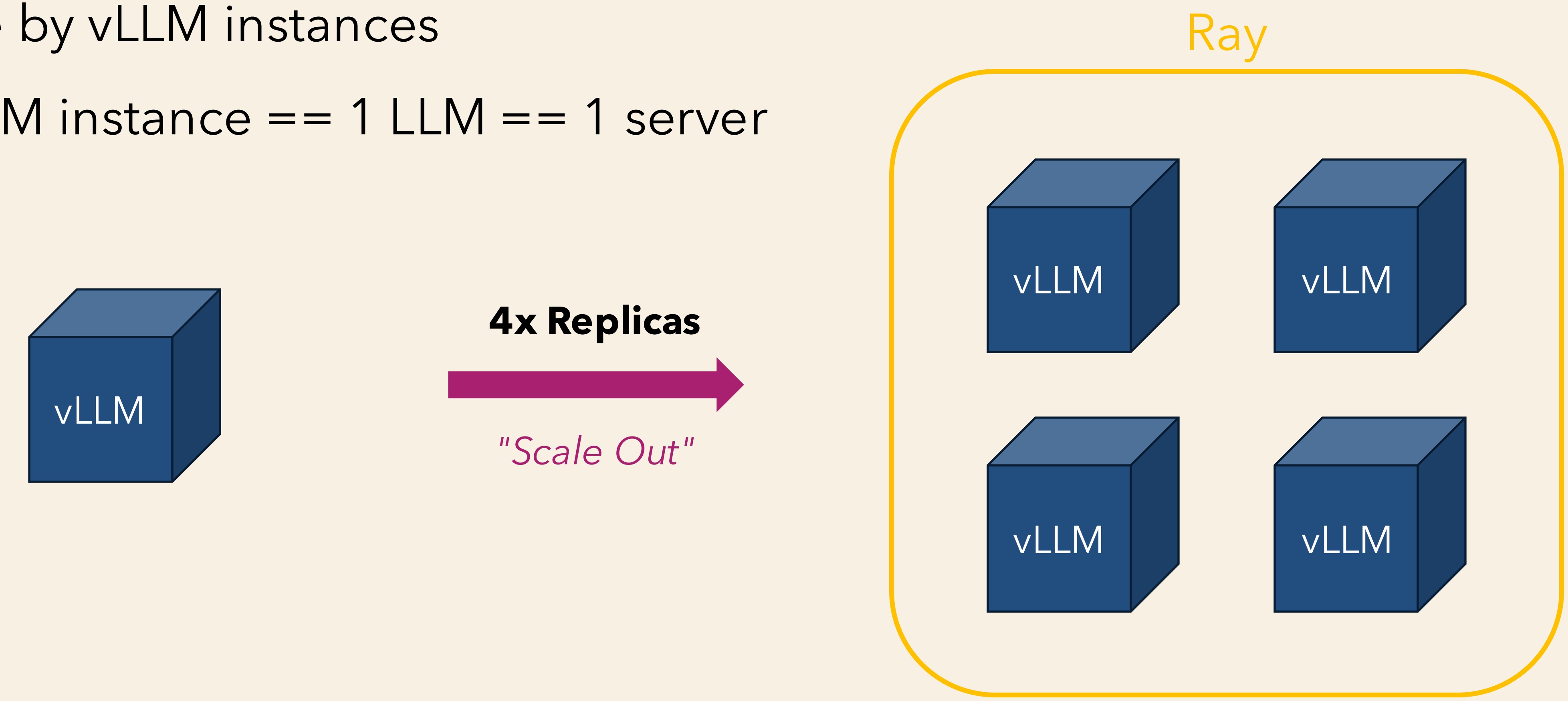
LLM MODEL SERVING

Team 6
Fengyu Gao, Shunqiang Feng, Wei Shen, Zihan Zhao

Splitwise (ISCA '24)
&
DistServe (OSDI '24)

How to scale LLM services?

- Scale by vLLM instances
- 1 vLLM instance == 1 LLM == 1 server



However, it is not ideal...

- Ideal scenario
  - Just-right GPU compute utilization
  - Just-right GPU memory utilization
  - Just-right interconnect utilization (e.g. NVLink, Infiniband)
  - ……

The reality is...

- You have scaled to Nx replicas to satisfy the demand, but each with
  - $\ll$100% GPU compute utilization on average
  - $\ll$100% GPU memory utilization on average
  - $\ll$100% interconnect utilization on average
  - ......

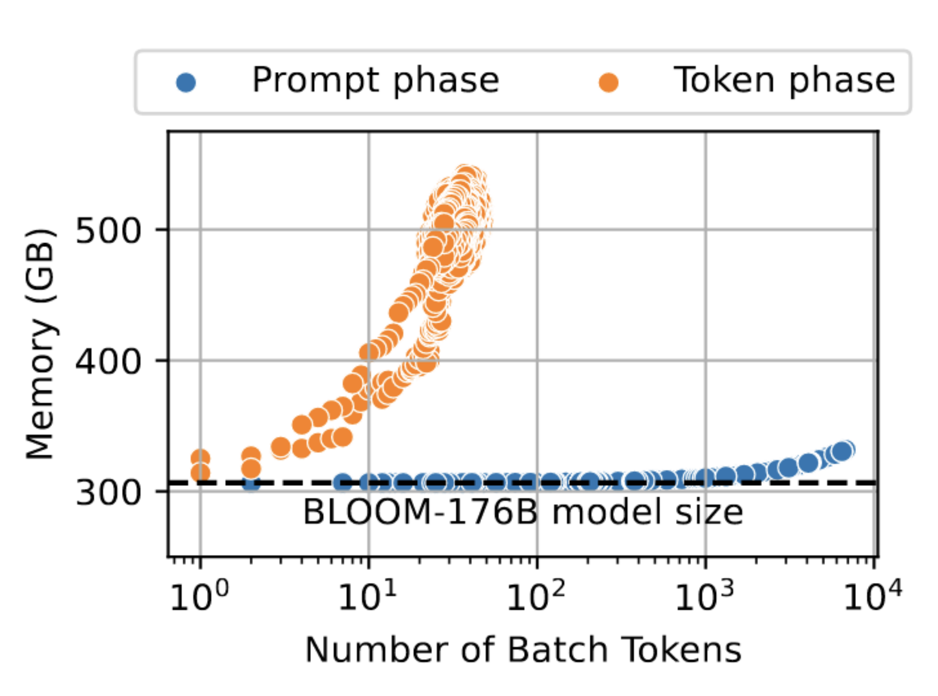
Use up **ALL OF** the available resources before scaling

One Request, Two Phases

- Prefill
  - The process of generating the first token
  - A compute-intensive phase
- Decode
  - The process of generating subsequent tokens
  - A memory-intensive phase

Imbalanced Memory Usage

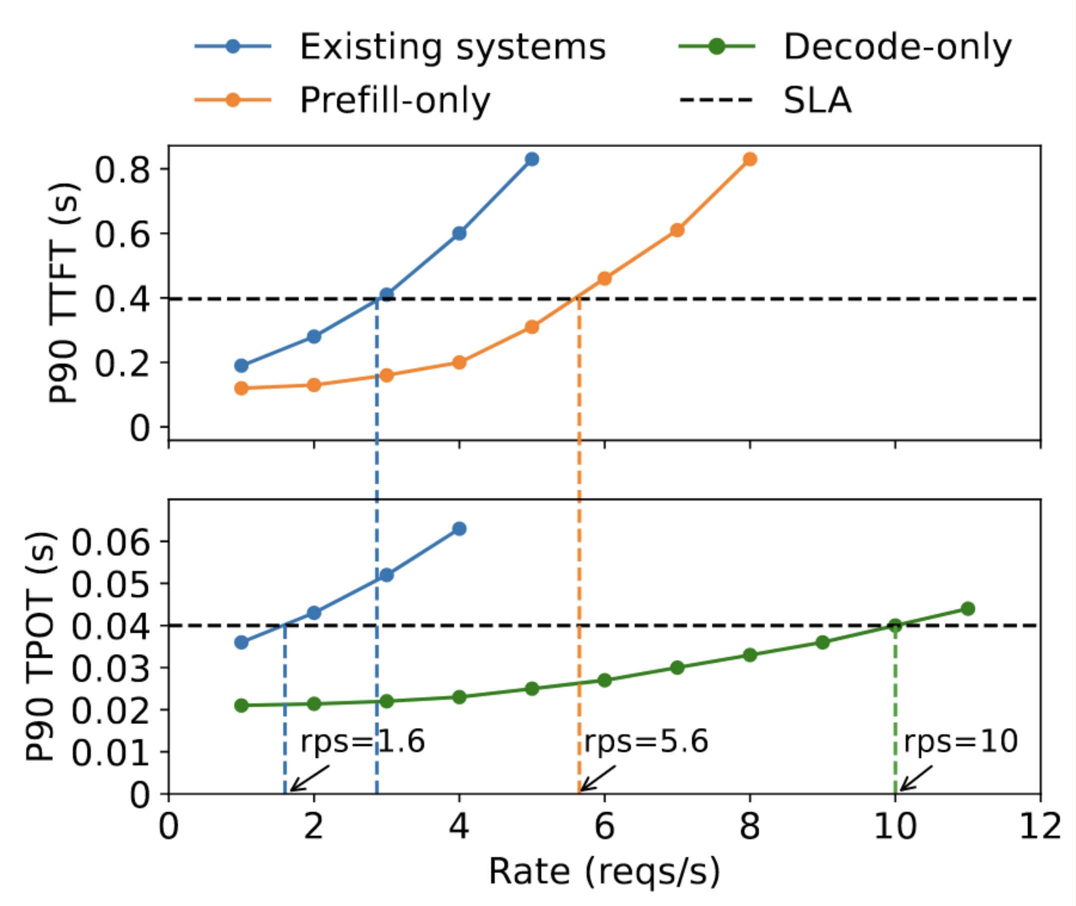
- Prefill (prompt phase)
  - Few additional memory used
- Decode (token phase)
  - Fast growing memory usage



Resource Contention

Metric	Importance to user
End-to-end (E2E) latency	Total query time that the user sees
Time to first token (TTFT)	How quickly user sees initial response
Time between tokens (TBT)	Average token streaming latency
Throughput	Requests per second

- Prefill, if served independently, finishes sooner
- Decode, if served independently, finishes sooner
- Prefill + decode is slow



