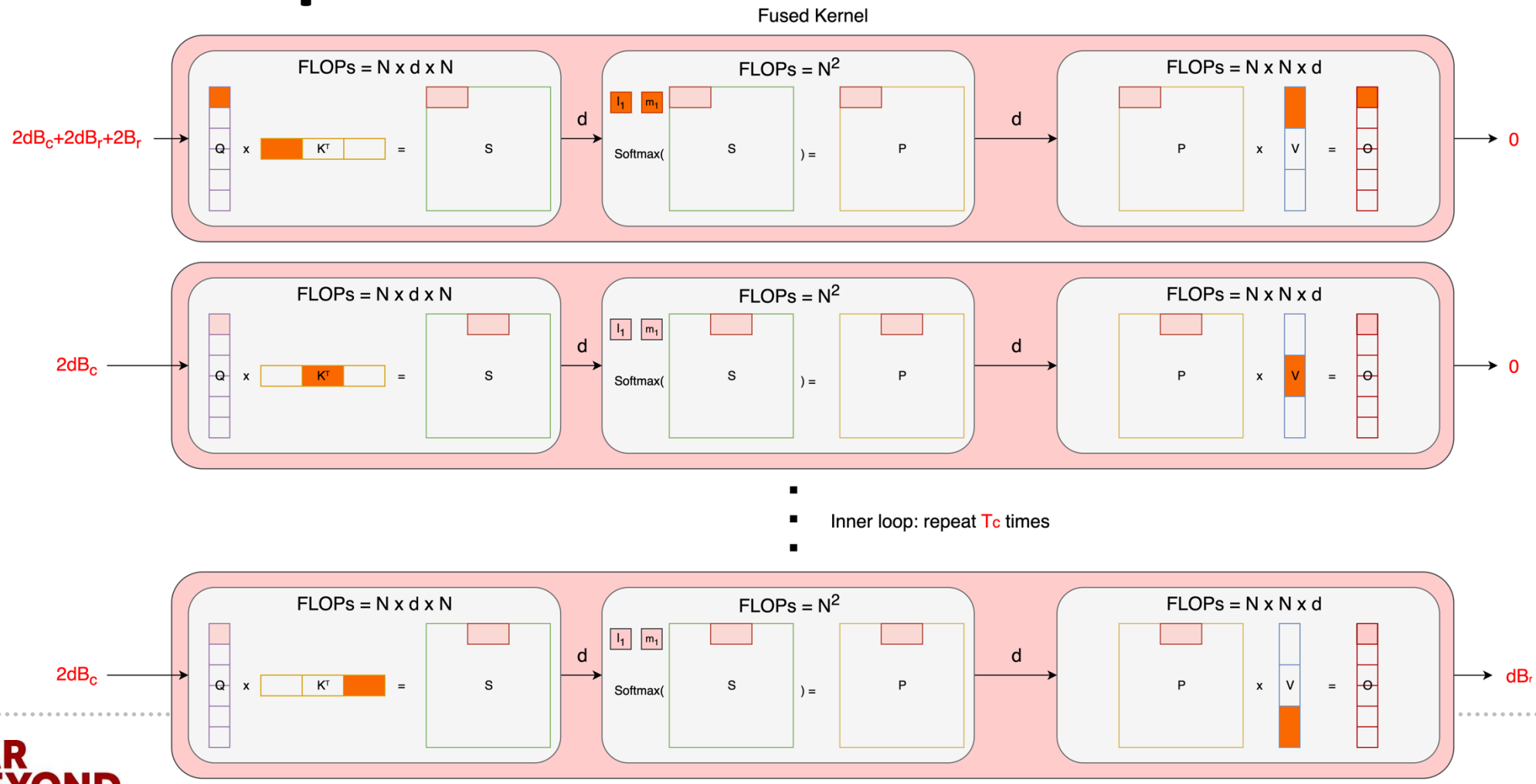
Stony Brook University

Possible Improvement

- Further reduce some constant terms in the IO complexity
- Idea
  - Make B_r equal to $B_c = \lceil \frac{M}{4d} \rceil$ instead of $\min(\lceil \frac{M}{4d} \rceil, d)$ (optional)
  - Swaps the tile traversal order and accumulates each O tile on-chip
- Reduce HBM IO vs the original forward algorithm without changing the math.
- Whether it wins in practice would need benchmarking
- Only works for FA-1

