Optimizing Attention for Modern Hardware

Tri Dao
https://tridao.me

Motivation: Modeling Long Sequences

Enable New Capabilities

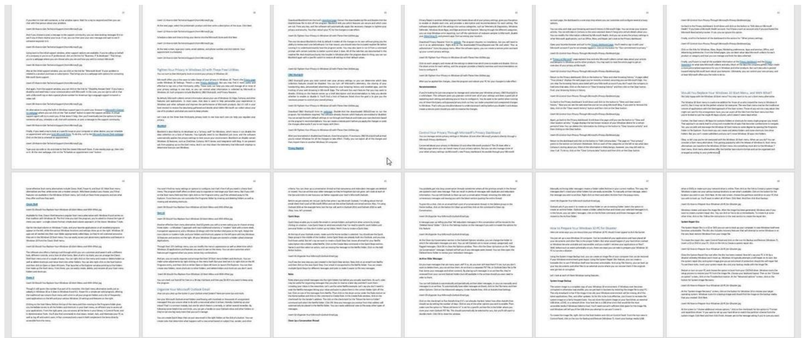
NLP: Large context required to understand books, plays, codebases.

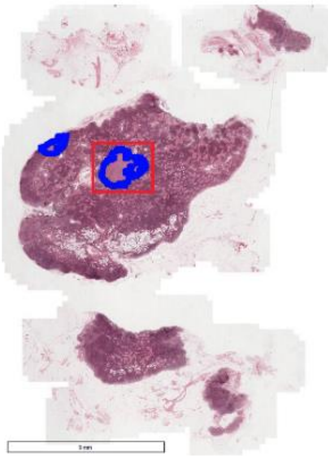
Close Reality Gap

Computer vision: higher resolution can lead to better, more robust insight.

Open New Areas

Time series, audio, video, medical imaging data naturally modeled as sequences of millions of steps.

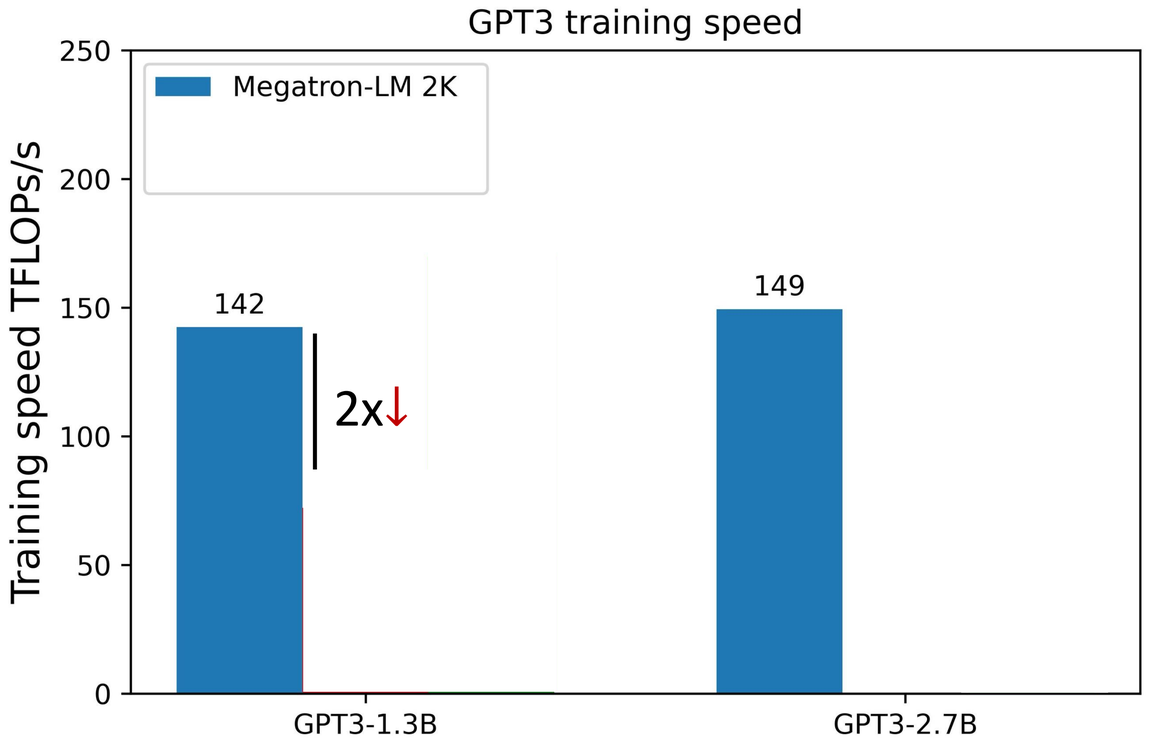




Efficiency is the Bottleneck for Modeling Long Sequences with Attention

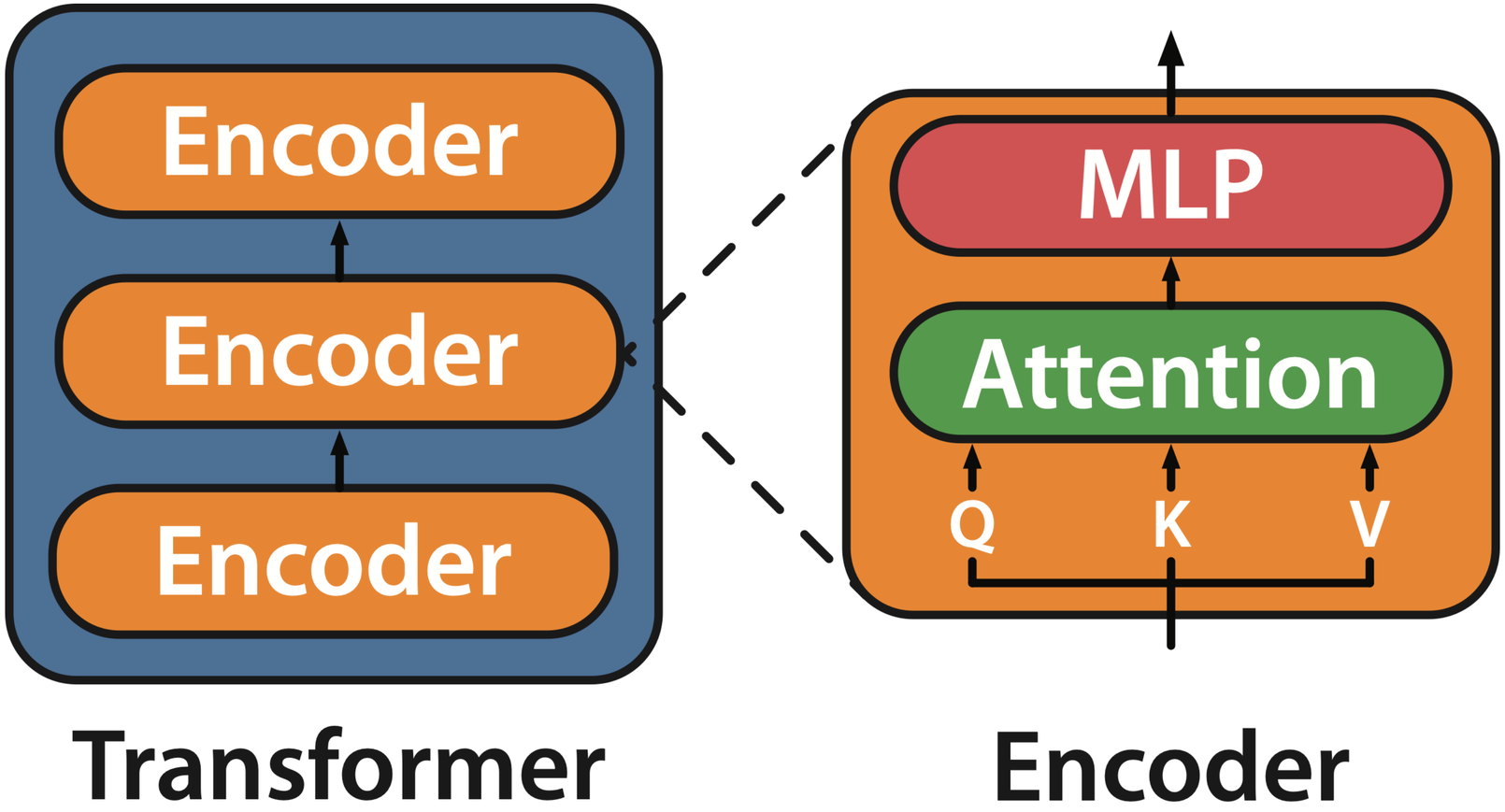
Context length: how many other elements in the sequence does the current element interact with.

Increasing context length slows down (or stops) training

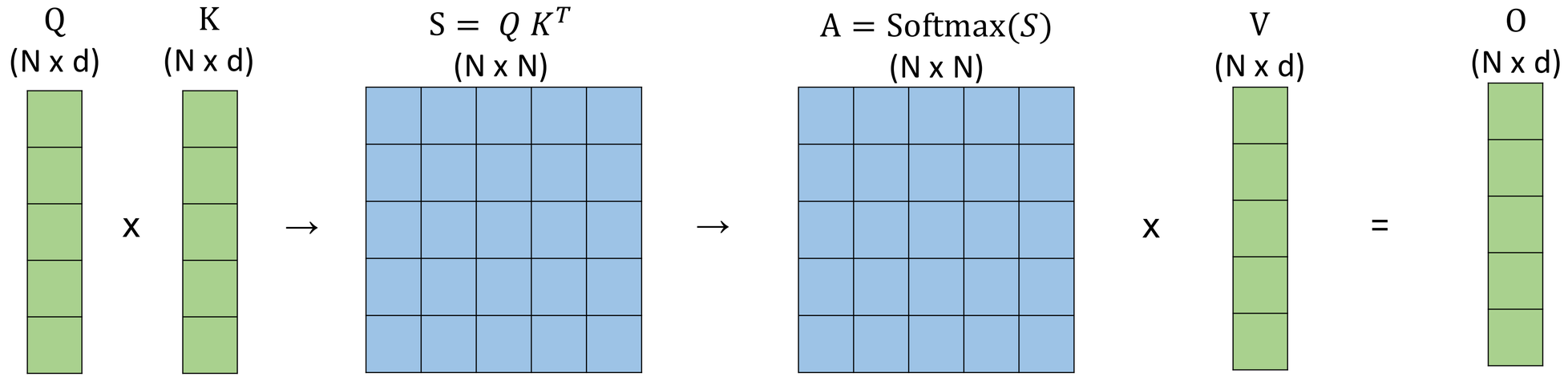


How to efficiently scale models to longer sequences?

Background: Attention is the Heart of Transformers



Background: Attention Mechanism



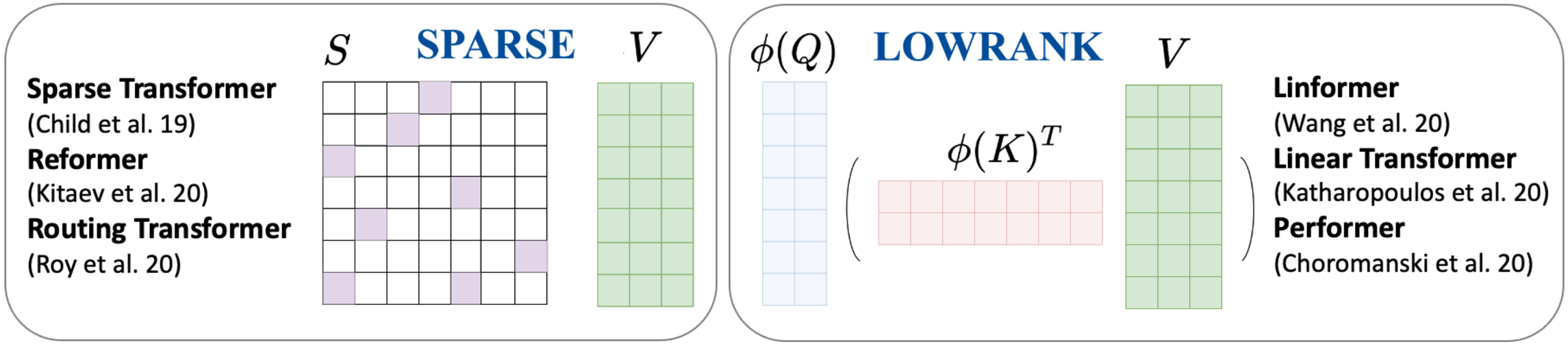
Typical sequence length N: 1K – 8K
Head dimension d: 64 – 128

$$\text{Softmax}([s_1, \cdots, s_N]) = \left[\frac{e^{s_1}}{\sum_i e^{s_i}}, \cdots, \frac{e^{s_N}}{\sum_i e^{s_i}}\right]$$

$$O = \text{Softmax}(QK^T)V$$

Attention scales quadratically in sequence length N

Background: Approximate Attention

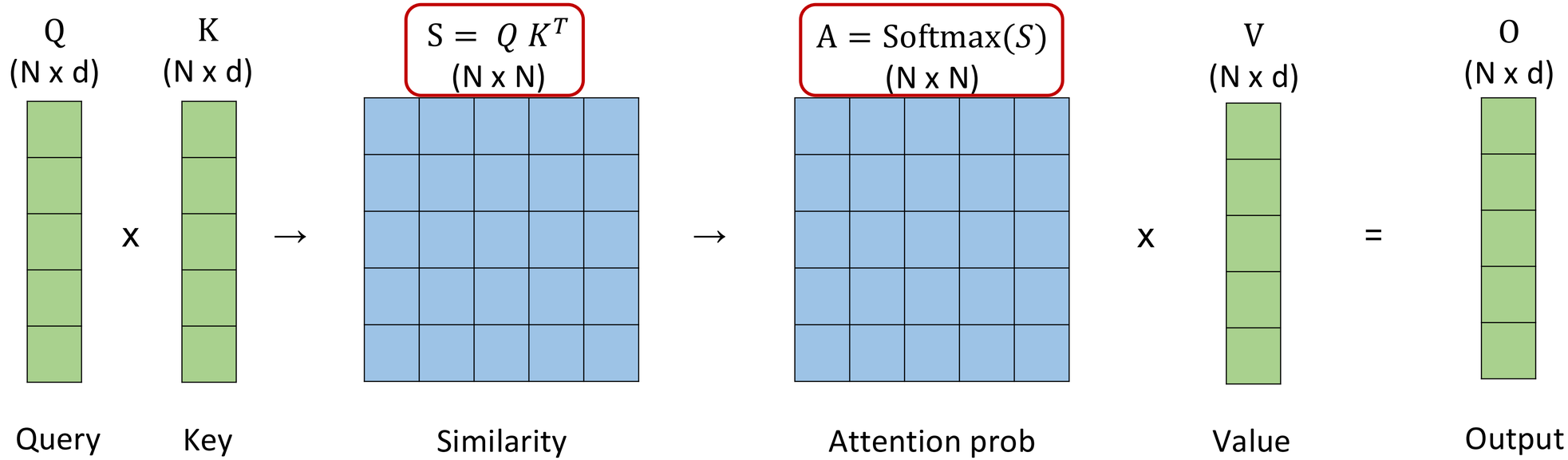


Approximate attention: tradeoff quality for ~~speed~~ fewer FLOPs

Survey: Tay et al. Long Range Arena : A Benchmark for Efficient Transformers. ***ICLR .2020***

Is there a fast, memory-efficient, and exact attention algorithm?