Solution

- DistServe reduces contention by separating phases on different hardware
- Splitwise chooses the "right" hardware to serve the separated phases

	A100	H100	Ratio
TFLOPs	19.5	66.9	3.43×
HBM capacity	80GB	80GB	1.00×
HBM bandwidth	2039GBps	3352GBps	1.64×
Power	400W	700W	1.75×
NVLink	50Gbps	100Gbps	2.00×
Infiniband	200GBps	400GBps	2.00×
Cost per machine [5]	$17.6/hr	$38/hr	2.16×

DistServe System

- Prefill and decode scale independently
- KV cache transfers through NVLink

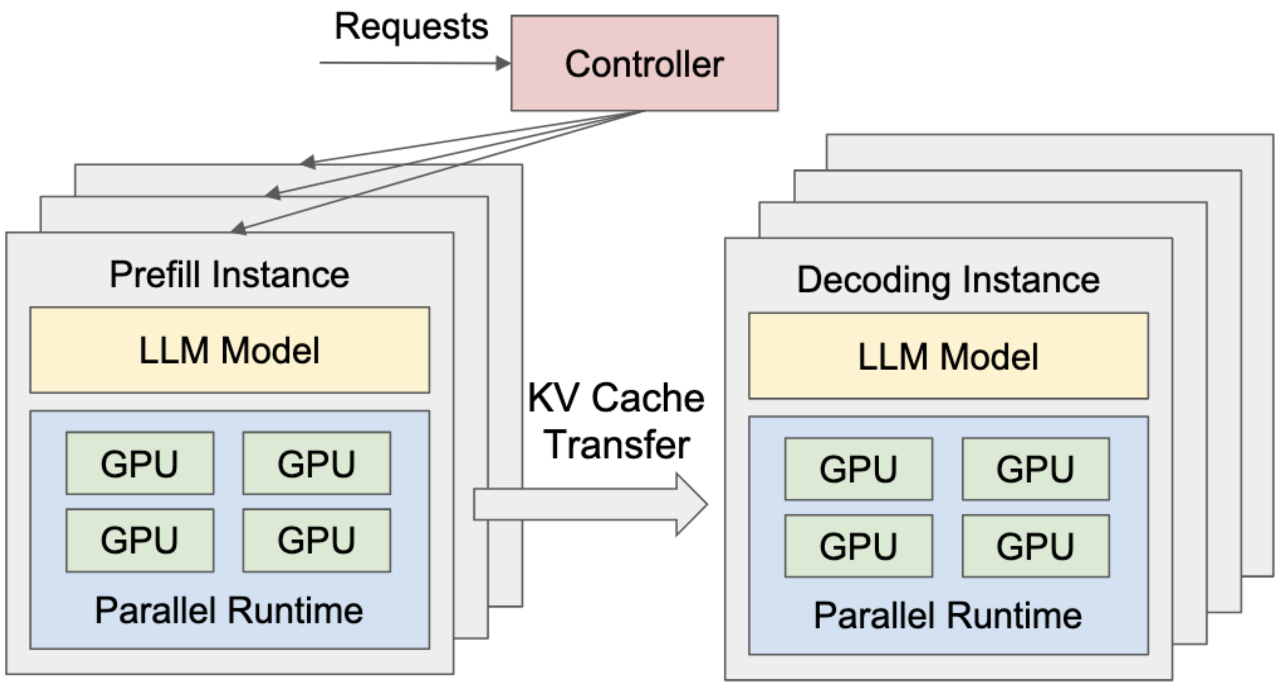
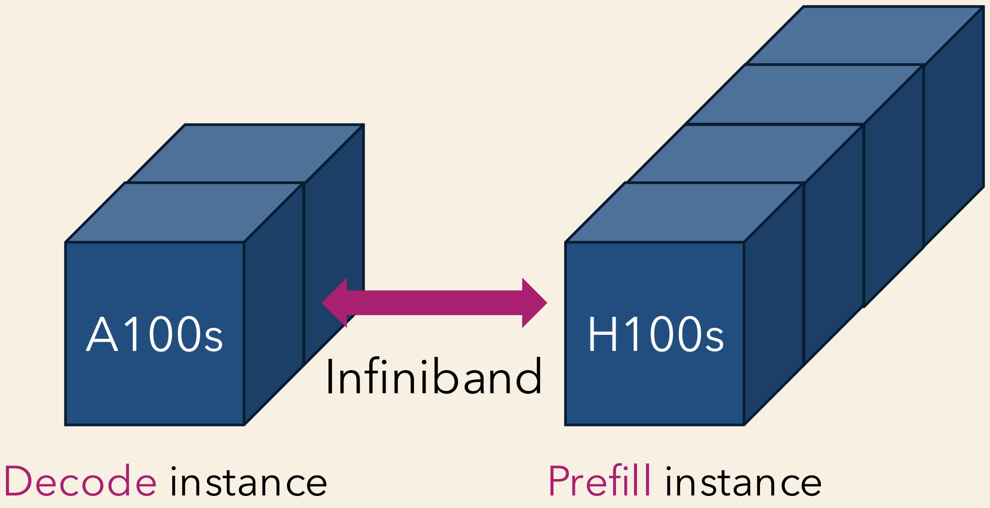


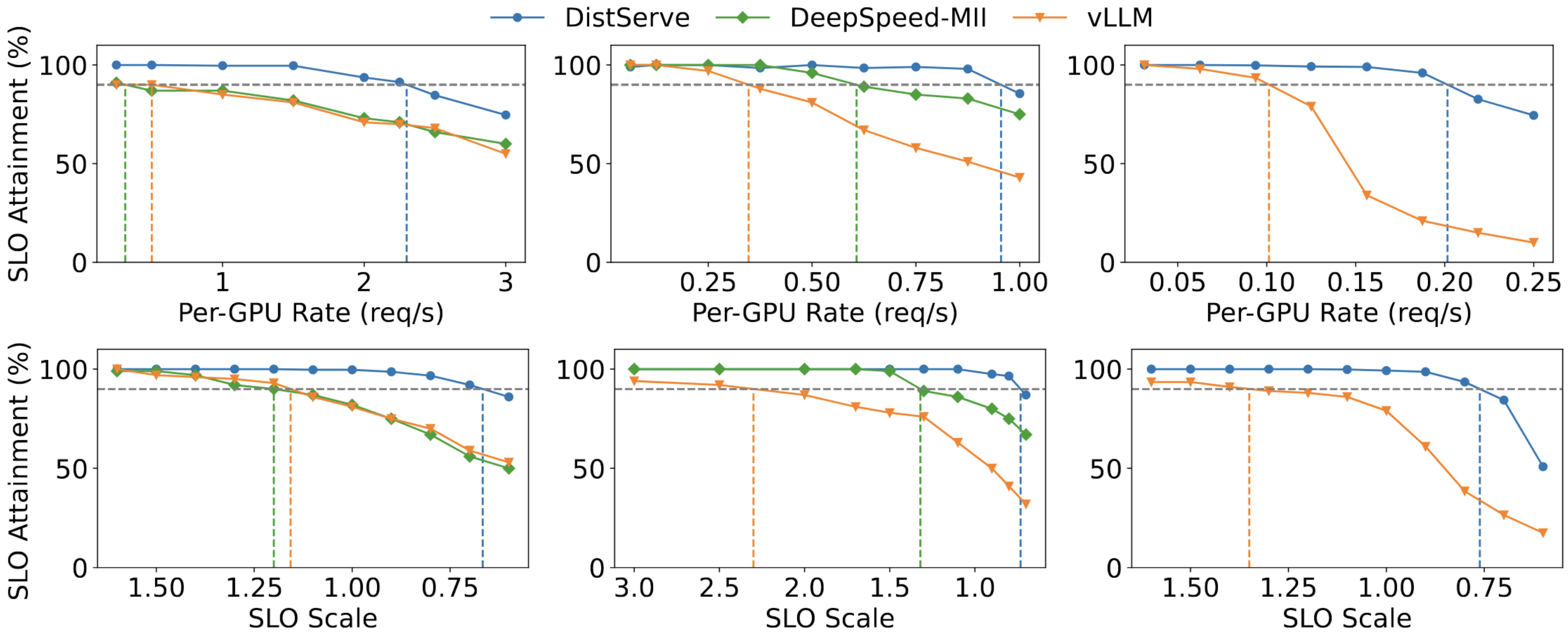
Figure 6: DistServe Runtime System Architecture

Splitwise System

- Prefill and decode are served and scheduled on different hardware based on needs
    - Computation
    - Memory
    - Power
    - Cost
- KV cache transfers through Infiniband



Evaluation – DistServe



(a) OPT-13B (b) OPT-66B (C) OPT-175B

Evaluation – Splitwise

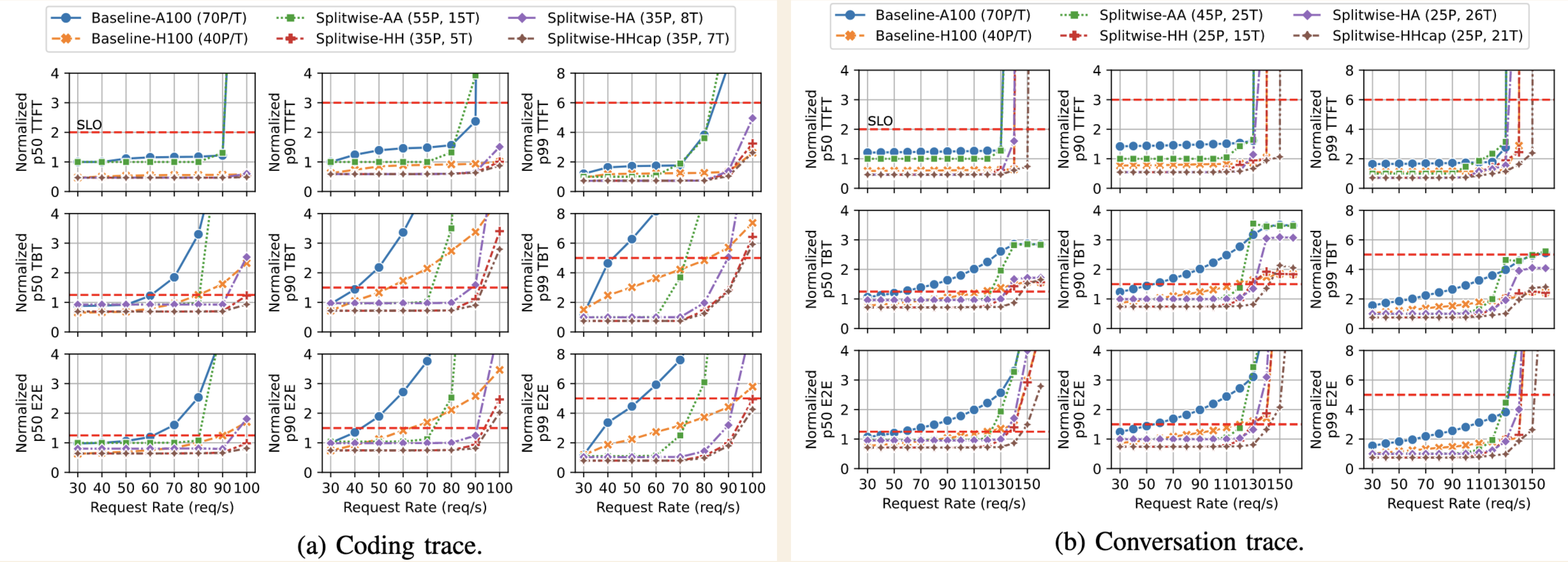


Fig. 16: Latency metrics across input loads for iso-power throughput optimized clusters. Dashed red lines indicate SLO.