Stony Brook University

Outer Loop Iteration 1



FAR BEYOND

Stony Brook University

Complexity Analysis

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

$T_c = \lceil \frac{N}{B_c} \rceil \qquad T_r = \lceil \frac{N}{B_r} \rceil$

$T_c = 4dN / M$

- Each inner loop iteration: $2dB_c$
- T_c iterations for inner loop: $T_c \times (2dB_c) + 3dB_r + 2B_r$
  - $2dN + 3dB_r + 2B_r$
- T_r outer loop iterations: $T_r \times (2dN + 3dB_r + 2B_r + dB_c)$
  - $8N^2d^2 / M + 3dN + 2N$
  - Original: $12N^2d^2 / M + 16N^2d / M + 3Nd$
- HBM access complexity: $O(N^2d^2M^{-1})$

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.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: 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.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: 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), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: 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.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

FAR BEYOND

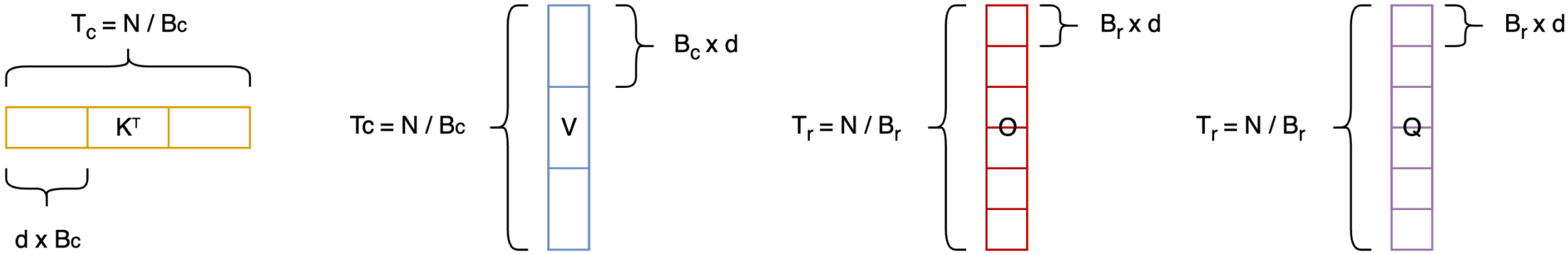
Original Algorithms (Review)

Tiling – Define Block Size

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

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



Stony Brook University

Tiling Visualization – Outer Loop Iteration 1

- Inner loop: repeat T_r times

FAR BEYOND

Stony Brook University

Tiling Visualization – Outer Loop Iteration 2

- Inner loop: repeat T_r times

FAR BEYOND

Stony Brook University

FlashAttention IO Complexity Analysis

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

$T_c = \lceil \frac{N}{B_c} \rceil \qquad T_r = \lceil \frac{N}{B_r} \rceil$

$T_c = 4dN / M$

- Each inner loop iteration: $3dB_r + 4B_r$
- T_r iterations for inner loop: $T_r \times (3dB_r + 4B_r) + 3dB_c$
  - $3dN + 4N + 3dB_c$
- T_c outer loop iterations: $T_c \times (3dN + 4N + 3dB_c)$
  - $12N^2d^2 / M + 16N^2d / M + 3Nd$
- HBM access complexity: $O(N^2d^2M^{-1})$

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.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: 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.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: 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), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: 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.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

FAR BEYOND

Q&A