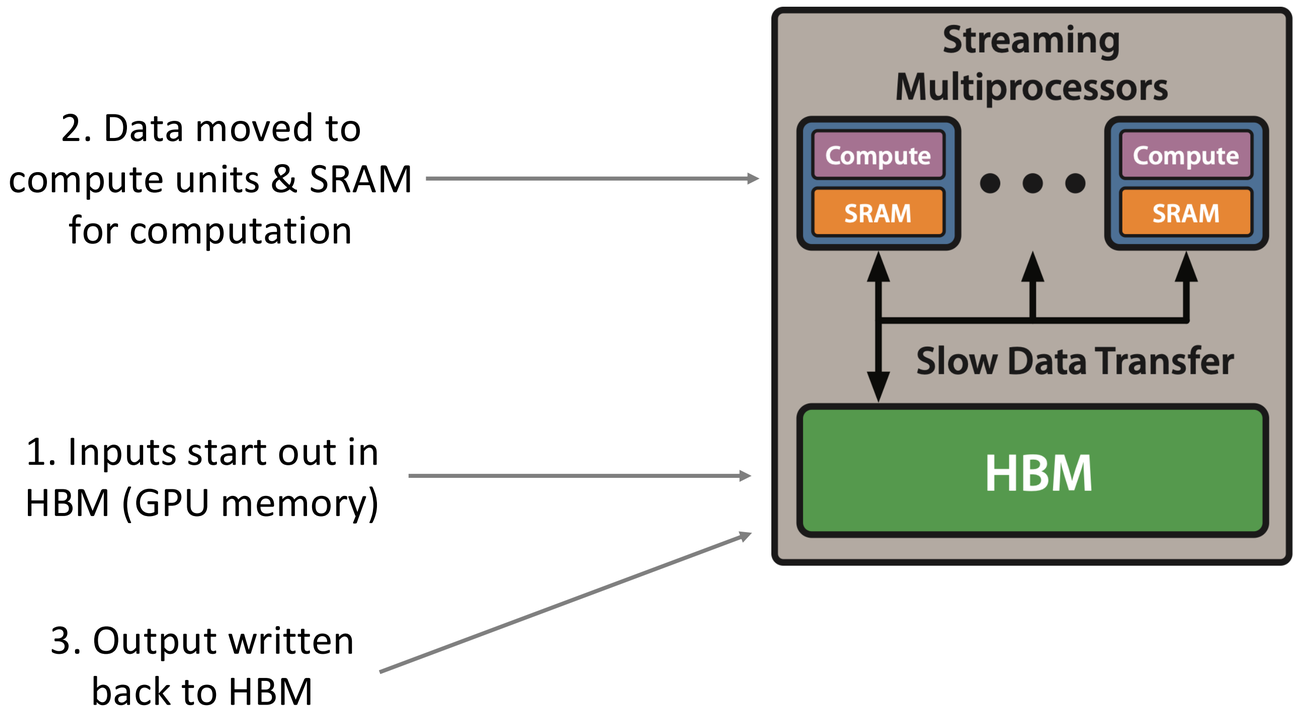
Our Observation: Attention is Bottlenecked by Memory Reads/Writes

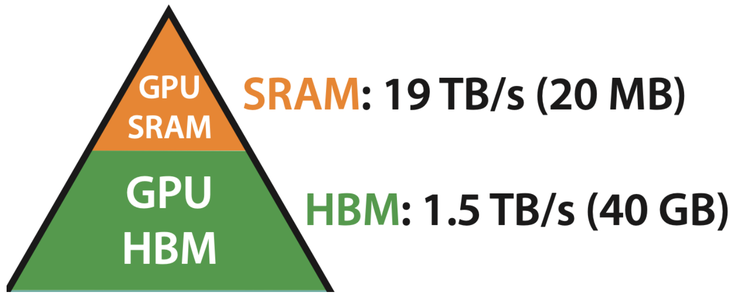


Typical sequence length N: 1K – 8K
Head dimension d: 64-128

The biggest cost is in moving the bits!
Standard implementation requires repeated R/W from slow GPU memory

Background: GPU Compute Model & Memory Hierarchy





Blogpost: Horace He, Making Deep Learning Go Brrrr From First Principles.

Can we exploit the memory asymmetry to get speed up?
With IO-awareness (accounting for R/W to different levels of memory)

How to Reduce HBM Reads/Writes: Compute by Blocks

Challenges:

(1) Compute softmax normalization without access to full input.

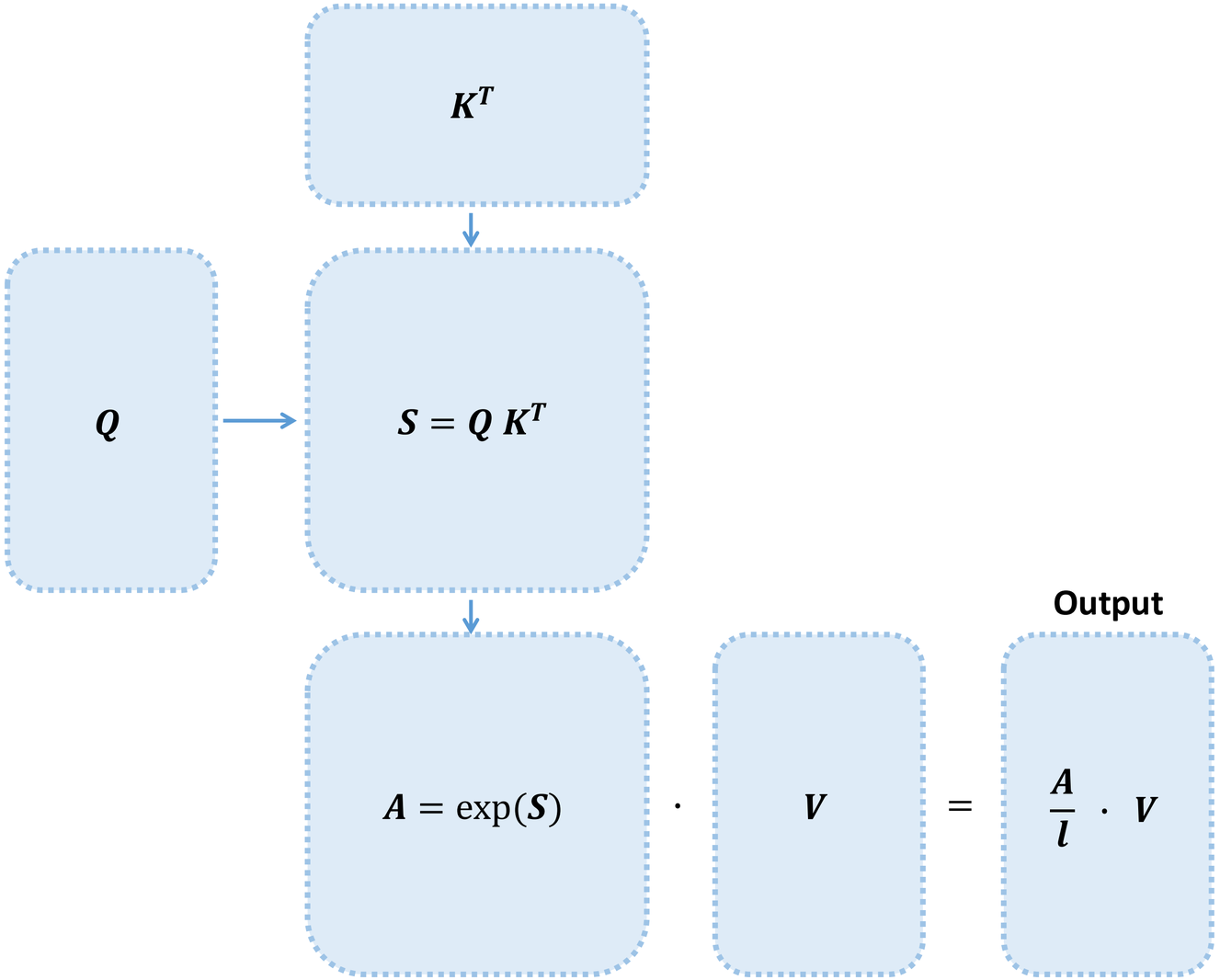
(2) Backward without the large attention matrix from forward.

Approaches:

(1) Tiling: Restructure algorithm to load block by block from HBM to SRAM to compute attention.

(2) Recomputation: Don’t store attn. matrix from forward, recompute it in the backward.

Attention Computation Overview



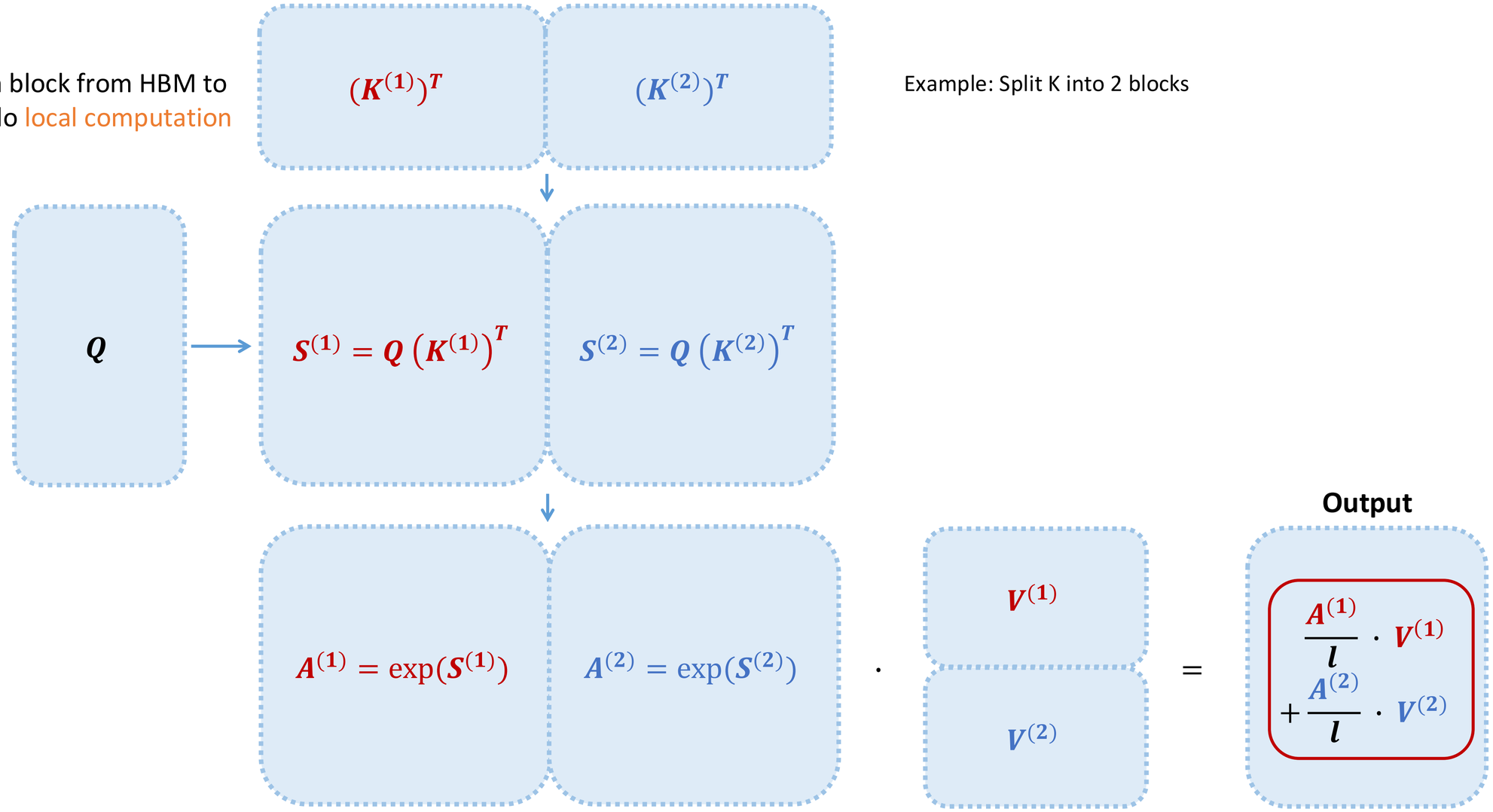
Softmax row-wise normalization constant

$$l = \sum_i \exp(S)_i$$

Compute by blocks: easy to split Q, but how do we split K & V?

Tiling – 1st Attempt: Computing Attention by Blocks

Goal:
Load each block from HBM to SRAM & do local computation



Example: Split K into 2 blocks

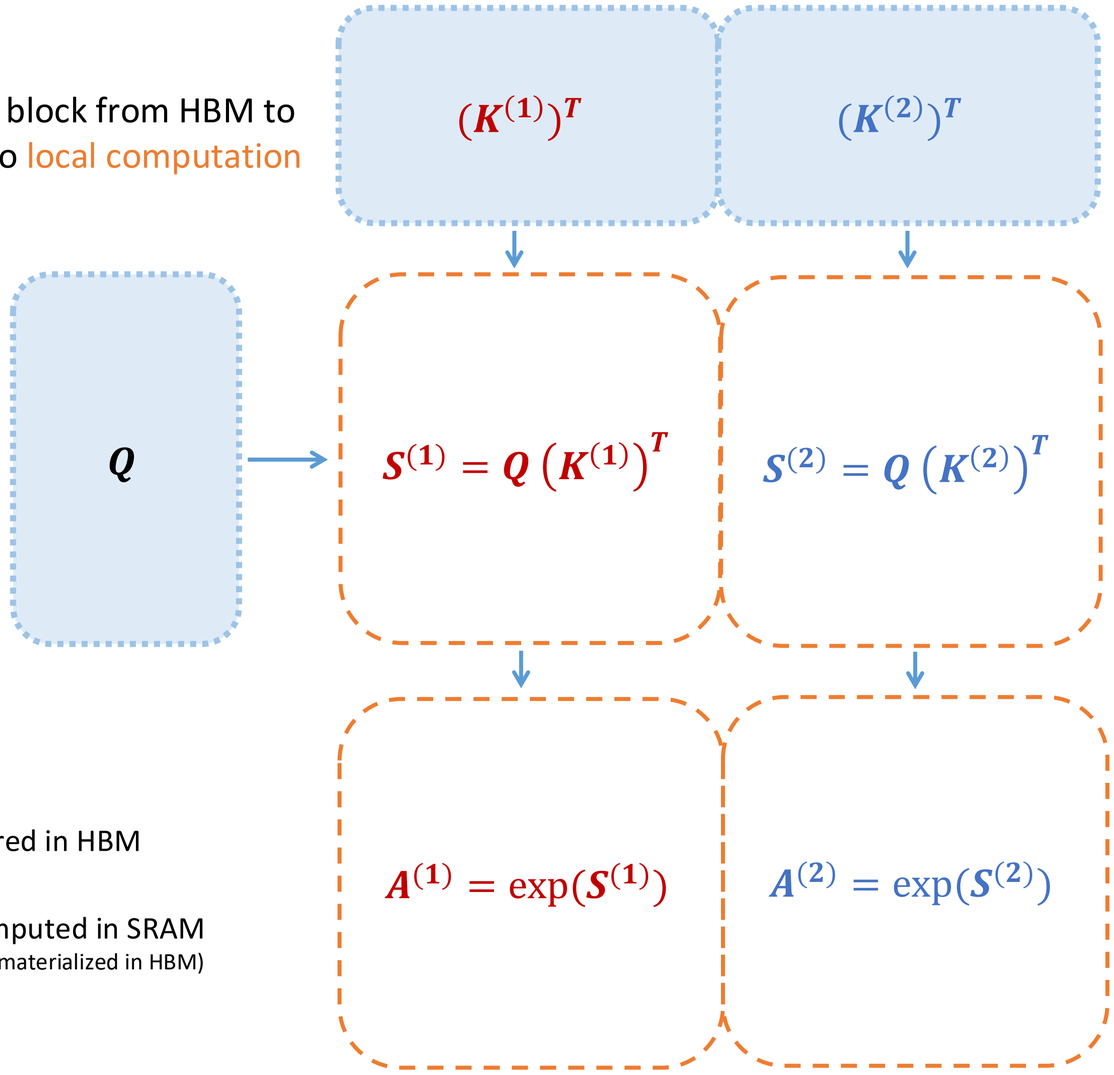
Softmax row-wise normalization constant

$$l = \sum_i \exp\left(S^{(1)}\right)_i + \sum_i \exp\left(S^{2}\right)_i$$

Challenge: How to compute softmax normalization with just local results?