Takeaways

- DistServe
  - Separates prefill and decode phases on different GPUs
  - Allows different phases to scale independently
- Splitwise
  - Separates prefill and decode phases on different machines
  - Maximizes hardware utilization based on needs and cluster settings

More efficient LLM serving by disaggregation

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

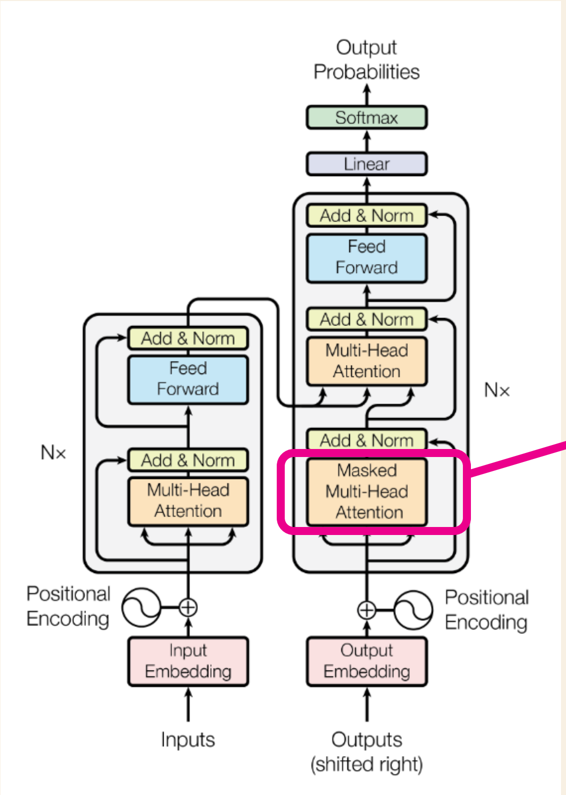
17k+ ★

FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning

Shunqiang Feng (mpp7ez)

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

1. Introduction
Standard Attention Mechanics



1. Input: Queries (Q), Keys (K), Values (V)
2. Compute $S = QK^t$, then $P = \text{softmax}(S)$, then $O = PV$
3. Requires storing **N×N matrix** in memory
4. Complexity: **$O(N^2)$** in time and memory

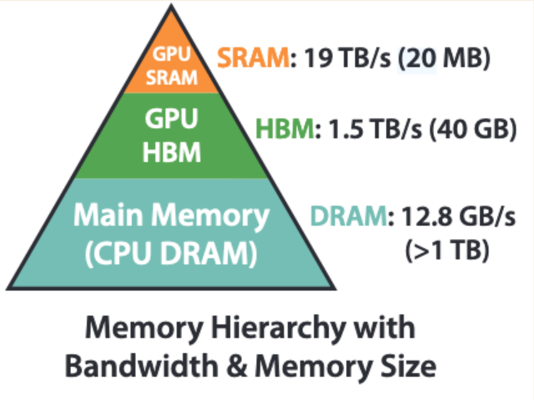
This limits the ability of Transformers to model long contexts!

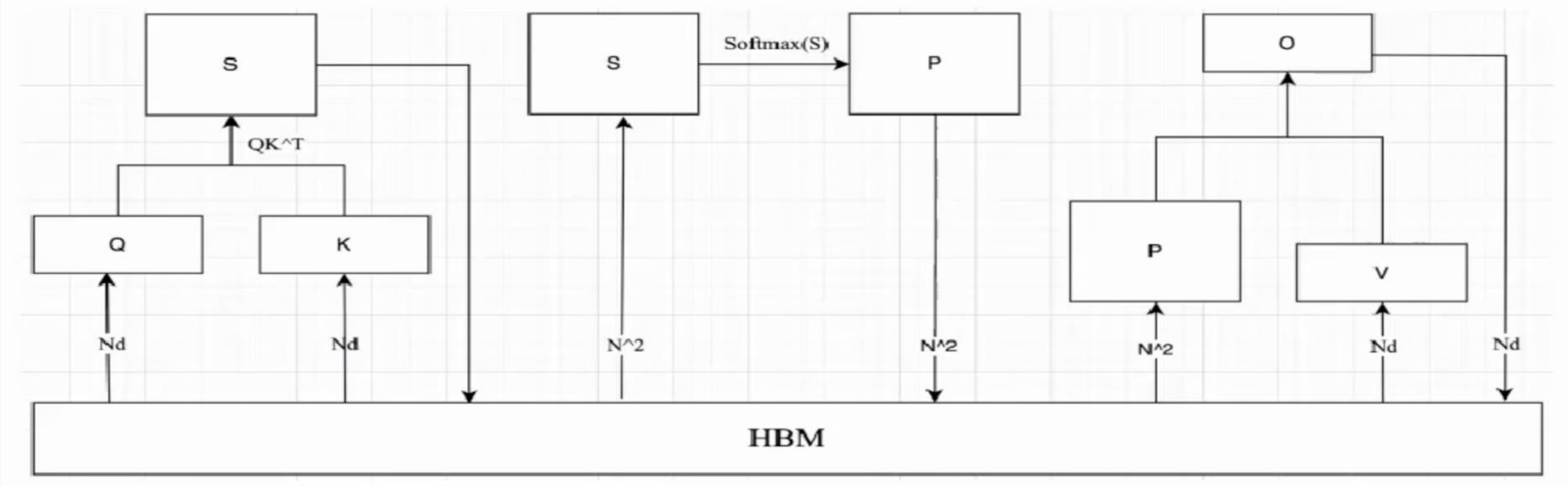
1. Introduction
Previous Work

Approximate Attention

- Sparse & low-rank methods reduce **FLOPs**, not necessarily runtime
- Root cause: **ignores memory (I/O) bottlenecks**

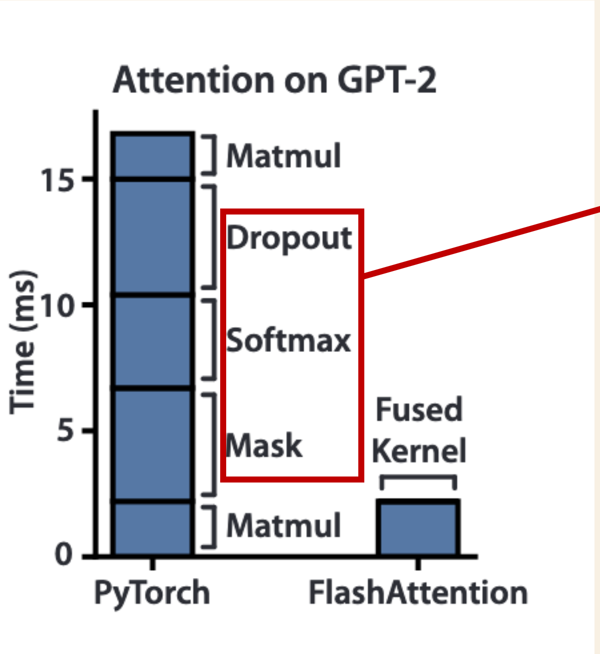
1. Introduction
GPU Memory Bottlenecks





- GPU HBM is **large but slow**, SRAM is **fast but small**
- Standard attention reads/writes large matrices multiple times
- IO dominates runtime, especially for long sequences
- Empirical: Standard attention often becomes **memory-bound**

1. Introduction
GPU Memory Bottlenecks



Memory-Bound

2. Algorithm Overview

- Modern GPUs: **Memory access is the bottleneck**
- Make attention **IO-aware**

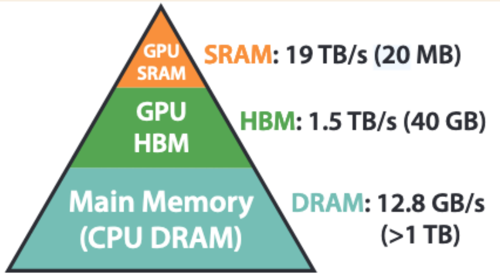
⇨

- **Exact** attention with fewer memory accesses
- Achieves up to **9× speedup** over standard attention

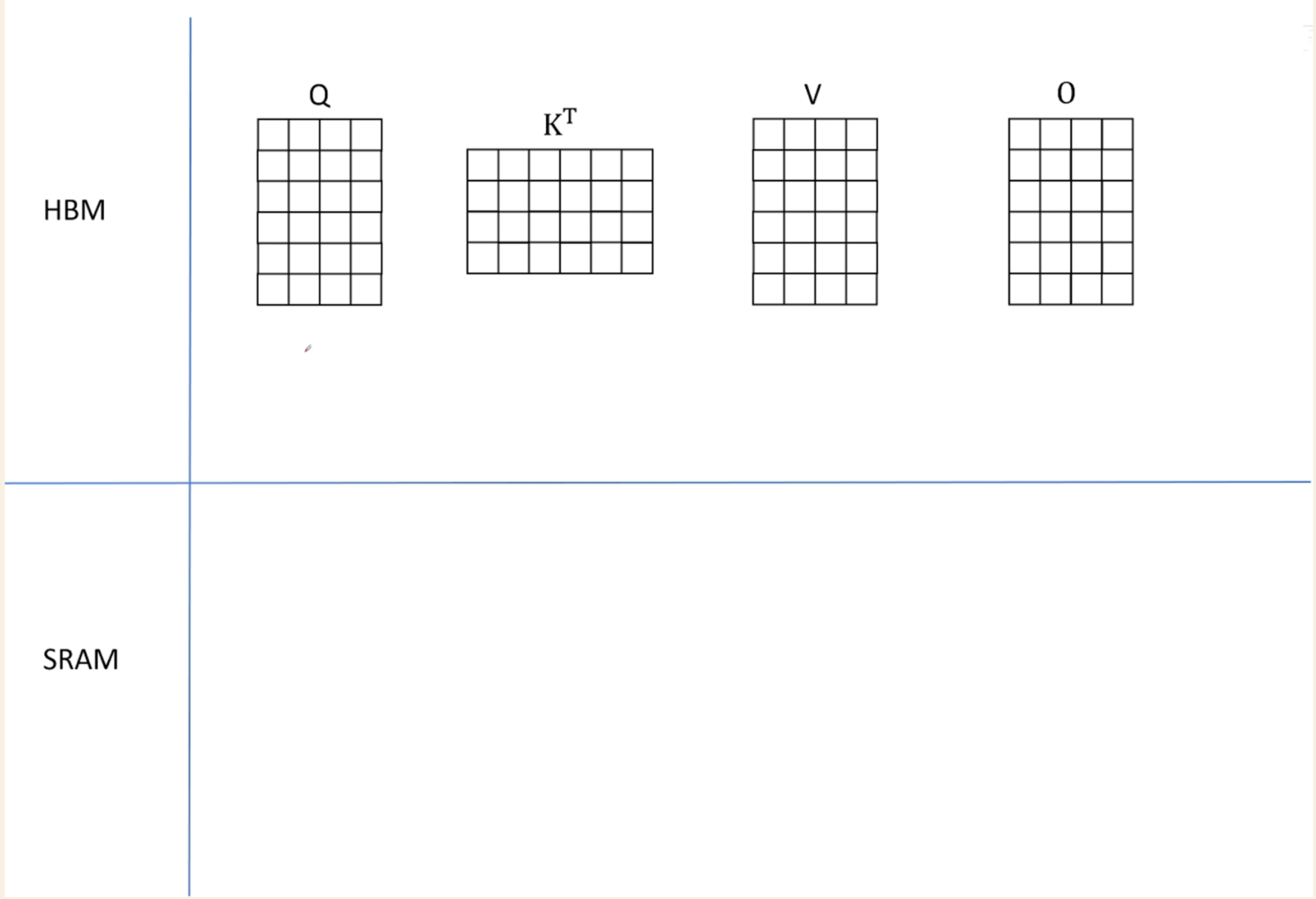
↓

- Avoid materializing large attention matrix
- Use **tiling**: load small blocks of Q, K, V into fast SRAM
- **Fuse** all attention steps into a **single kernel**
- Output written once to HBM – huge IO savings!