Tiling – 2nd Attempt: Computing Attention by Blocks, with Softmax Rescaling

Goal:
Load each block from HBM to SRAM & do local computation



Output we want: $l = \sum_i \exp(S^{(1)})_i + \sum_i \exp(S^2)_i$

$$O = \frac{A^{(1)}}{l} \cdot V^{(1)} + \frac{A^{(2)}}{l} \cdot V^{(2)}$$

$$\cdot \begin{matrix} V^{(1)} \\ V^{(2)} \end{matrix} =$$

Output

$$O^{(1)} = \frac{A^{(1)}}{l^{(1)}} \cdot V^{(1)}$$

$$O^{(2)} = \frac{l^{(1)}}{l^{(2)}} O^{(1)} + \frac{A^{(2)}}{l^{(2)}} \cdot V^{(2)}$$

Wrong denominator ☹

Local computation

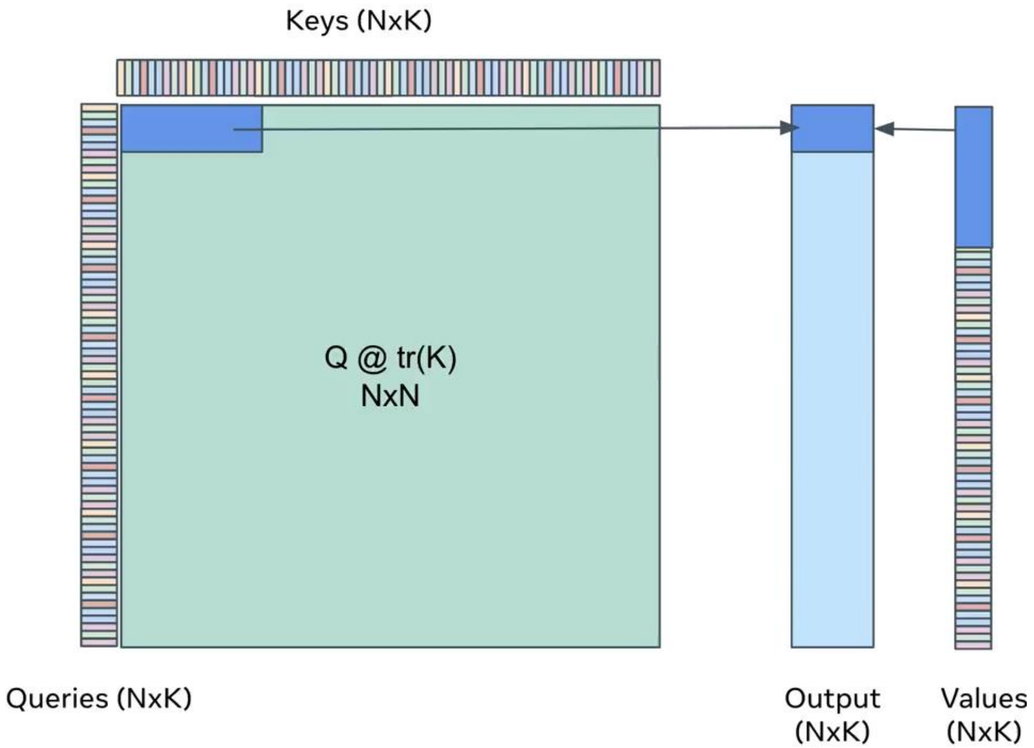
Rescaling to correct denominator

$$l^{(1)} = \sum_i \exp(S^{(1)})_i \qquad l^{(2)} = l^{(1)} + \sum_i \exp(S^{(2)})_i$$

Tiling + Rescaling allows local computation in SRAM, without writing to HBM, and get the right answer!

Tiling

Decomposing large softmax into smaller ones by scaling.



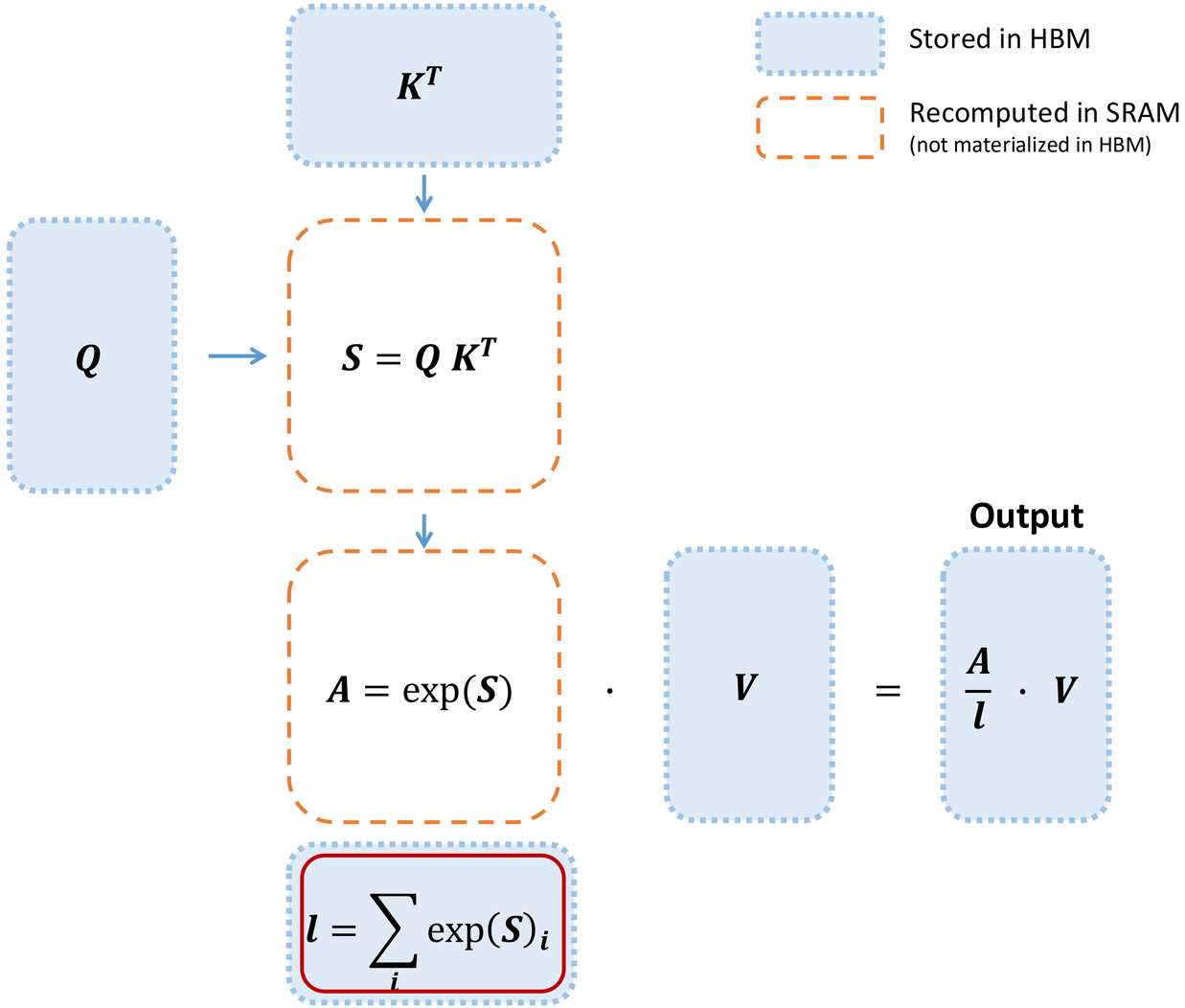
1. Load inputs by blocks from HBM to SRAM.
2. On chip, compute attention output with respect to that block.
3. Update output in HBM by scaling.

Animation credit: Francisco Massa

Recomputation (Backward Pass)

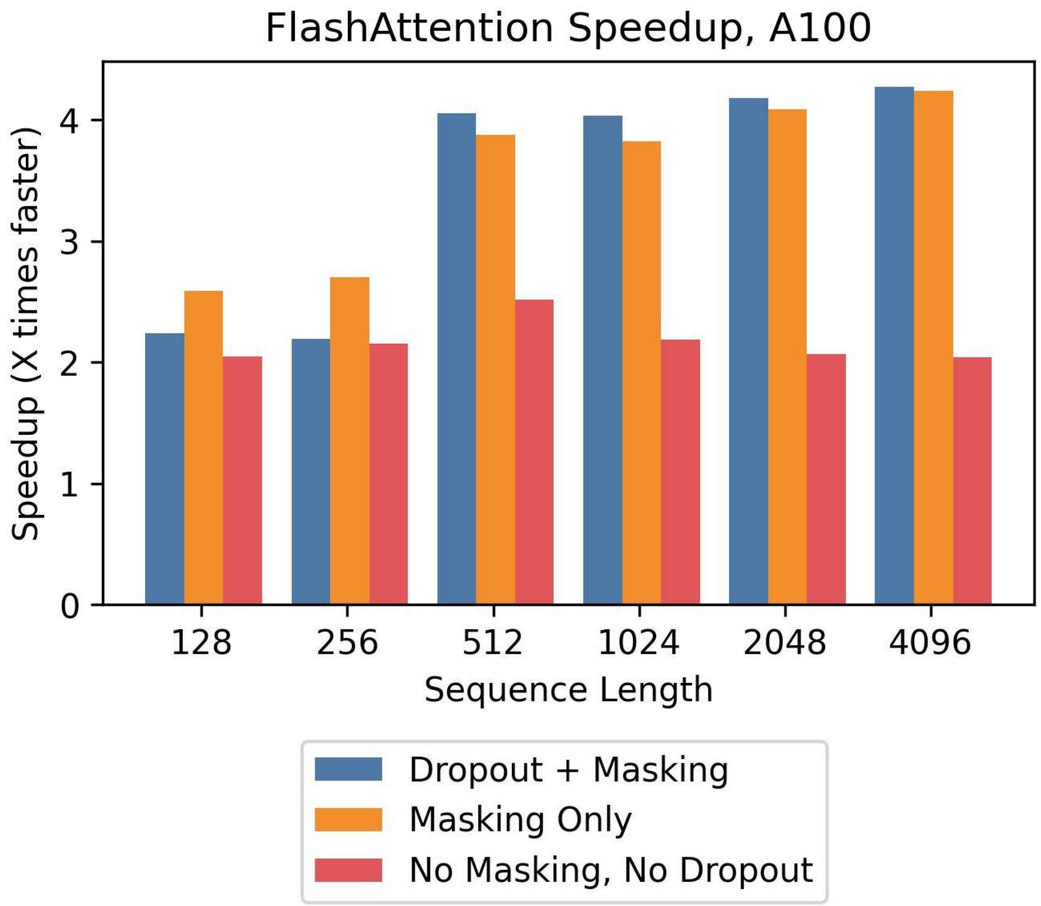
By storing softmax normalization from forward (size N), quickly recompute attention in the backward from inputs in SRAM.

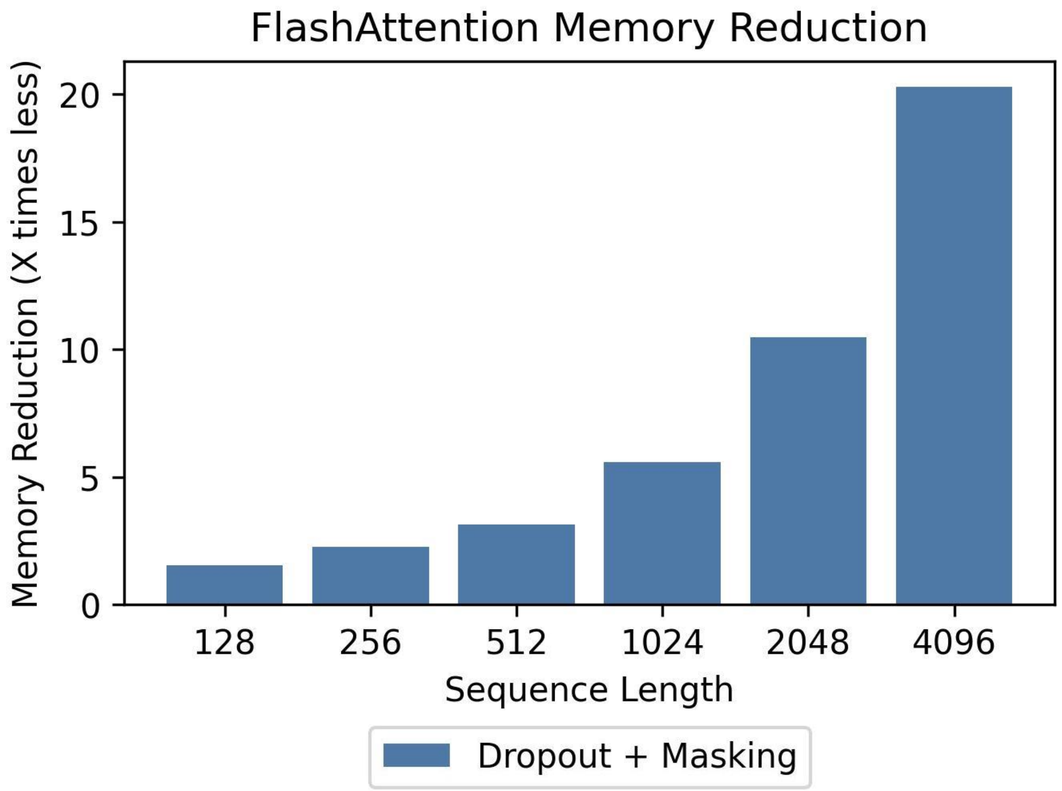
Attention	Standard	FlashAttention
GFLOPs	66.6	75.2 (↑13%)
HBM reads/writes (GB)	40.3	4.4 (↓9x)
Runtime (ms)	41.7	7.3 (↓6x)



FlashAttention speeds up backward pass even with increased FLOPs.

FlashAttention: 2-4x speedup, 10-20x memory reduction





2-4x speedup — with no approximation!

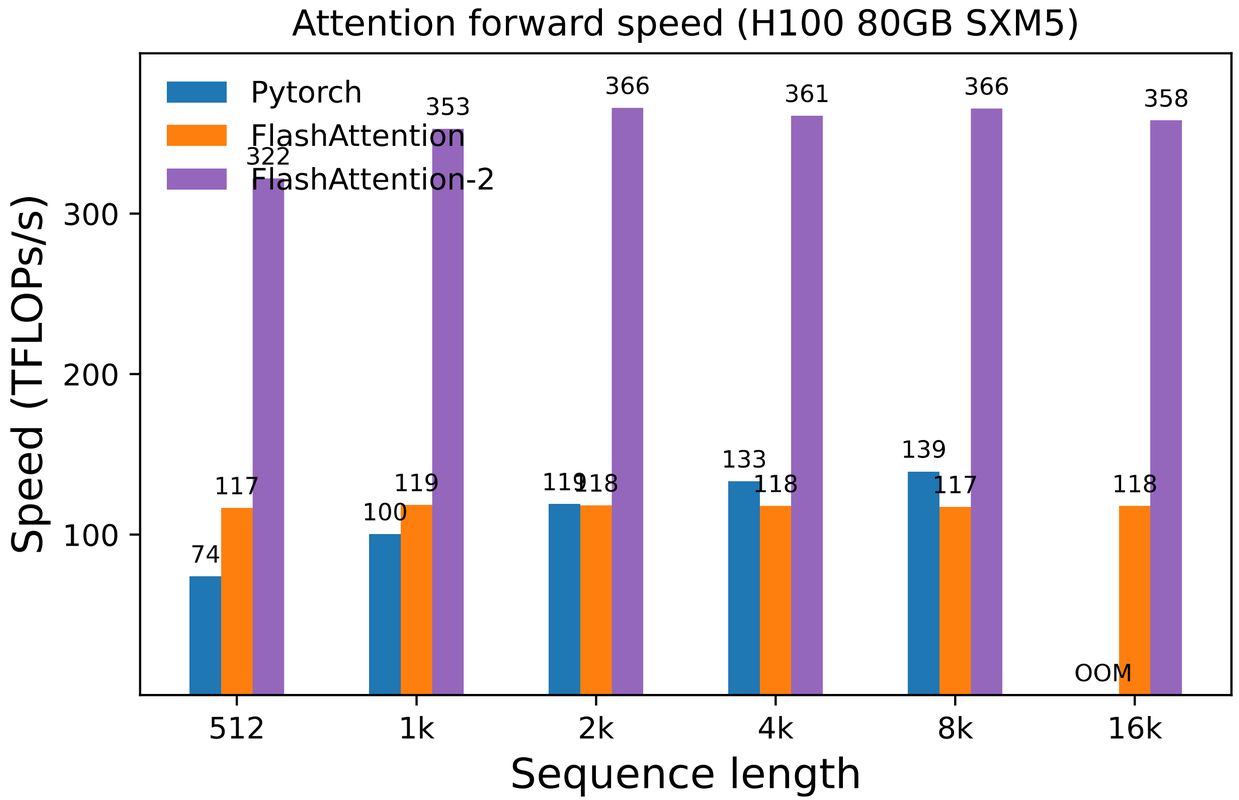
10-20x memory reduction — memory linear in sequence length

How has modern hardware changed and how can we redesign attention?

Challenge: Optimizing FlashAttention for Modern Hardware

FlashAttention-2 is highly optimized on A100, reaching 70% utilization

But, FA-2 only gets to 35-40% utilization on H100!