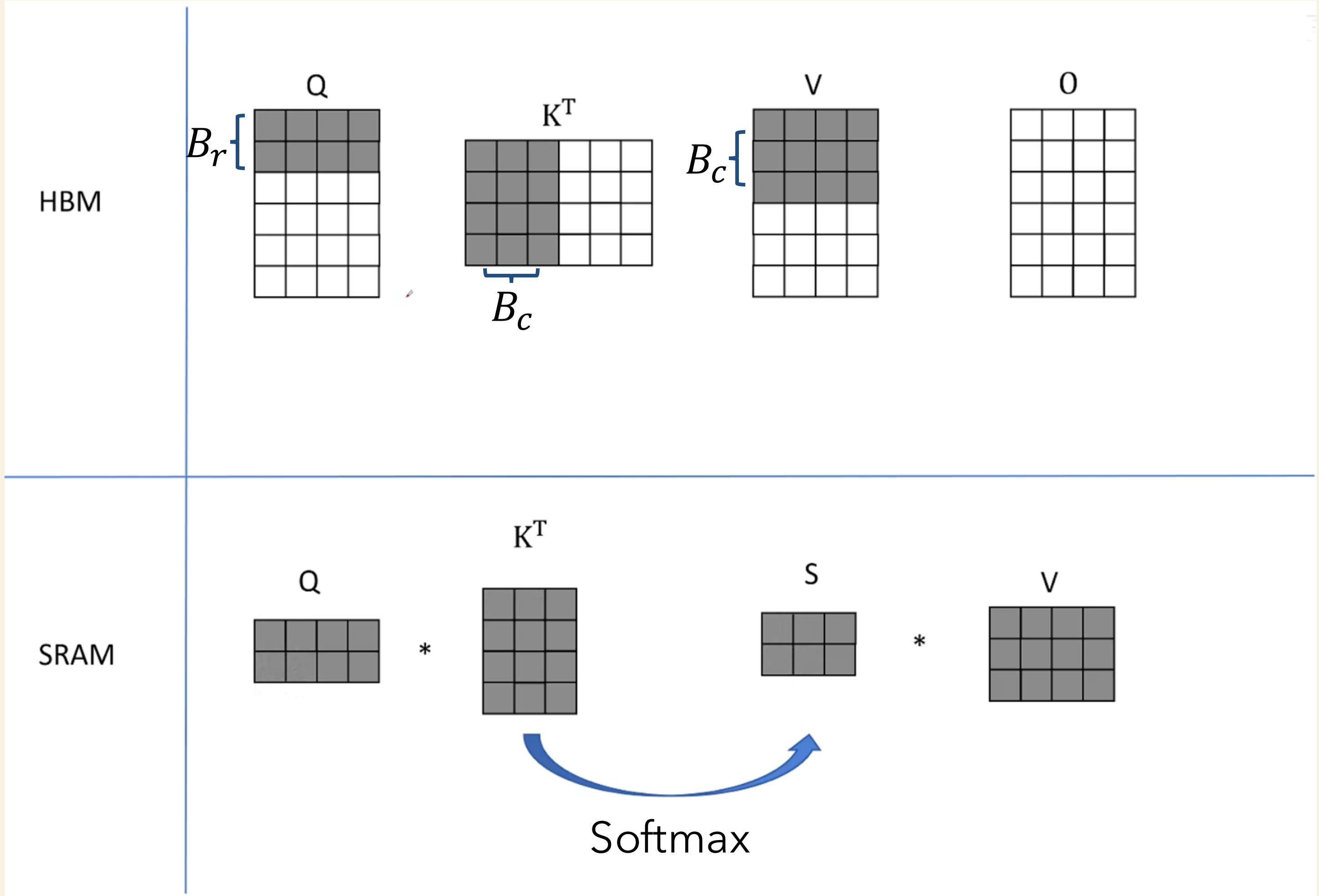
2. Algorithm
Inner Loop



Memory Hierarchy with Bandwidth & Memory Size

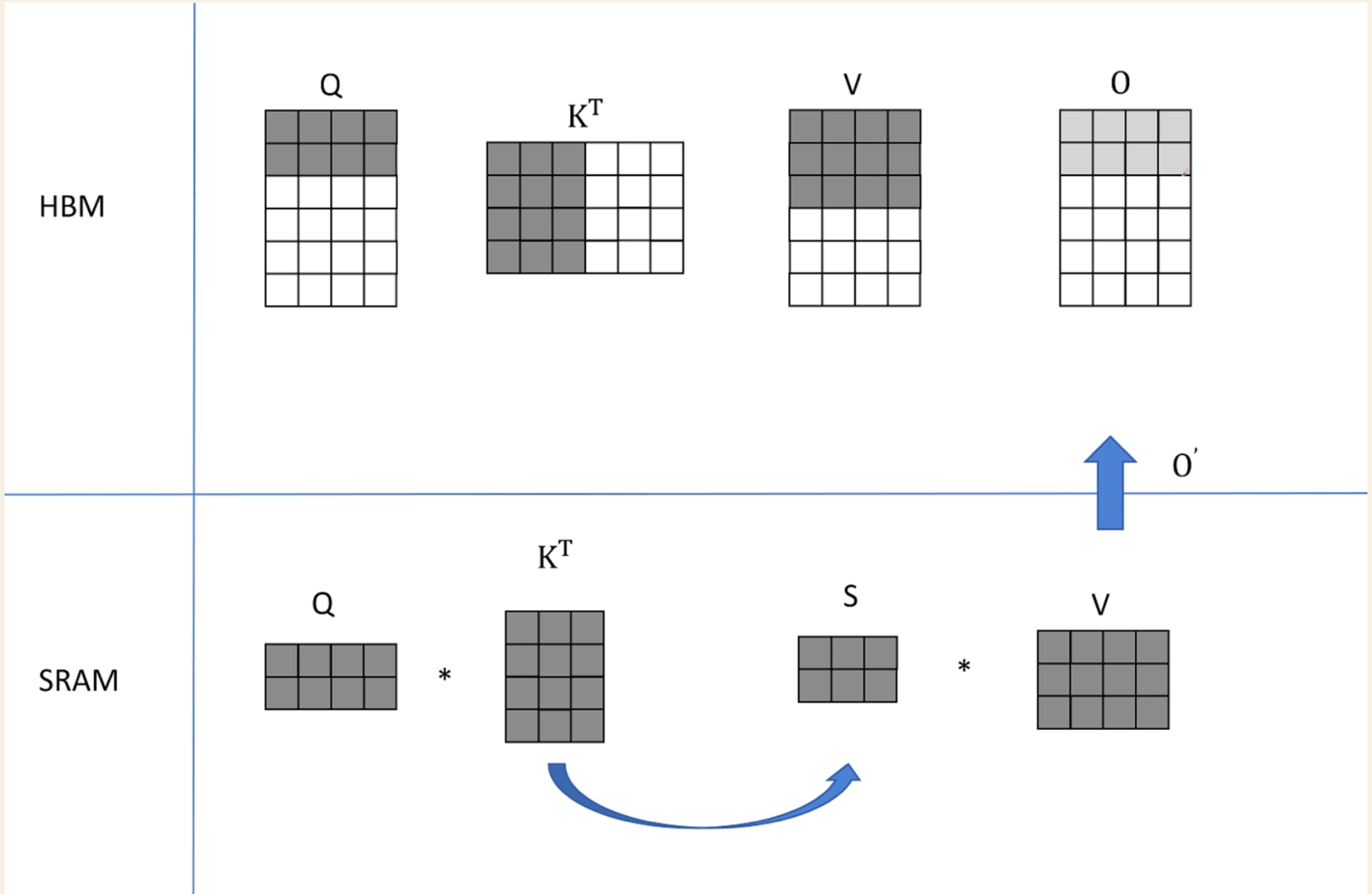


2. Algorithm
Inner Loop

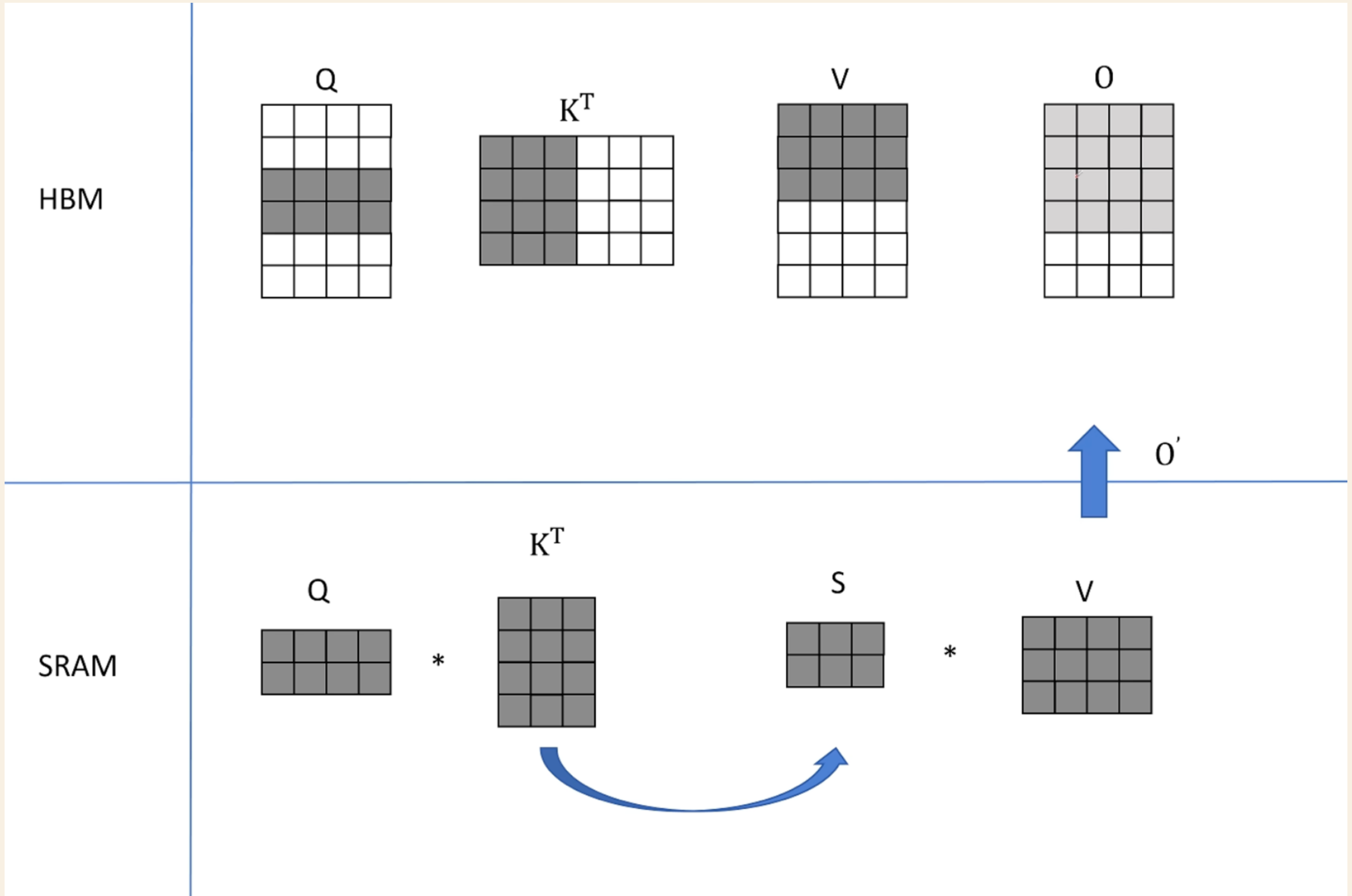


Here, we assume
$B_r = 2, B_c = 3$

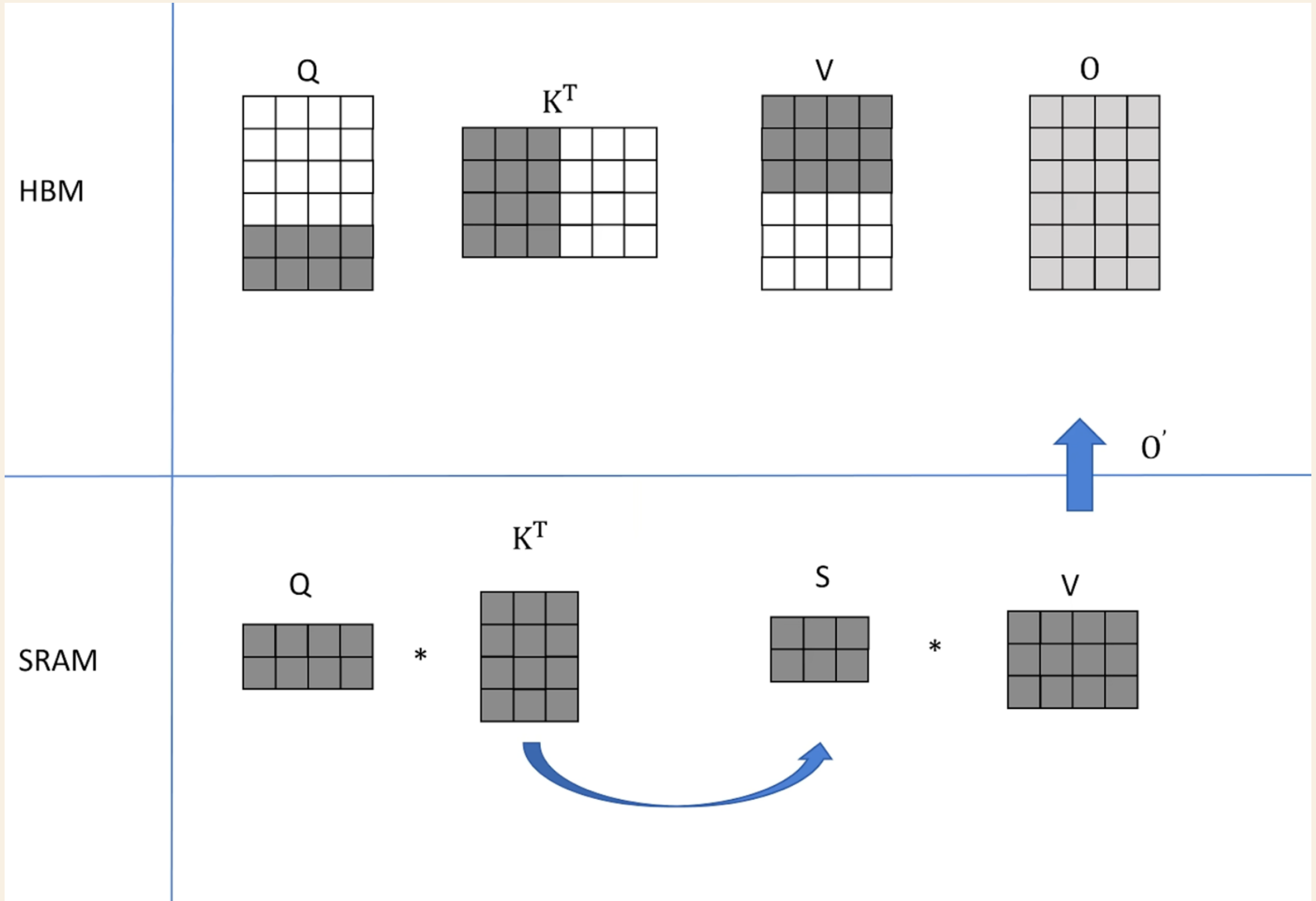
2. Algorithm
Inner Loop



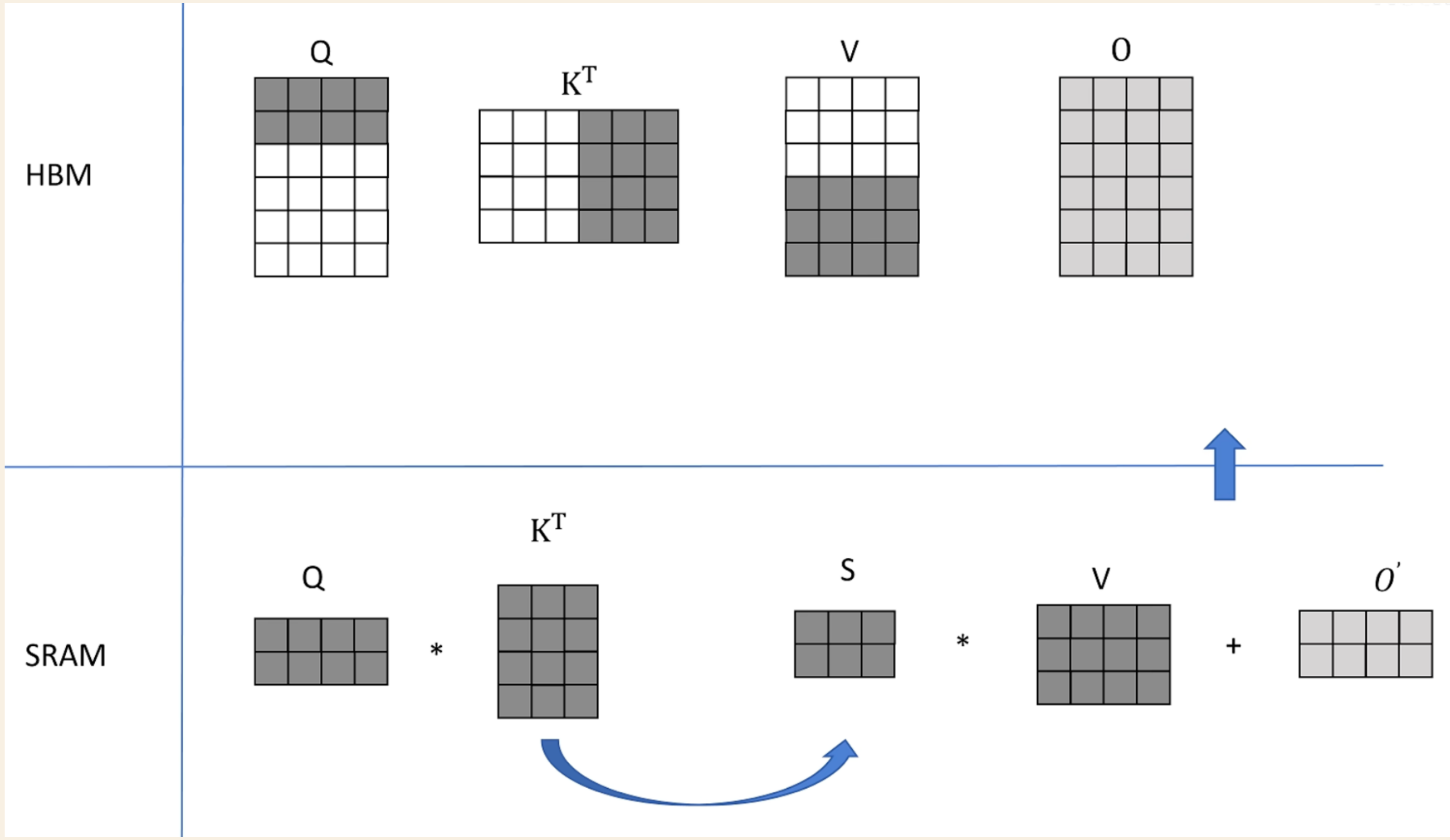
2. Algorithm
Inner Loop



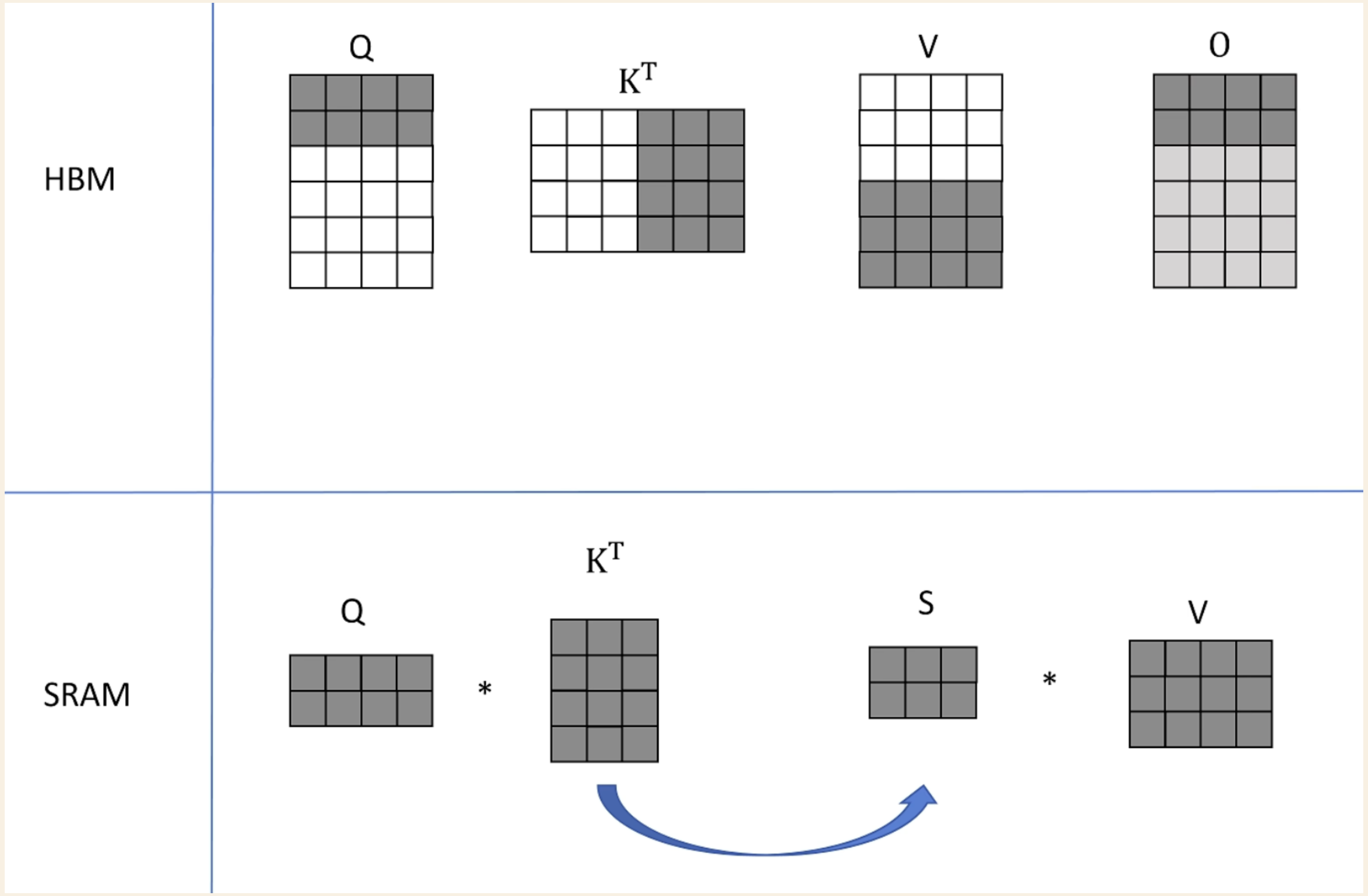
2. Algorithm
Inner Loop



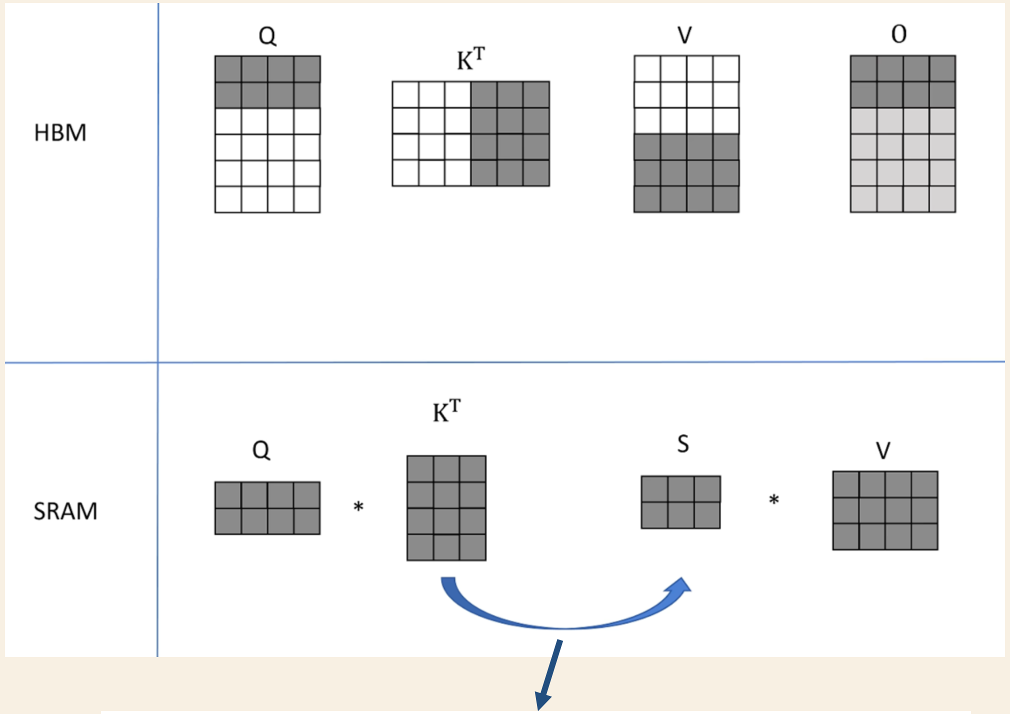
2. Algorithm
Outer Loop



2. Algorithm
Outer Loop