FlashAttention-3: Optimizing FlashAttention for Hopper Architecture

Jay Shah*, Ganesh Bikshandi*, Ying Zhang, Vijay Thakkar, Pradeep Ramani, Tri Dao

1. **New instructions** on H100:
   - **WGMMA**: higher throughput MMA primitive, async
   - **TMA**: faster loading from gmem <-> smem, async, saves registers

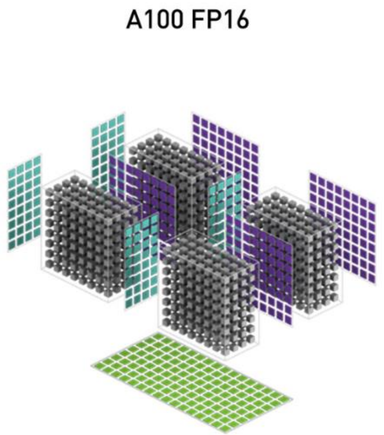
2. **Asynchrony**
   - Inter-warpgroup overlapping: warp-specialization, pingpong scheduling
   - Intra-warpgroup overlapping: softmax and async matmul

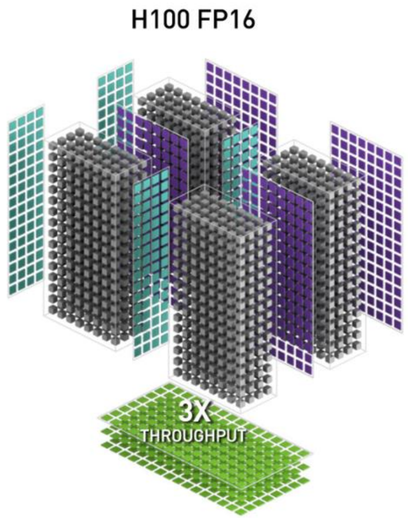
3. **Low-precision** – FP8
   - Solve for layout conformance, in-kernel V transpose

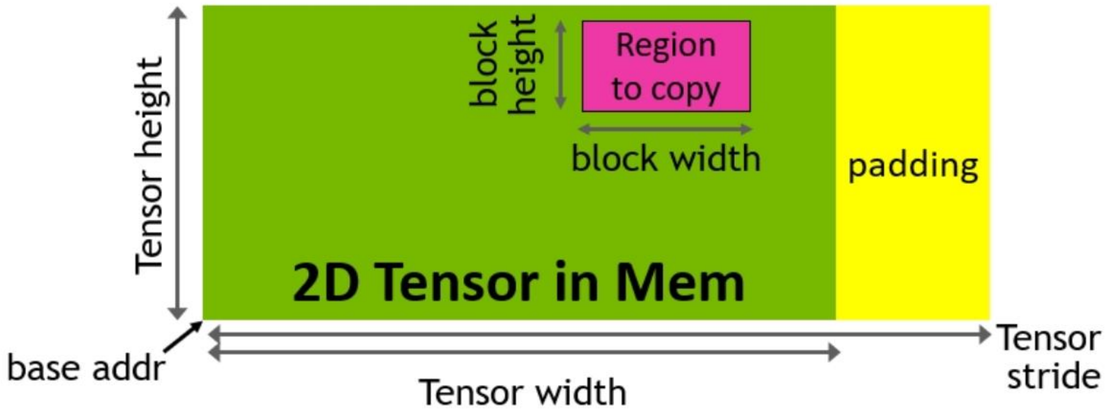
Plus: Persistent kernels, LPT scheduling for causal attention, GQA packing, MLA

Upshot: 1.6-3x speedup on **Hopper**, algorithmic ideas apply for **Blackwell**

New Instructions on Hopper: WGMMA & TMA







wgmma necessary, mma.sync can only reach 2/3 peak throughput

TMA: accelerate gmem -> smem copy, saves registers

Both WGMMA and TMA are **asynchronous** instructions:
threads issue them and can then do other work while they execute.

Asynchrony: Overlapping GEMM and Softmax

Why overlap?

Special Function Units (SFU) have very low throughput relative to Tensor Cores.

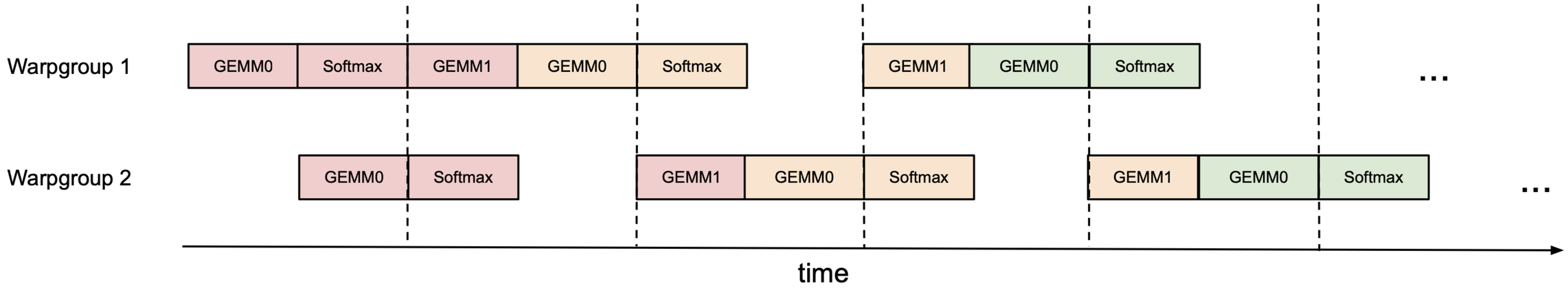
Example: headdim 128, block size 128 x 128
FP16 WGMMA: 2 x 2 x 128 x 128 x 128 = 8.4 MFLOPS, 4096 FLOPS/cycle -> **2048 cycles**
MUFU.EX2: 128 x 128 = 16k OPS, 16 OPS/cycle -> **1024 cycles**

MUFU.EX2 takes 50% the cycles of WGMMA!
FP8, or Blackwell is even worse: WGMMA and EX2 both take 1024 cycles.
We want to be doing EX2 while tensor cores are busy with WGMMA.

Inter-warpgroup Overlapping of GEMM and Softmax

Easy solution: leave it to the warp schedulers!

This works reasonably well, but we can do better.

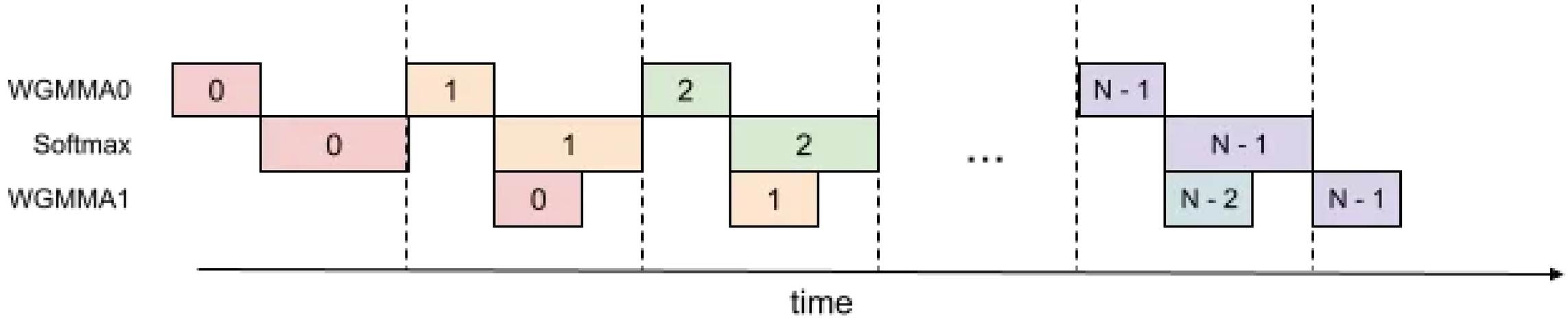


Pingpong scheduling using synchronization barriers (with bar.sync):
580 TFLOPS -> 640 TFLOPS

Intra-warpgroup Overlapping of GEMM and Softmax

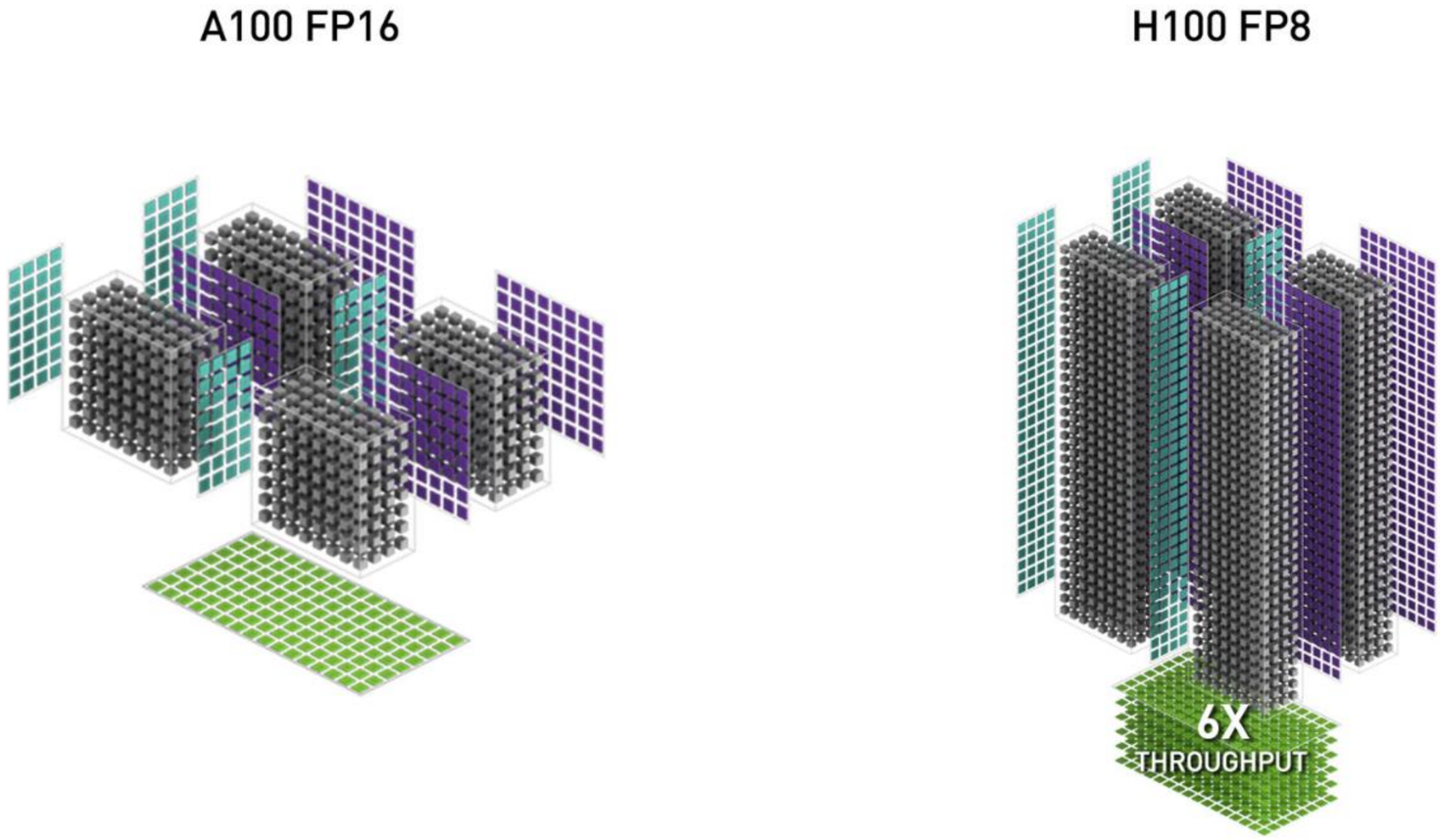
Per warpgroup, can finally exploit **asynchrony** of WGMMA.

- Overlap GEMM1 for kth iteration with softmax for (k+1)th iteration.
- Uses more registers since accumulator for next GEMM0 and operand for current GEMM1 are now live concurrently.



2-stage intra-warpgroup overlapping: 640 TFLOPS -> 670 TFLOPS

Low-precision: FP8



FP8 Tensor Cores double WGMMA throughput, but trade off accuracy

FP8 Attention with Incoherent Processing

Can multiply Q and K with a random orthogonal matrix (Hadamard) to "spread out" the outliers.
Note: S = Q.K^T = (Q J)(K J)^T for orthogonal matrix J since J.J^T = I by definition.

Reduces quantization error by 2.6x on normally distributed QKV data with 0.1% entries given large magnitude (to simulate outliers).

Method	Baseline FP16	FLASHATTENTION-2 FP16	FLASHATTENTION-3 FP16
RMSE	3.2e-4	**1.9e-4**	**1.9e-4**

Method	Baseline FP8	FLASHATTENTION-3 FP8	No block quant	No incoherent processing
RMSE	2.4e-2	**9.1e-3**	9.3e-3	2.4e-2

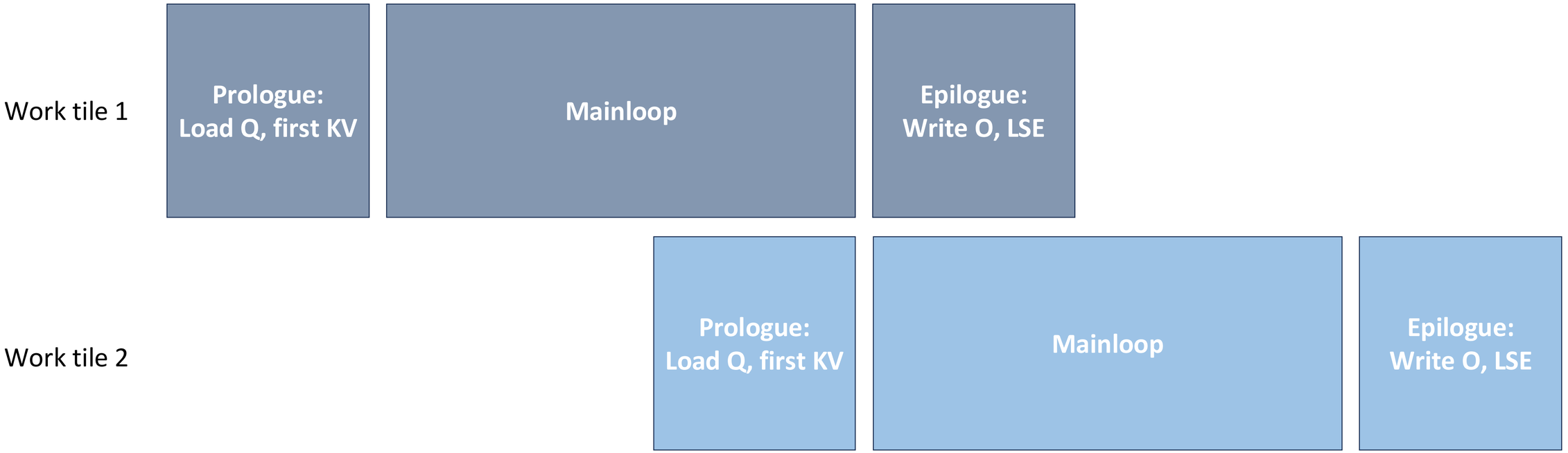
Persistent Kernels: Hiding Prologue & Epilogue

Motivation: Tensor Cores are so fast, Prologue/Epilogue latency become non-trivial.

Idea: Fixed number of CTAs (= num SMs), **persistent** across work tiles.

- Asynchronous TMA store to overlap epilogue, prologue, and mainloop.

Persistent Kernels: Hiding Prologue & Epilogue

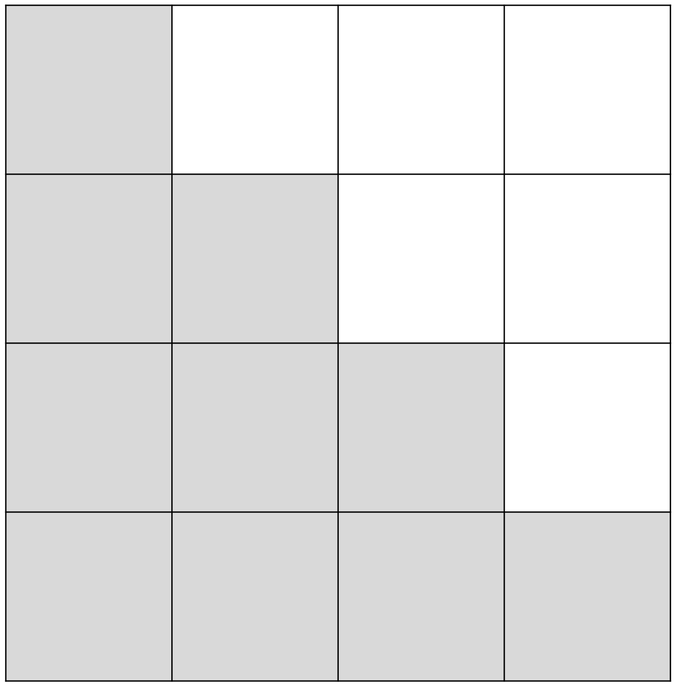


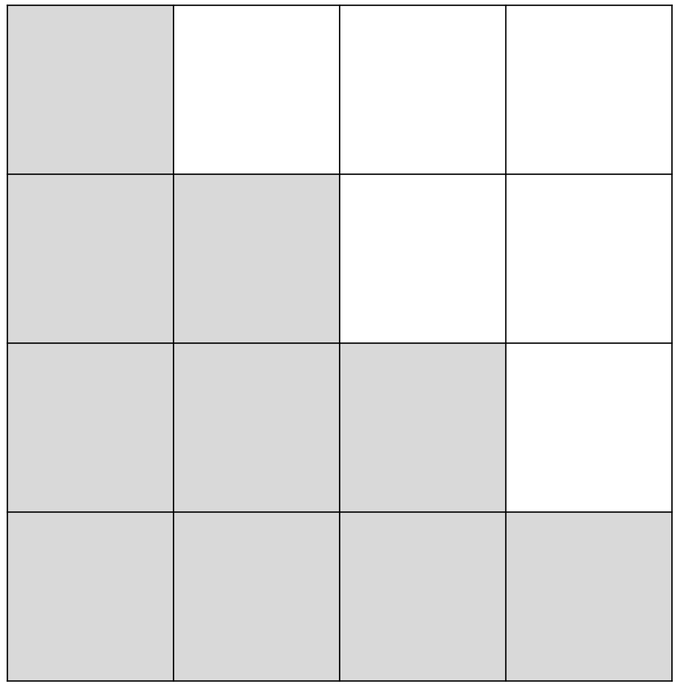
Persistent kernels: 670 TFLOPS -> 700 TFLOPS

Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)



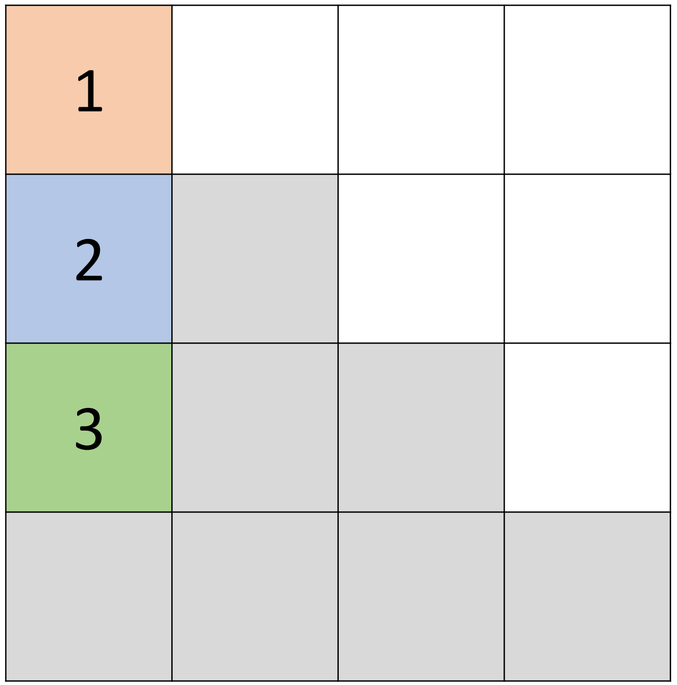


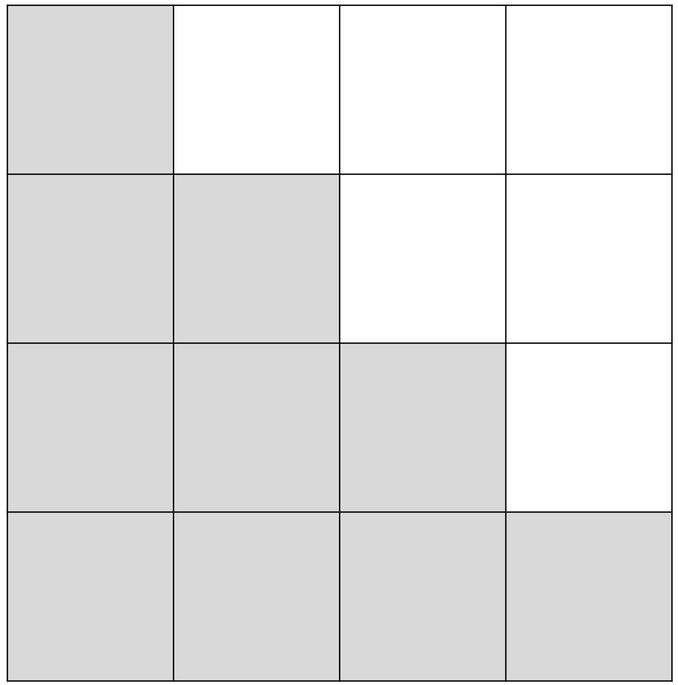
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 1



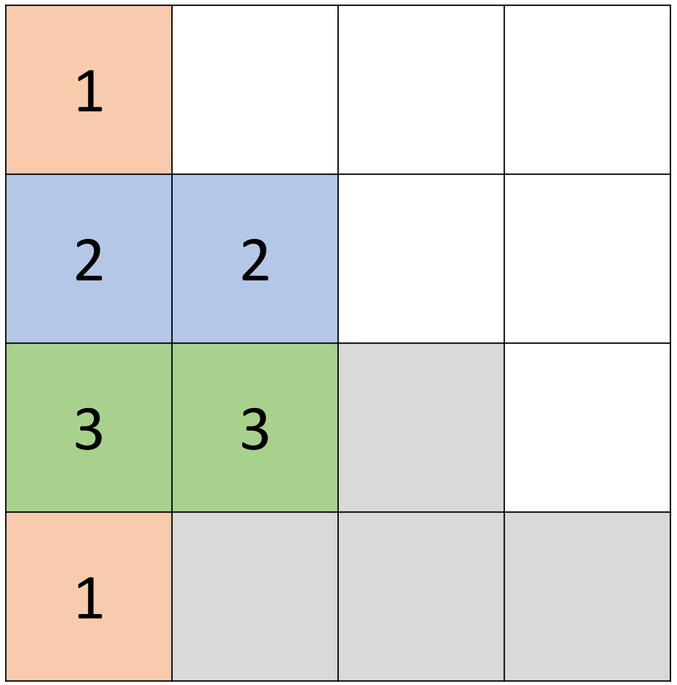


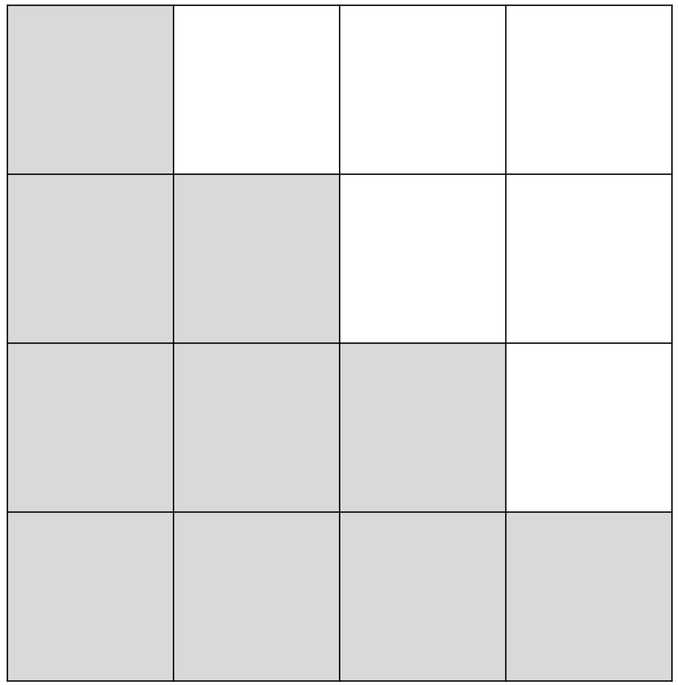
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 2



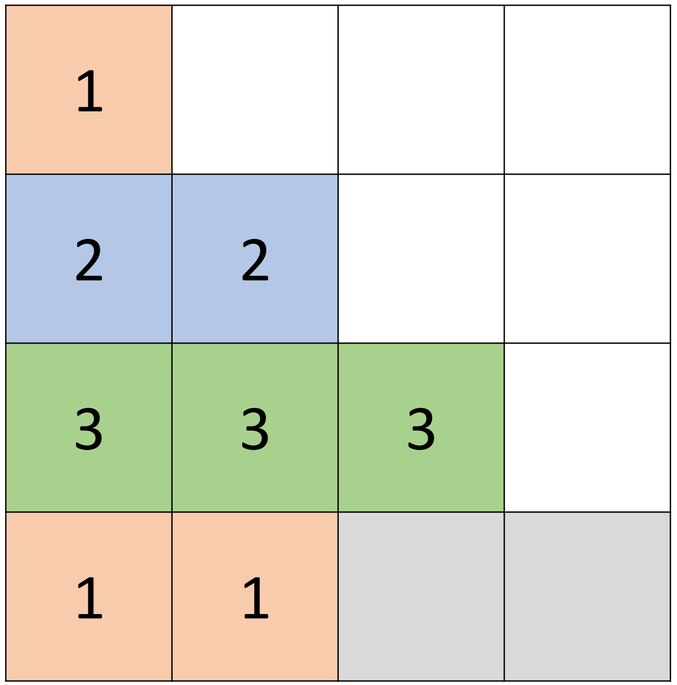


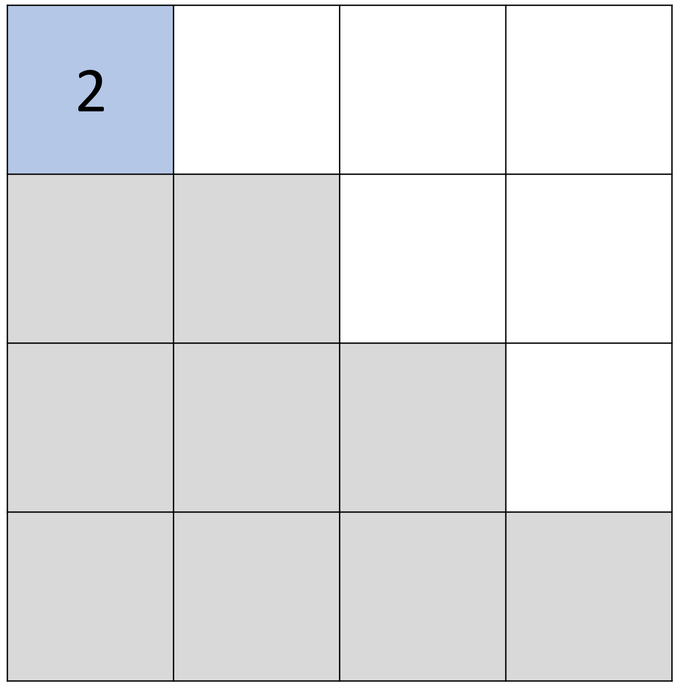
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 3



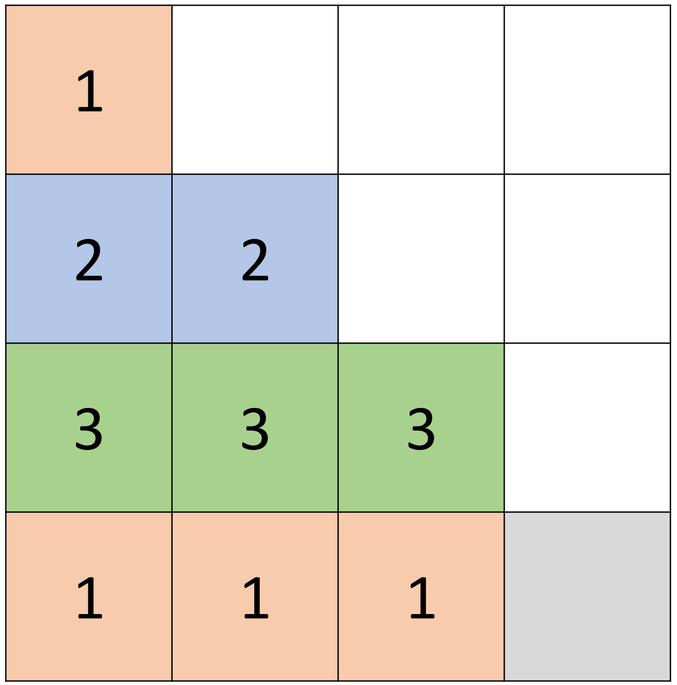


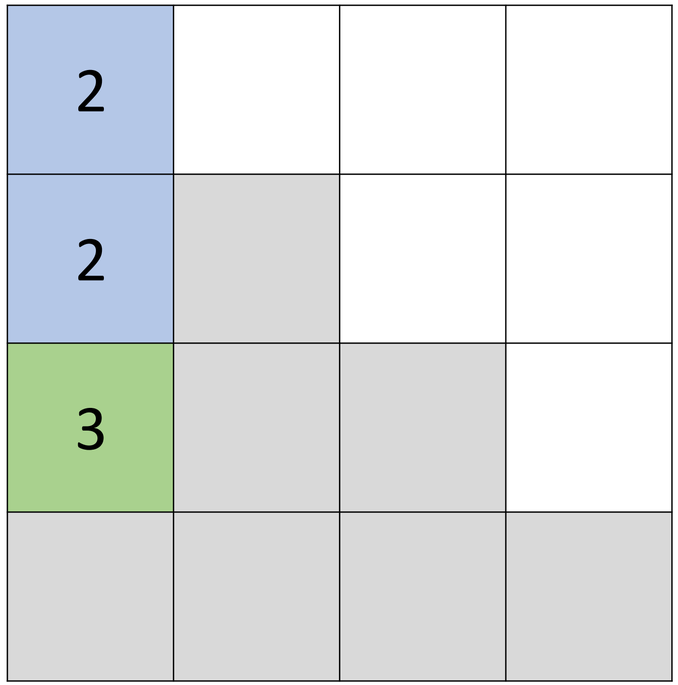
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 4



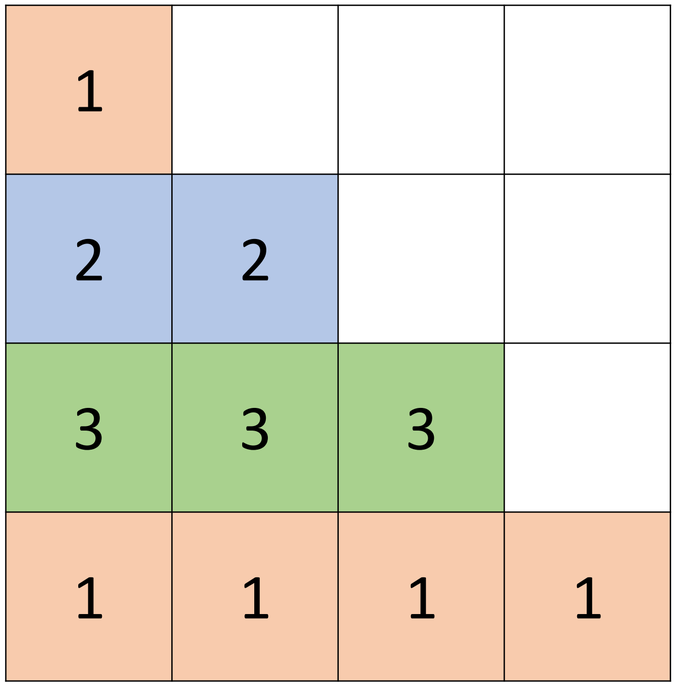


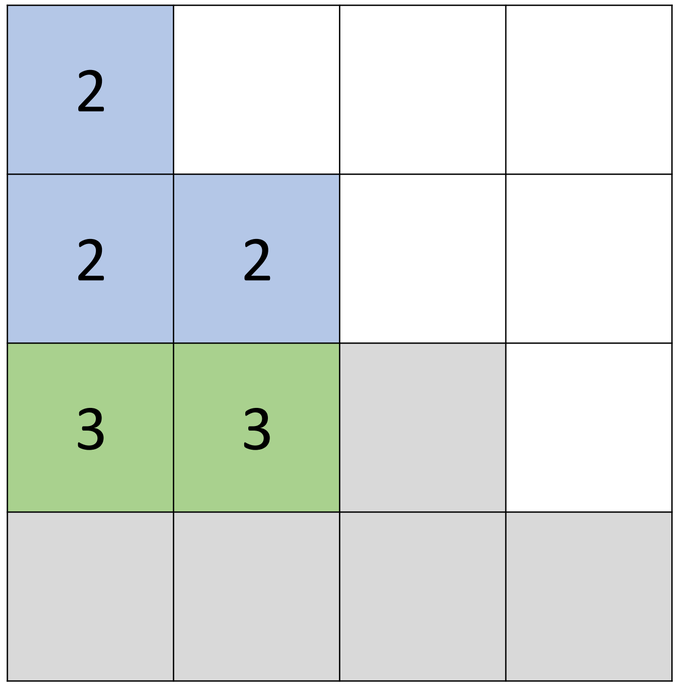
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 5