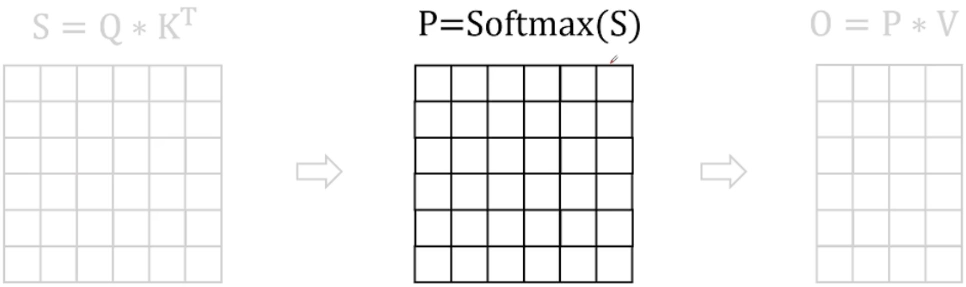


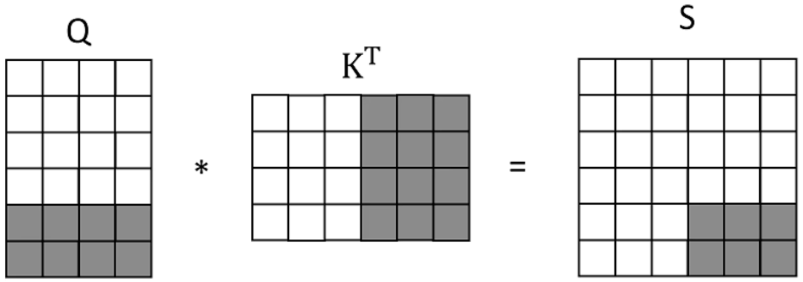
2. Algorithm
Softmax



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

$$\text{softmax}(\{x_1,\ldots,x_N\}) = \{\frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}\}_{i=1}^{N}$$





2. Algorithm
Safe Softmax

$$x = [x_1, \dots, x_N]$$

$$m(x) := max(x)$$

$$p(x) := [e^{x_1 - m(x)}, \dots, e^{x_N - m(x)}]$$

$$l(x) := \sum_i p(x)_i$$

$$softmax(x) := \frac{p(x)}{l(x)}$$

Before introducing the solution, we first introduce the concept of safe softmax.