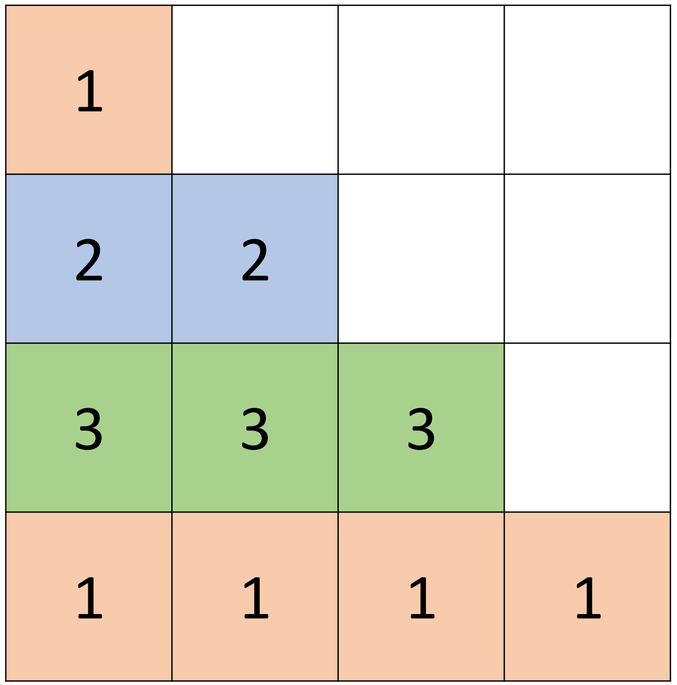


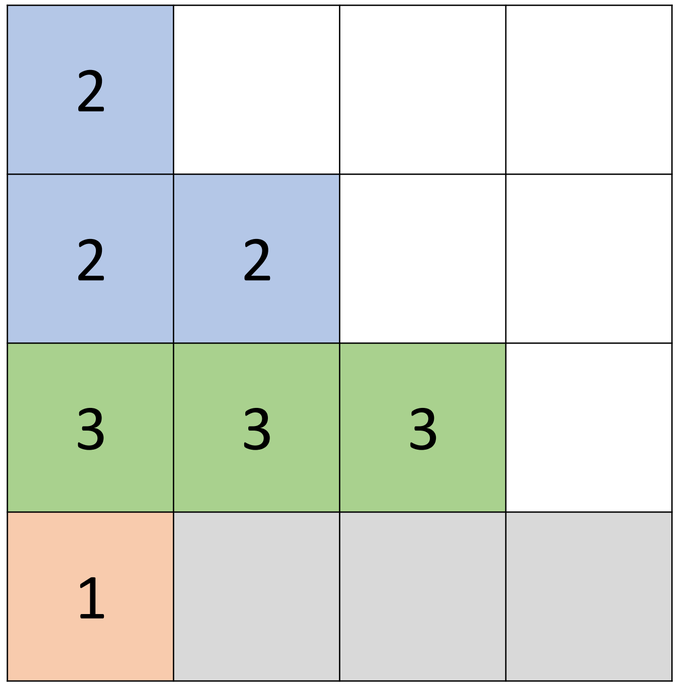
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 6



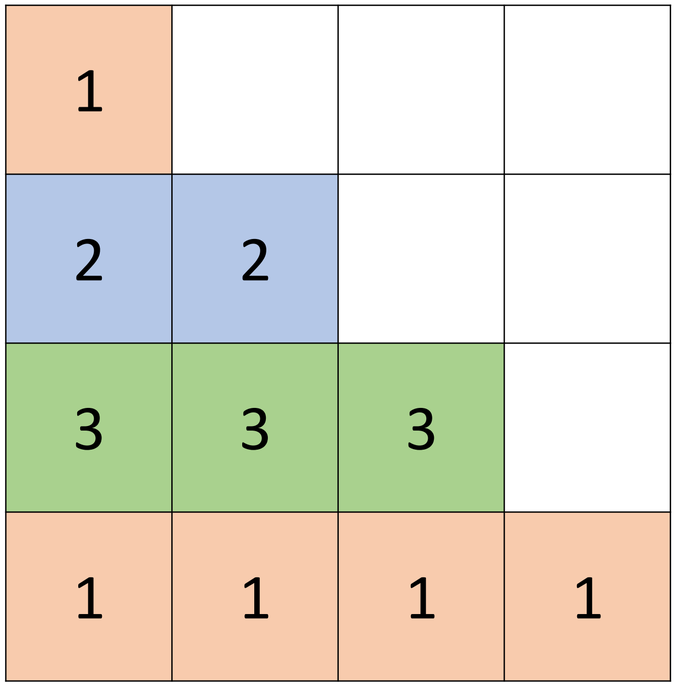


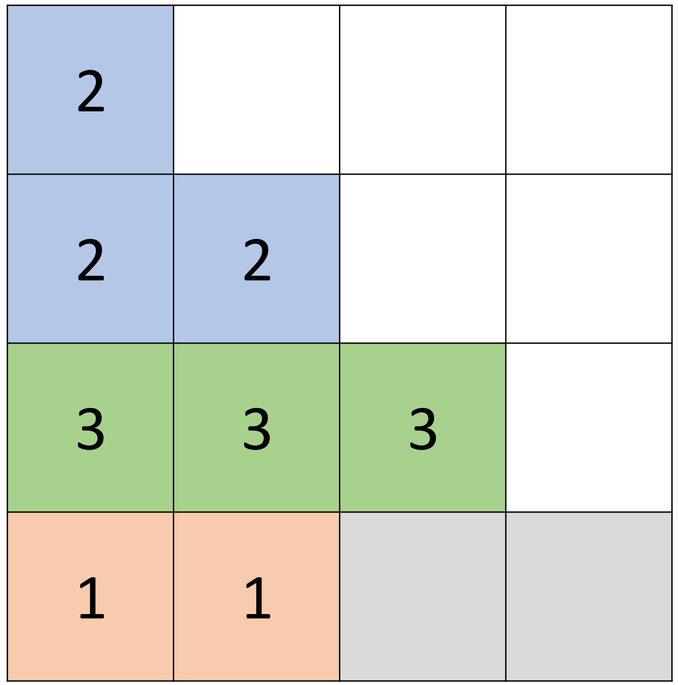
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 7



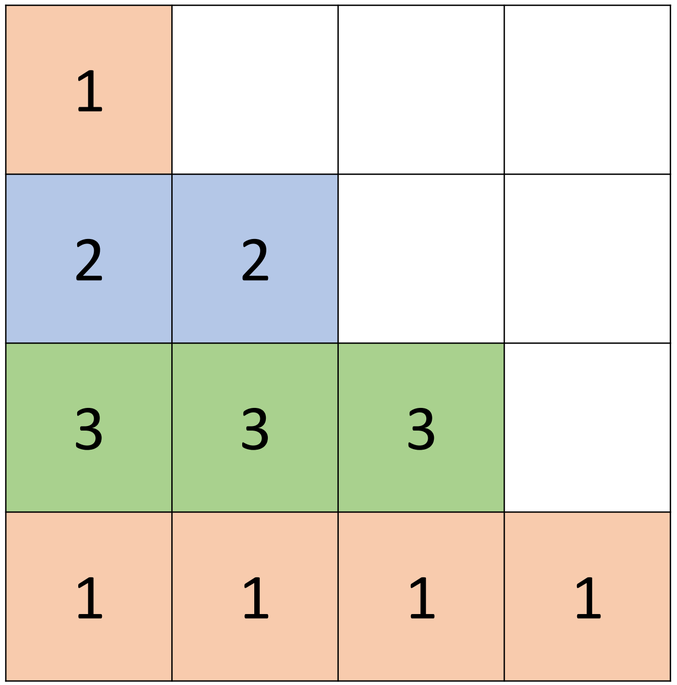


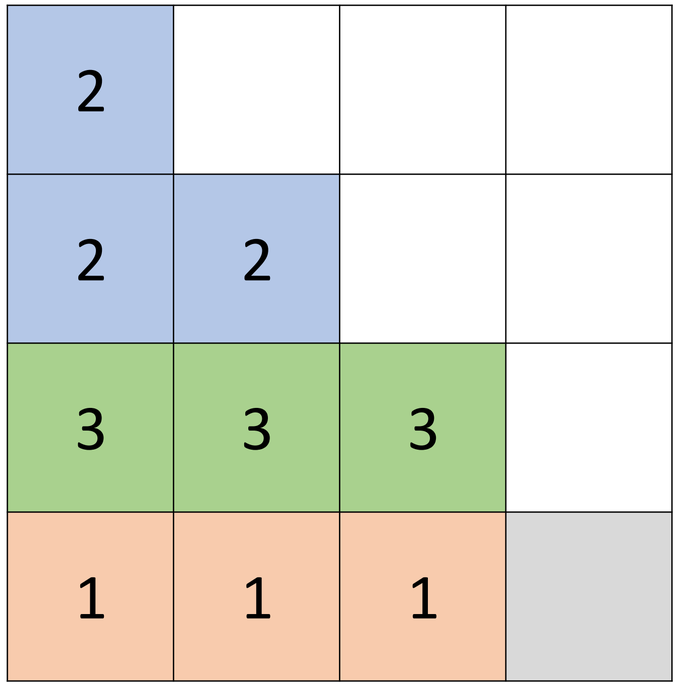
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 8





Load Balancing: Causal Attention

Challenge: Unequal work per tile

Example: 2 batches, 3 workers (SMs)

T = 9

1			
2	2		
3	3	3	
1	1	1	1

2			
2	2		
3	3	3	
1	1	1	1

Work distribution: [9, 5, 6] blocks. Longest tile is always scheduled last!

Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

SIAM J. APPL. MATH.
Vol. 17, No. 2, March 1969

BOUNDS ON MULTIPROCESSING TIMING ANOMALIES*

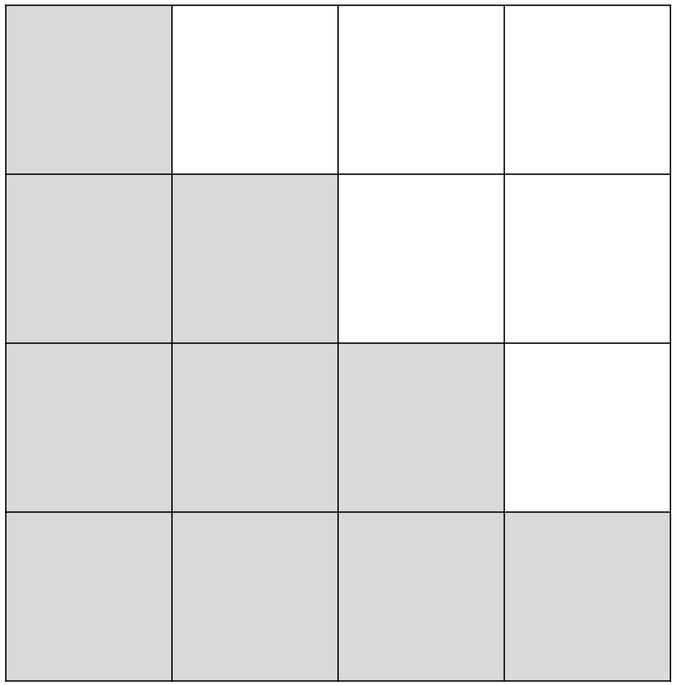
R. L. GRAHAM†

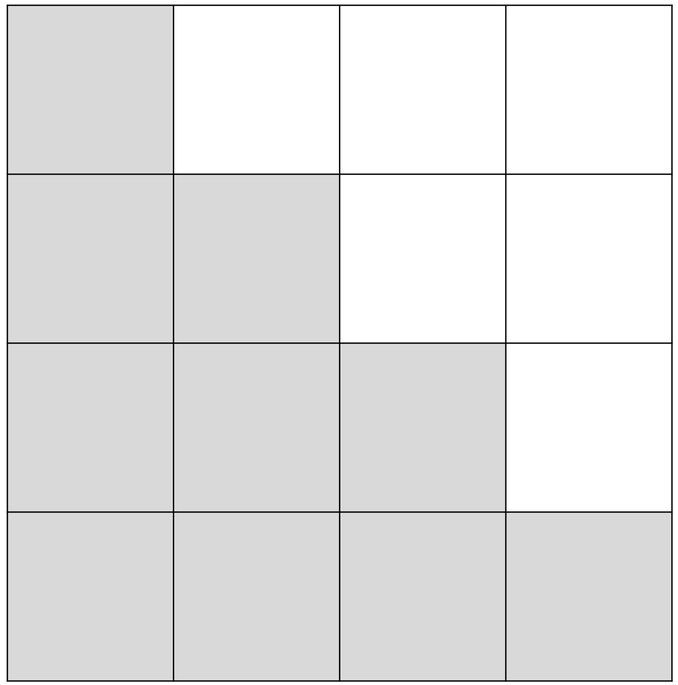
1. Introduction. It is well known (cf. [5], [6], [8]) to workers in the field of parallel computation that a multiprocessing system consisting of many identical processors acting in parallel may exhibit certain somewhat unexpected "anomalies," even though the system operates under a rather natural set of rules; e.g., it can happen that increasing the number of processors can *increase* the length of time required to execute a given set of tasks. In this paper we study a typical model of such a multiprocessing system, and we determine the precise extent by which the execution time for a set of tasks can be influenced because of these timing anomalies. A special case of this model will be shown to generate an interesting number-theoretic question, partial answers to which are given in the latter half of the paper.

Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first



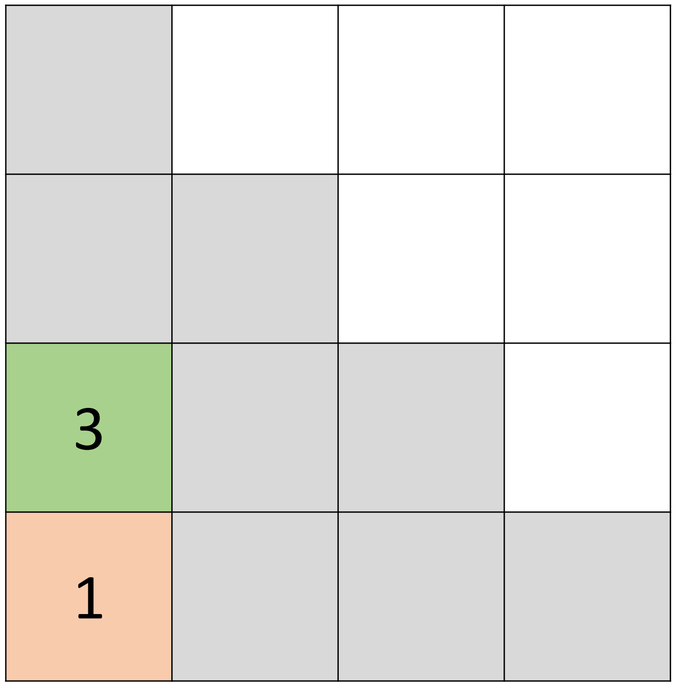


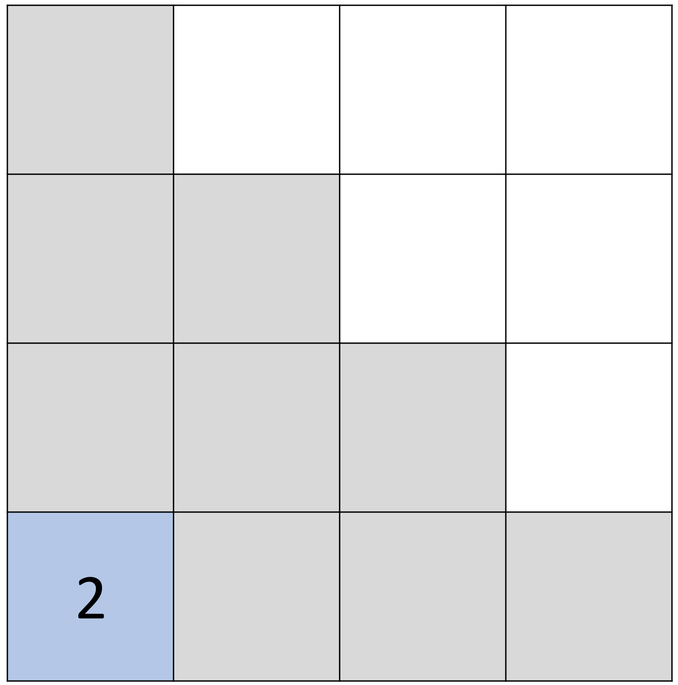
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 1



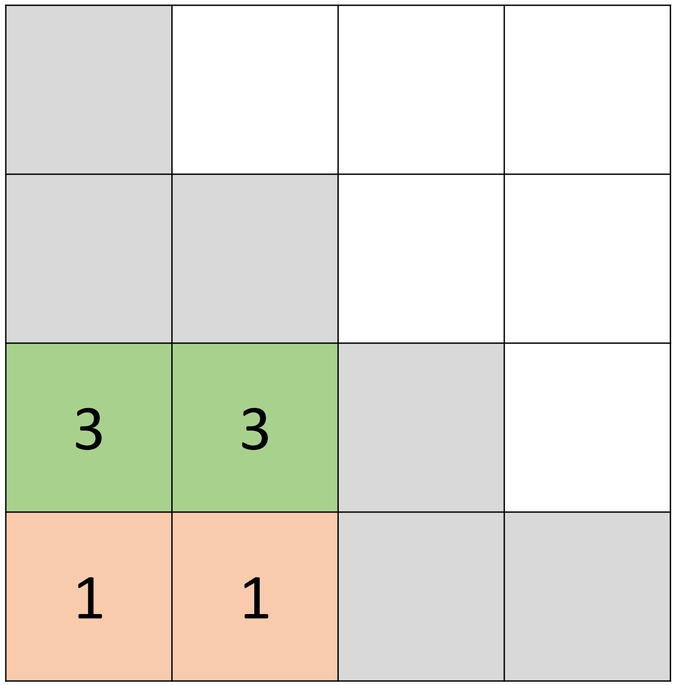


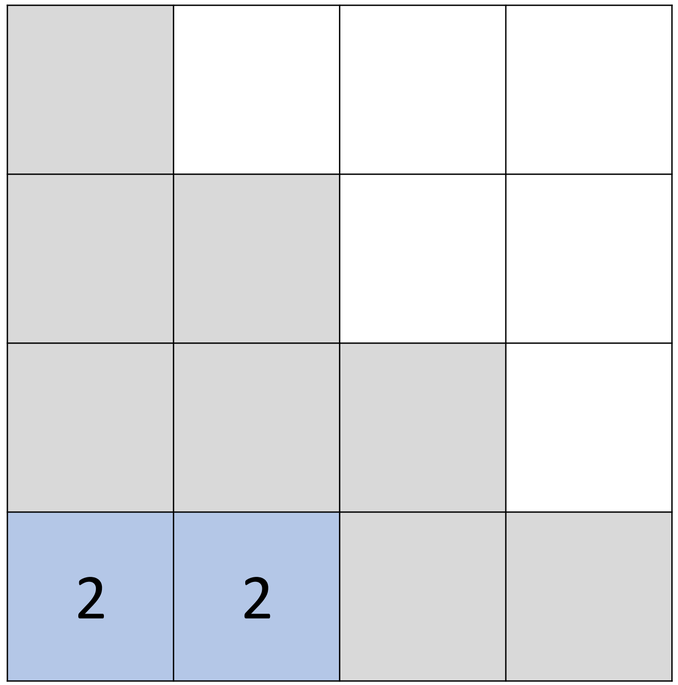
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 2



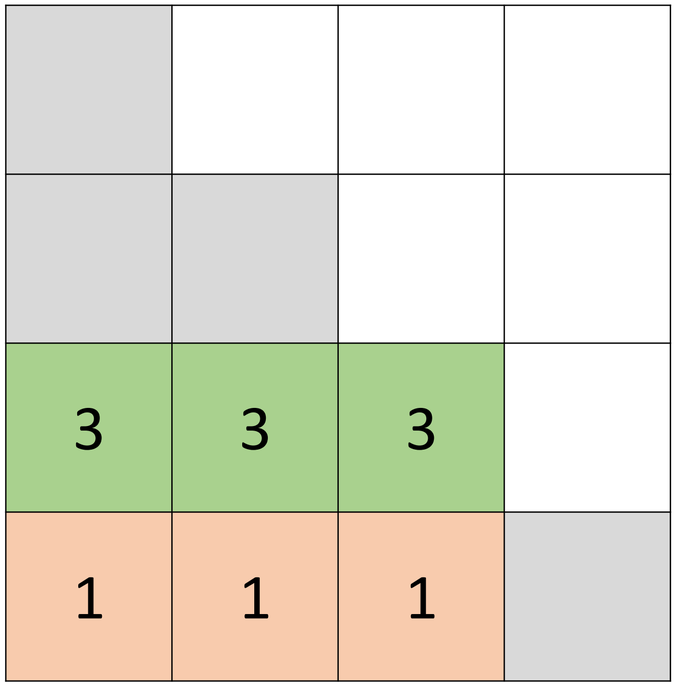


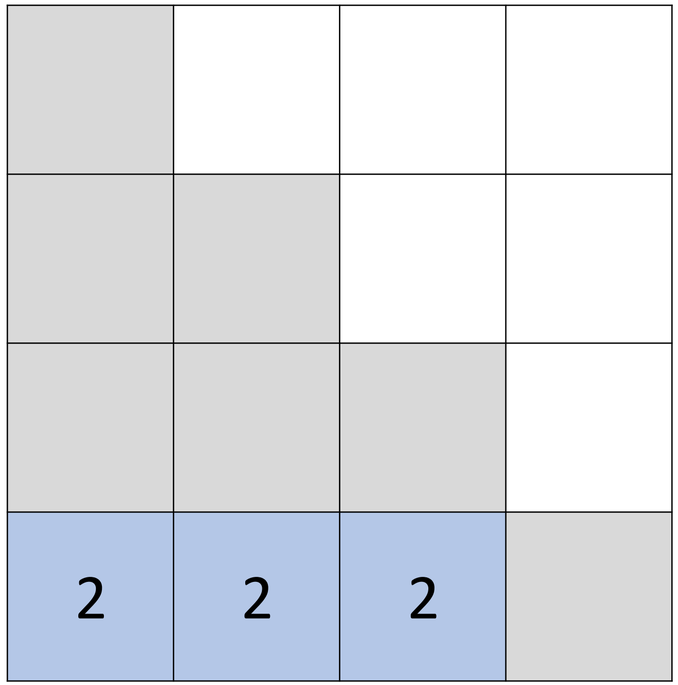
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 3



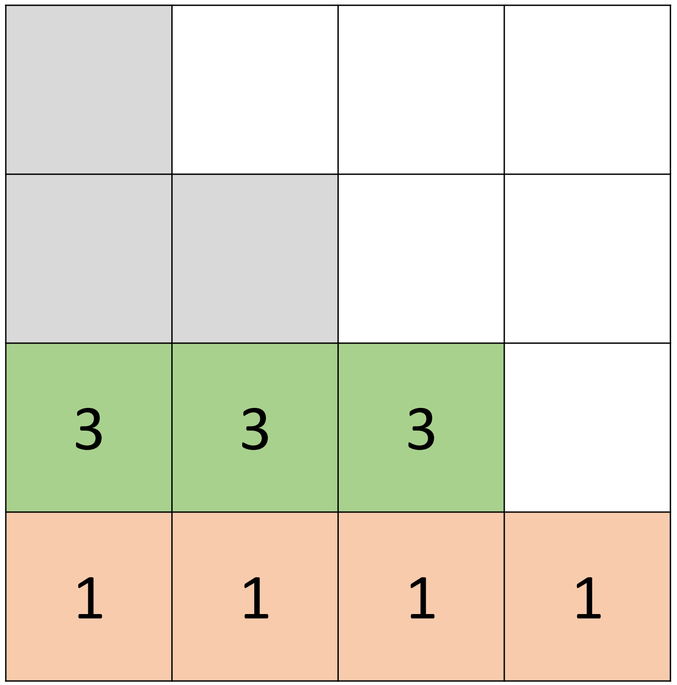


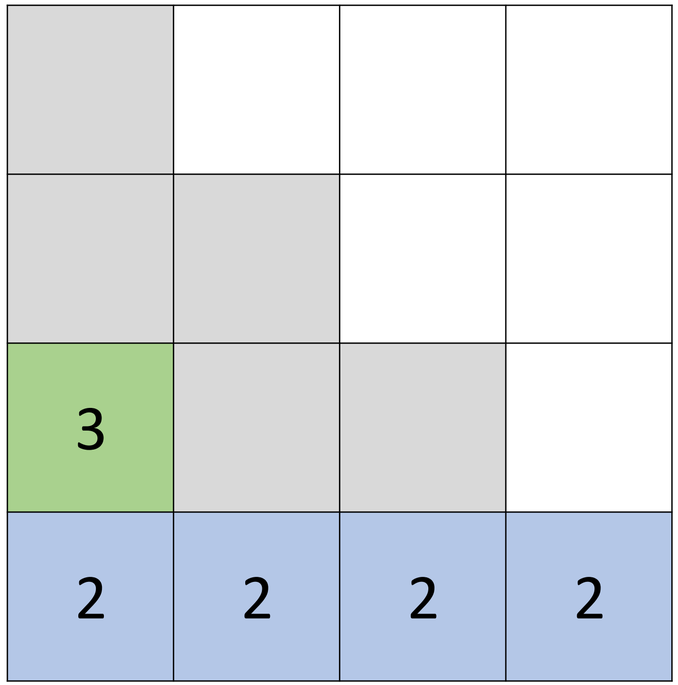
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 4



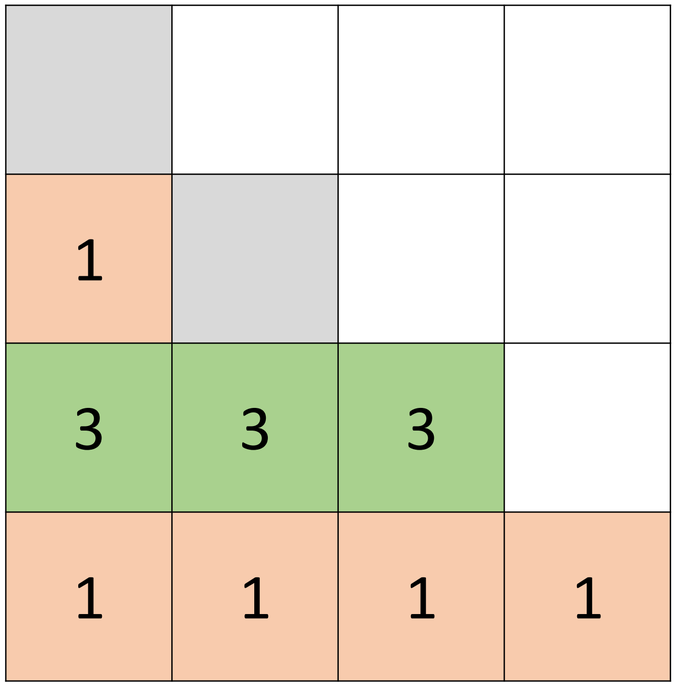


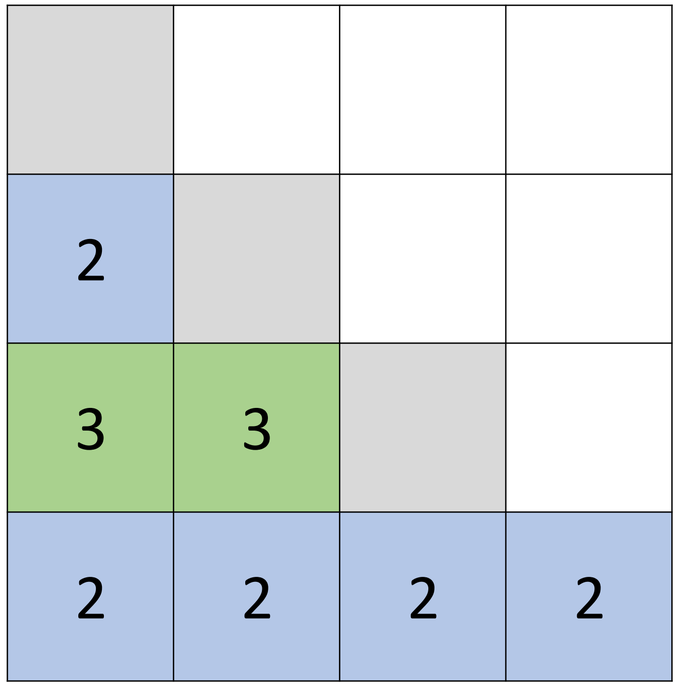
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 5



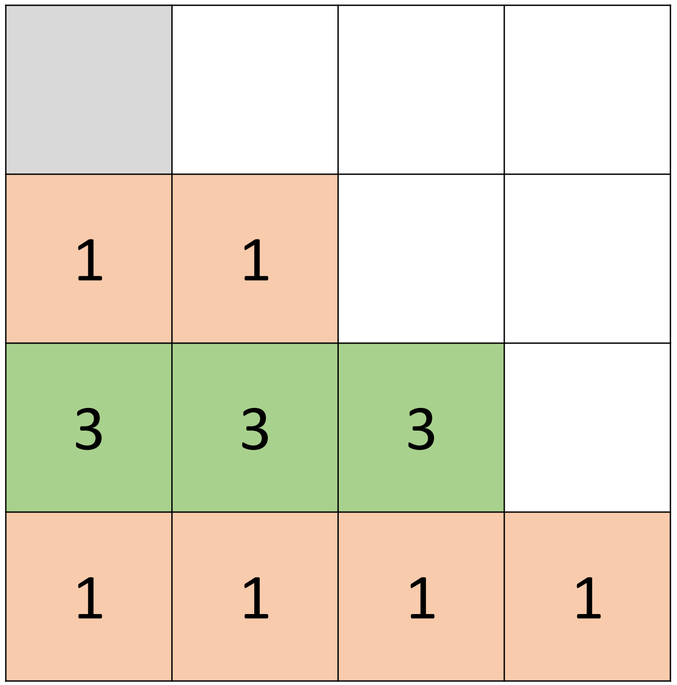


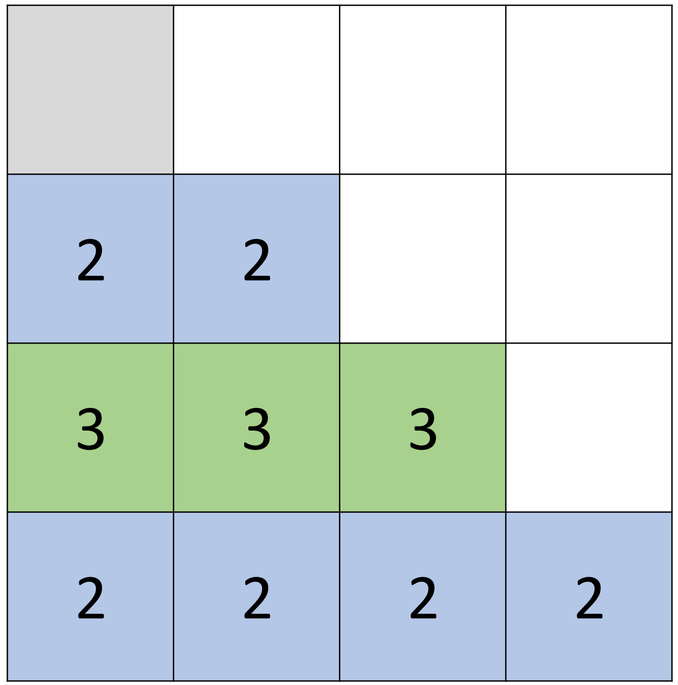
Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 6