For FP16, max number is 66536 < e^{12}

So, Safe Softmax is proposed to avoid overflow.

2. Algorithm
Online Softmax

$$x = [x_1, \dots, x_N]$$

$$m(x) := max(x)$$

$$p(x) := [e^{x_1 - m(x)}, \dots, e^{x_N - m(x)}]$$

$$l(x) := \sum_i p(x)_i$$

$$softmax(x) := \frac{p(x)}{l(x)}$$

$$x = [x_1, \dots, x_N, \dots x_{2N}]$$

$x^1 = [x_1, \dots, x_N]$	$x^2 = [x_{N+1}, \dots x_{2N}]$
$m(x^1) \quad p(x^1) \quad l(x^1)$	$m(x^2) \quad p(x^2) \quad l(x^2)$

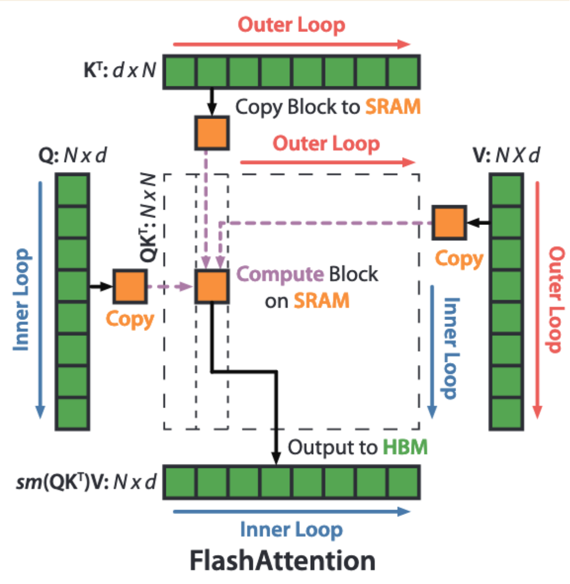
$$m(x) := max(m(x^1),\ m(x^2))$$

$$p(x) := [e^{m(x^1) - m(x)}\, p(x^1), e^{m(x^2) - m(x)}\, p(x^2)]$$

$$l(x) := e^{m(x^1) - m(x)}\, l(x^1) + e^{m(x^2) - m(x)}\, l(x^2)$$

$$softmax(x) := \frac{p(x)}{l(x)}$$

2. Algorithm Pseudocode



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

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 $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ 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 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 $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

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

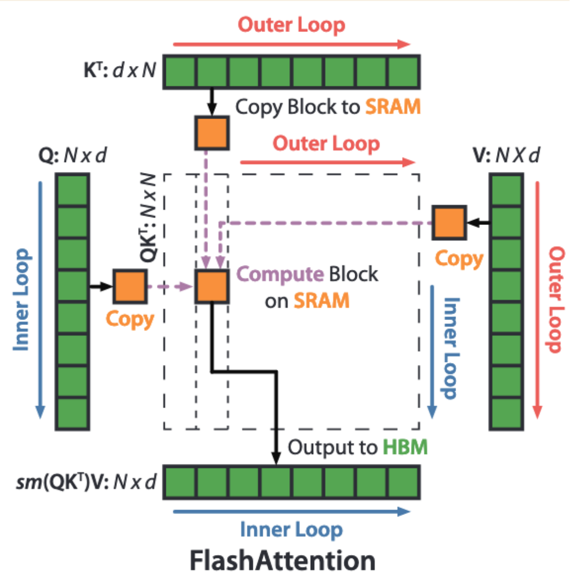
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

14: **end for**

15: **end for**

16: Return $\mathbf{O}$.

2. Algorithm
Pseudocode



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

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 $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ 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.

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

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

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

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

  end for

end for

Return $\mathbf{O}$.

2. Algorithm
Extension: Block-Sparse FlashAttention

"skip-if-zero" logic:

- Built on top of FlashAttention by adding a **block-level sparsity mask**
- Attention is only computed for **blocks where the mask is 1**
- **Zero blocks are skipped**, saving computation and memory access

3. Result
Faster

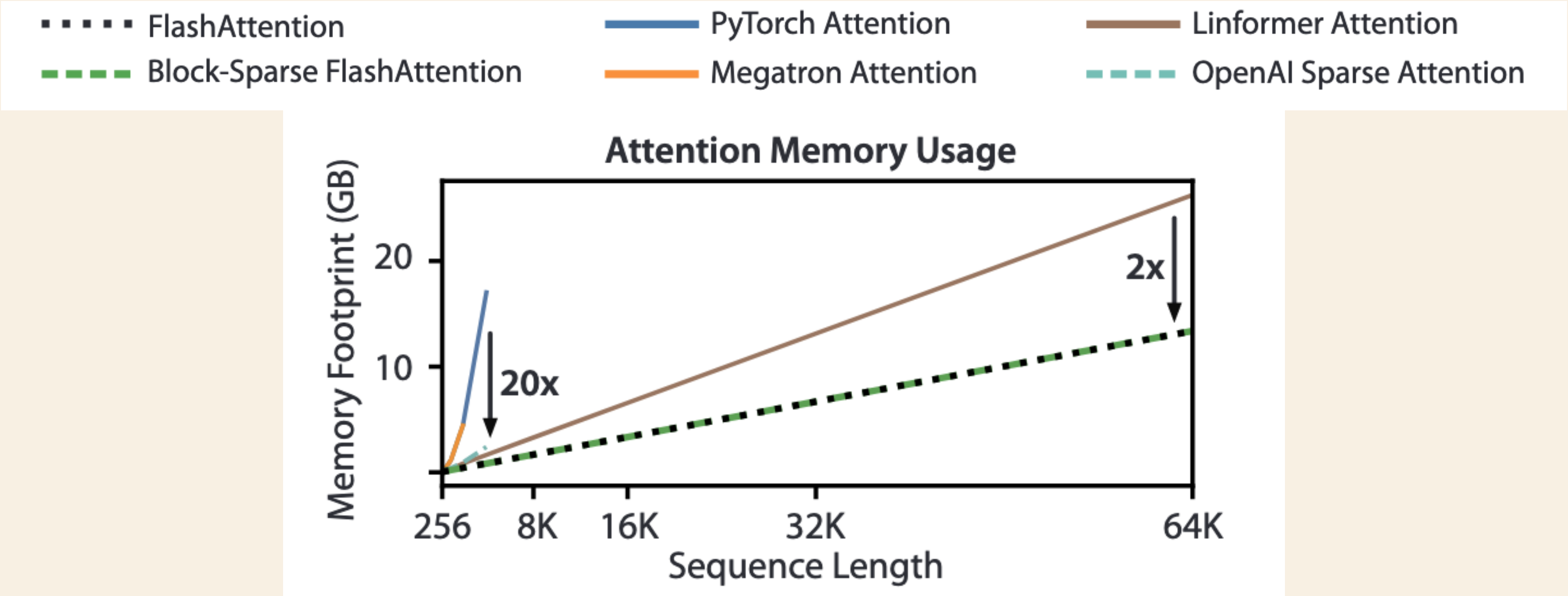
BERT Implementation	Training time (minutes)
Nvidia MLPerf 1.1 [58]	20.0 ± 1.5
FLASHATTENTION (ours)	**17.4** ± 1.4

BERT

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×)**

GPT-2

3. Result
Less Memory



FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning

1. Recap of V1
Problem

- As context length increases, FlashAttention's efficiency lags behind primitives like GEMM:

- Flash Attention
  - Forward pass: 30-50% of theoretical max FLOPs/s on A100 GPU.
  - Backward pass: 25-35% of theoretical max FLOPs/s on A100 GPU.
- GEMM
  - achieves **80-90%** of theoretical max throughput.

1. Recap of V1
Root Cause

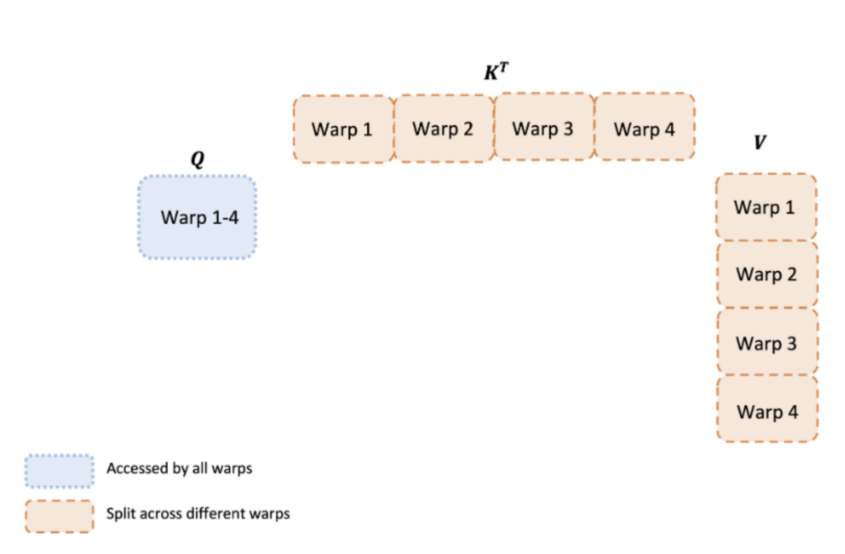
Suboptimal Work Partitioning on GPU

- **Low Occupancy**: Limited parallelism over batch size and number of heads, especially for long sequences with small batch sizes.
- **Unnecessary Shared Memory Access**: Inefficient distribution of work between thread blocks and warps leads to excessive shared memory reads/writes.

2. Solution
a) Better Work Partitioning

- **GPU Work Partitioning**: Each thread block contains multiple warps (e.g., 4 or 8 warps, 32 threads per warp).
- **Goal**: Optimize work distribution to minimize shared memory access and synchronization overhead.

2. Solution
a) Better Work Partitioning



(a) FlashAttention

Work Distribution: K^T and V are split across 4 warps (Warp 1-4).

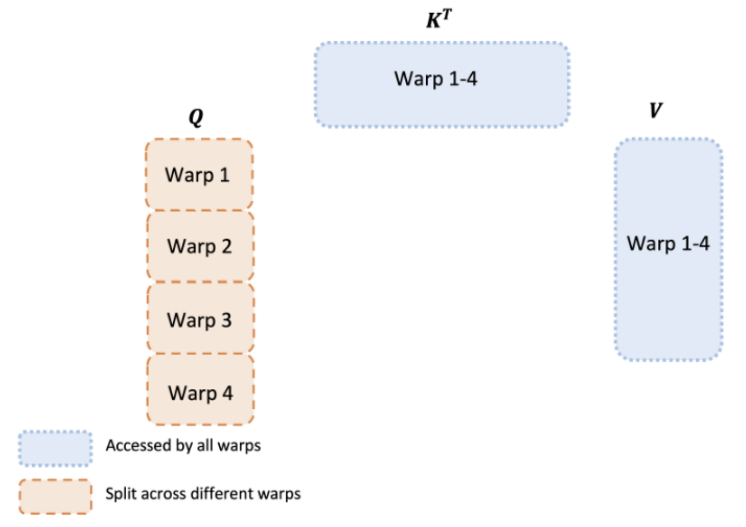
- QK^T is accessed by all warps.

Computation: Each warp computes a slice of QK^T ("sliced-K").

- Results are written to shared memory, synchronized, and accumulated to compute the output.

Drawback: High shared memory reads/writes and synchronization overhead.

2. Solution
a) Better Work Partitioning



(b) FlashAttention-2

Work Distribution: Q is split across 4 warps (Warp 1-4).

- QK^T and V are accessed by all warps.

Computation: Each warp computes a slice of QK^T , then directly multiplies with V. Outputs are independent, requiring no shared memory communication or synchronization.
Advantage: Reduced shared memory access, lower latency, and ~2× speedup over v1.

2. Solution
b) Reduce Non-matmul Flops

GPU Hardware Insight

- **Tensor Cores (A100 GPU)**:
  - FP16/BF16 matmul: 312 TFLOPs/s.
  - FP32 non-matmul: 19.5 TFLOPs/s.
- **Cost**: Non-matmul FLOP is ~16× more expensive than matmul FLOP.

FlashAttention v1 Issue

- Non-matmul FLOPs (e.g., softmax, rescaling) bottleneck performance.

Optimizing Online Softmax: reduce the number of rescaling operations, boundary checks, and causal masking operations, without altering the output.

2. Solution
c) Better Parallelization

v1

- Parallelizes over batch size and heads.
- 1 thread block per head (Batch Size × Heads).
- A100 GPU (108 SMs): Efficient if thread blocks ≥ 80.
- **Issue**: Long sequences → small batch/heads → low occupancy.

v2 Solution

- Adds parallelization along sequence length:
- **Result**: Better GPU utilization, up to 2× speedup.

3. Result
A100

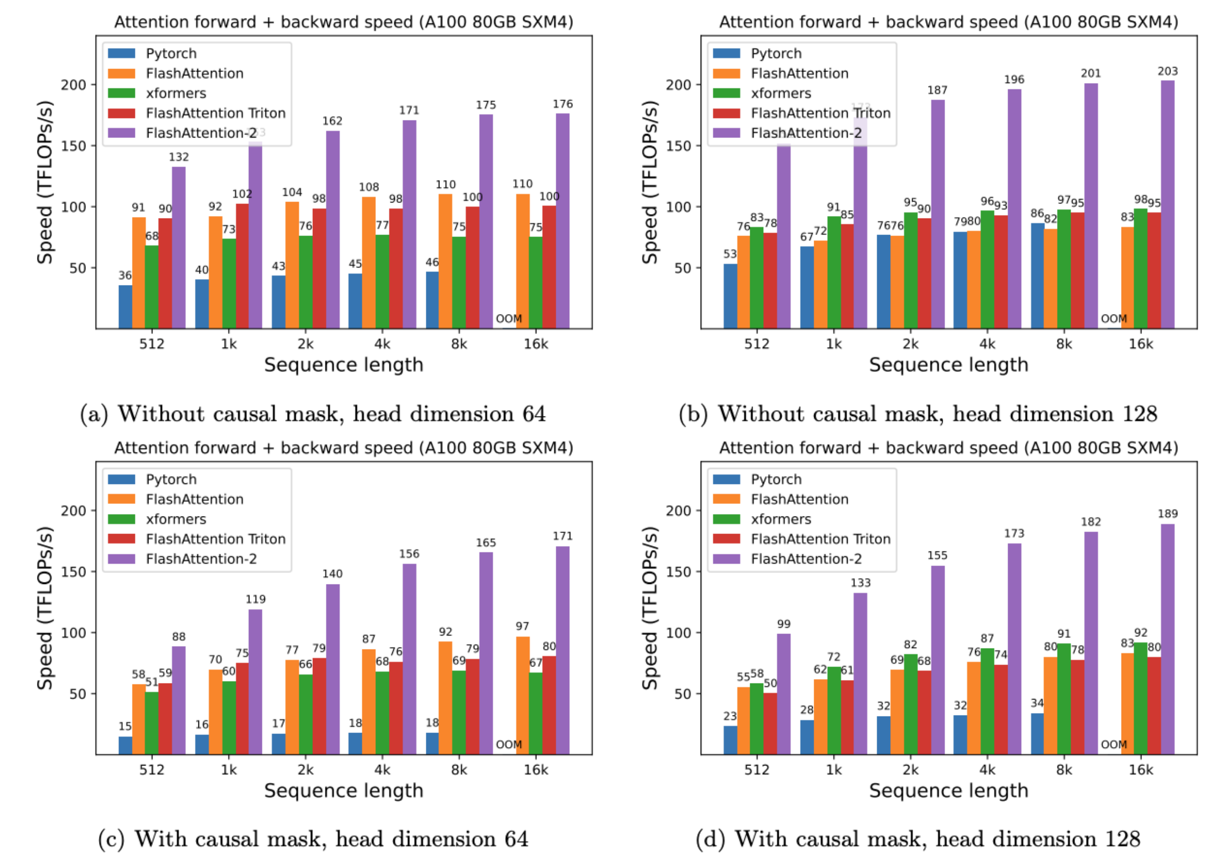
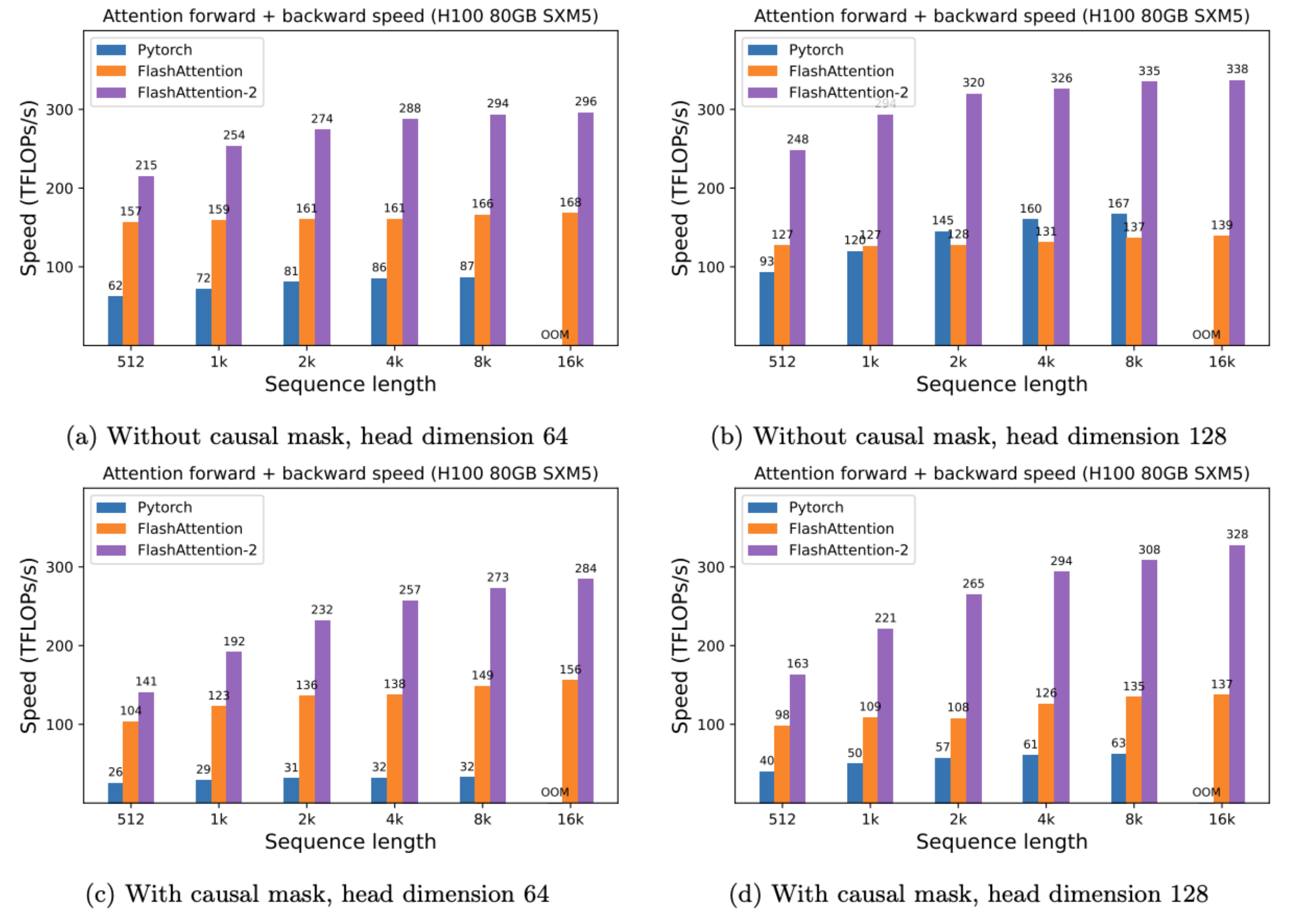


Figure 4: Attention forward + backward speed on A100 GPU

2x Faster than v1;
~9x faster than PyTorch

3. Result
H100



(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

Figure 7: Attention forward + backward speed on H100 GPU

Thank you!