Load Balancing: Causal Attention

Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

T = 7

1			
1	1		
3	3	3	
1	1	1	1

2			
2	2		
3	3	3	
2	2	2	2

Work distribution: [7, 7, 6] vs [9, 5, 6] previously

Load Balancing: Causal Attention

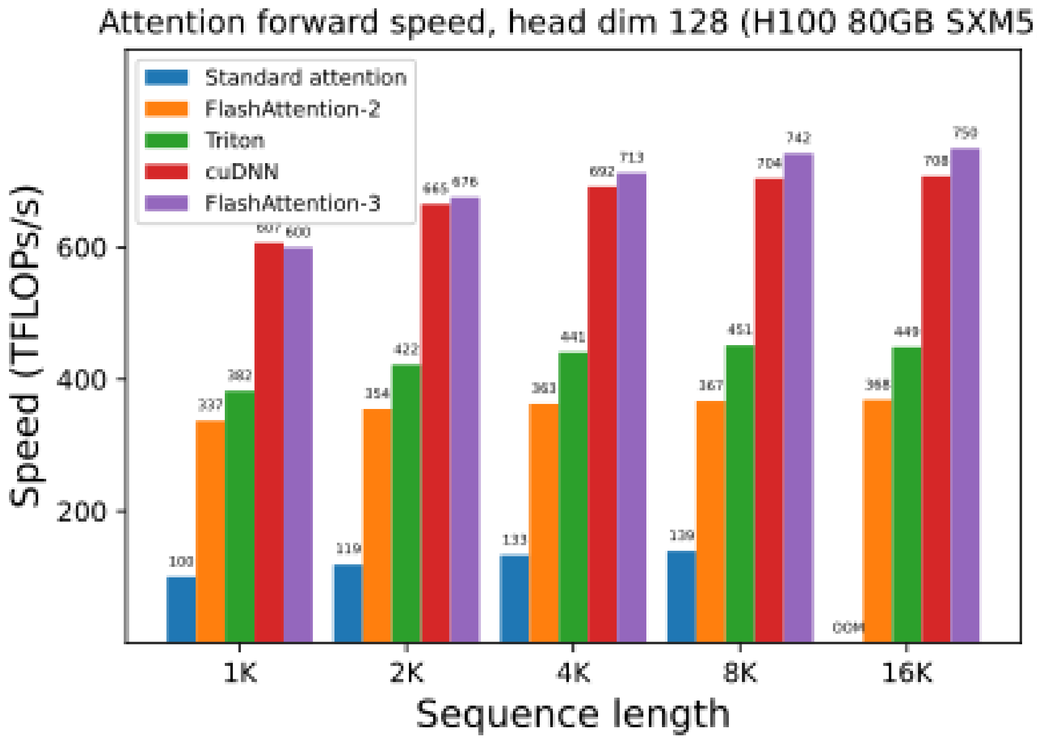
Challenge: Unequal work per tile

Idea: Assign work to workers using longest-processing-time-first

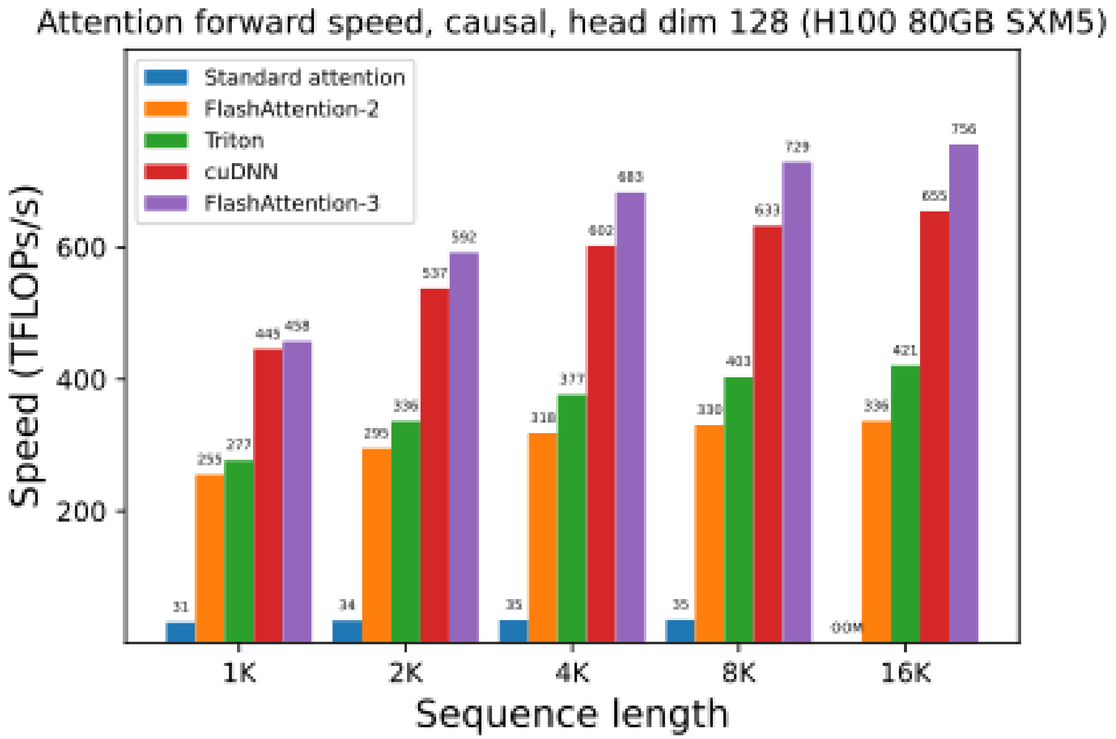
Caveat: Take care to not spill L2 cache (using L2 swizzling)

Causal attention speed 670 TFLOPS -> 730 TFLOPS

BF16 Benchmark: 1.8-2.2x speedup



Without causal mask

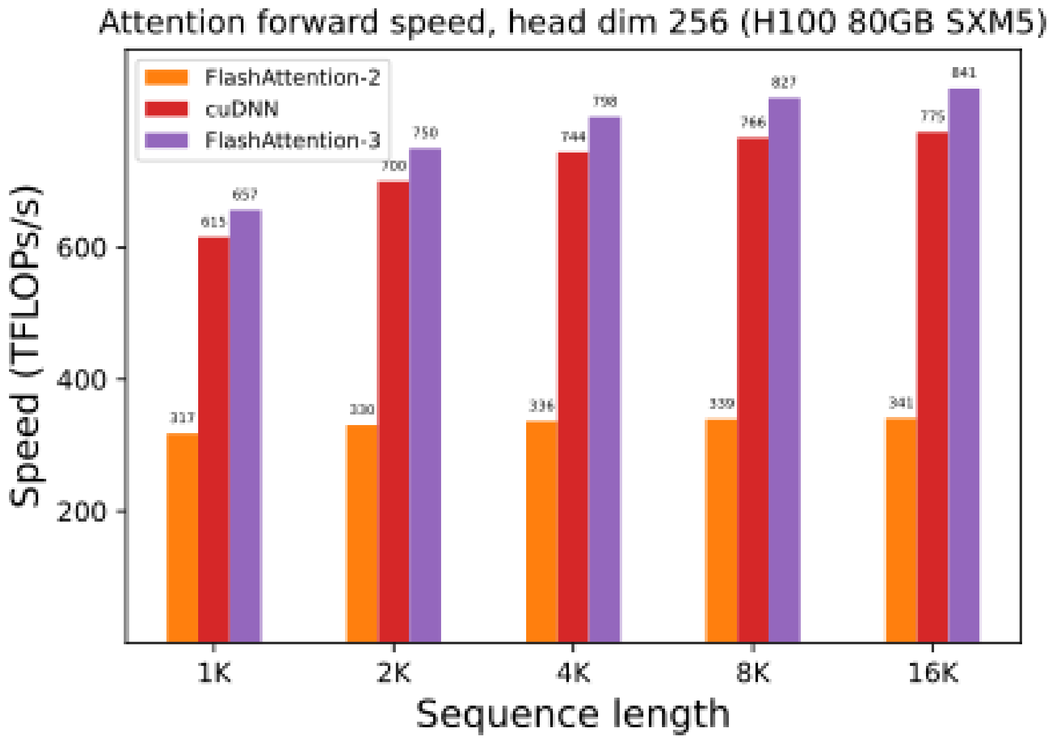


With causal mask

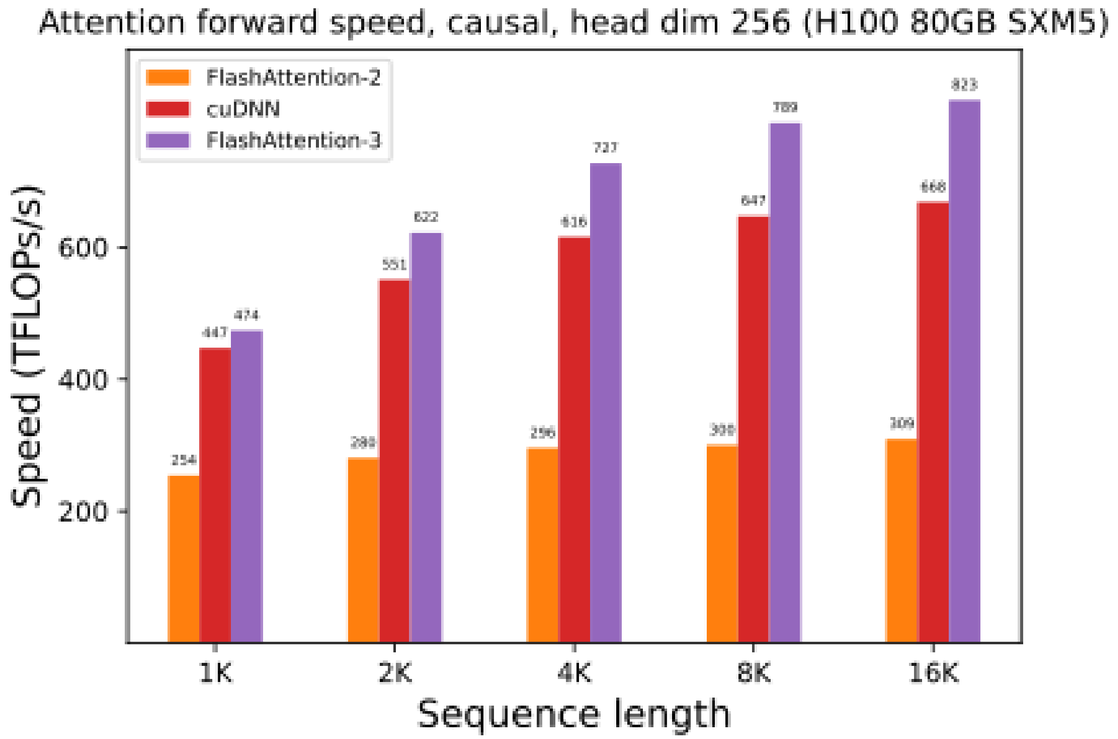
CUDA tool kit 12.8
Triton 3.1
cuDNN 9.6

Causal Attention 730-750 TFLOPS ≈ Matmul speed!

BF16 Benchmark: Reach up to 840 TFLOPS!



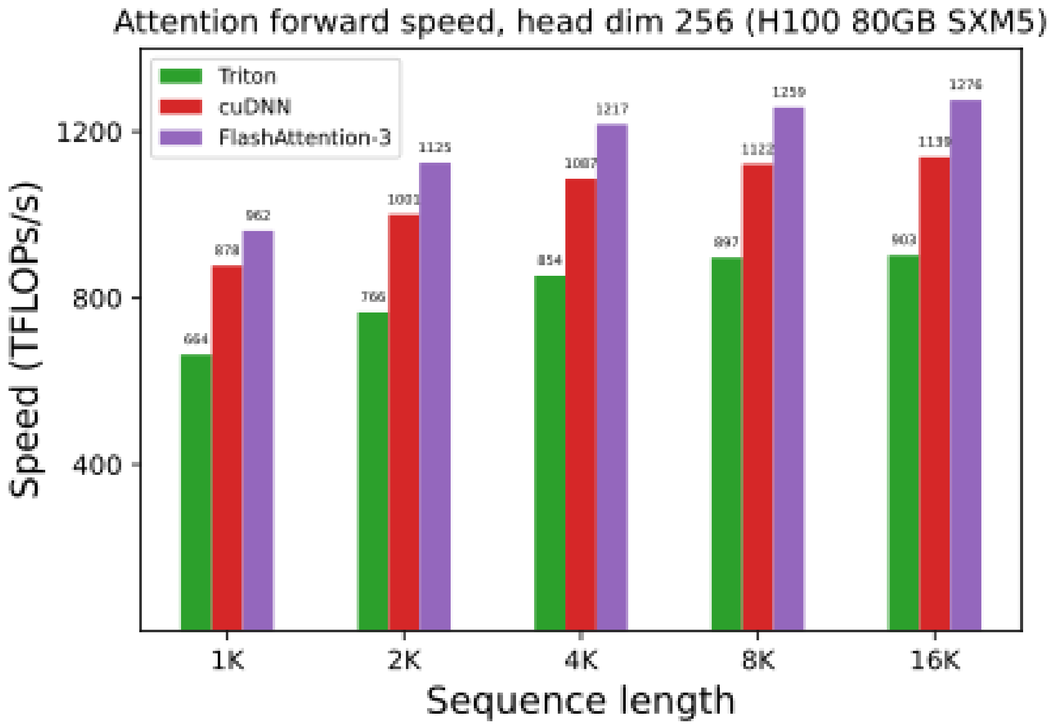
Without causal mask



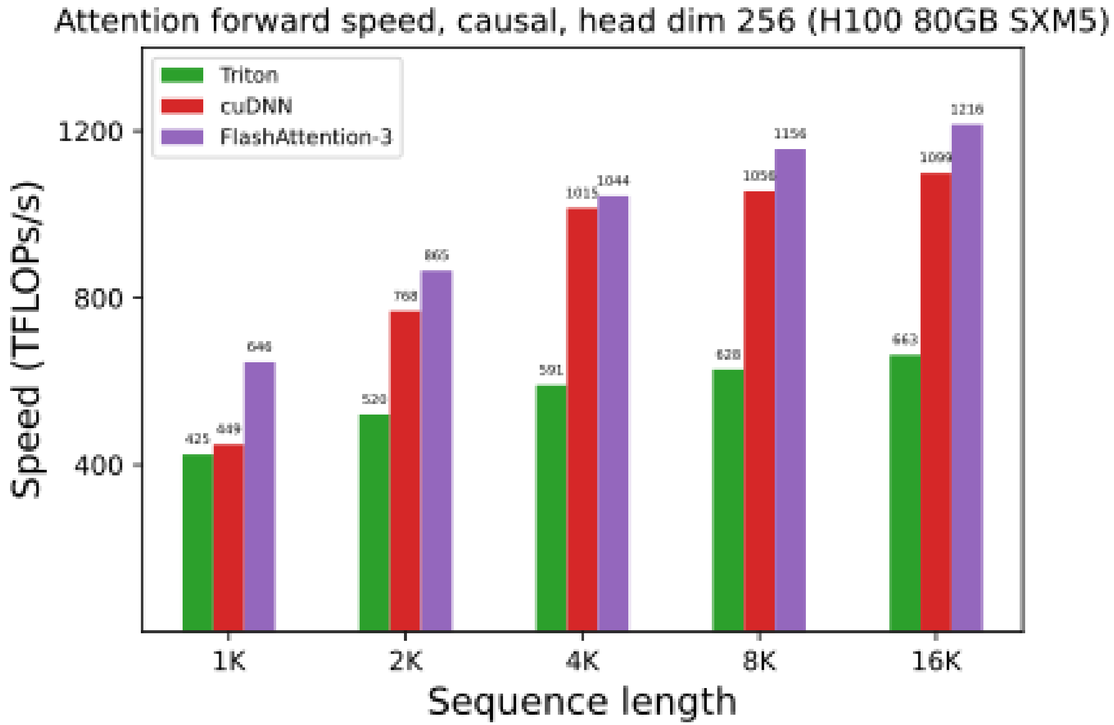
With causal mask

CUDA tool kit 12.8
Triton 3.1
cuDNN 9.6

FP8 Benchmark: Up to 1.3 PFLOPS!



Without causal mask



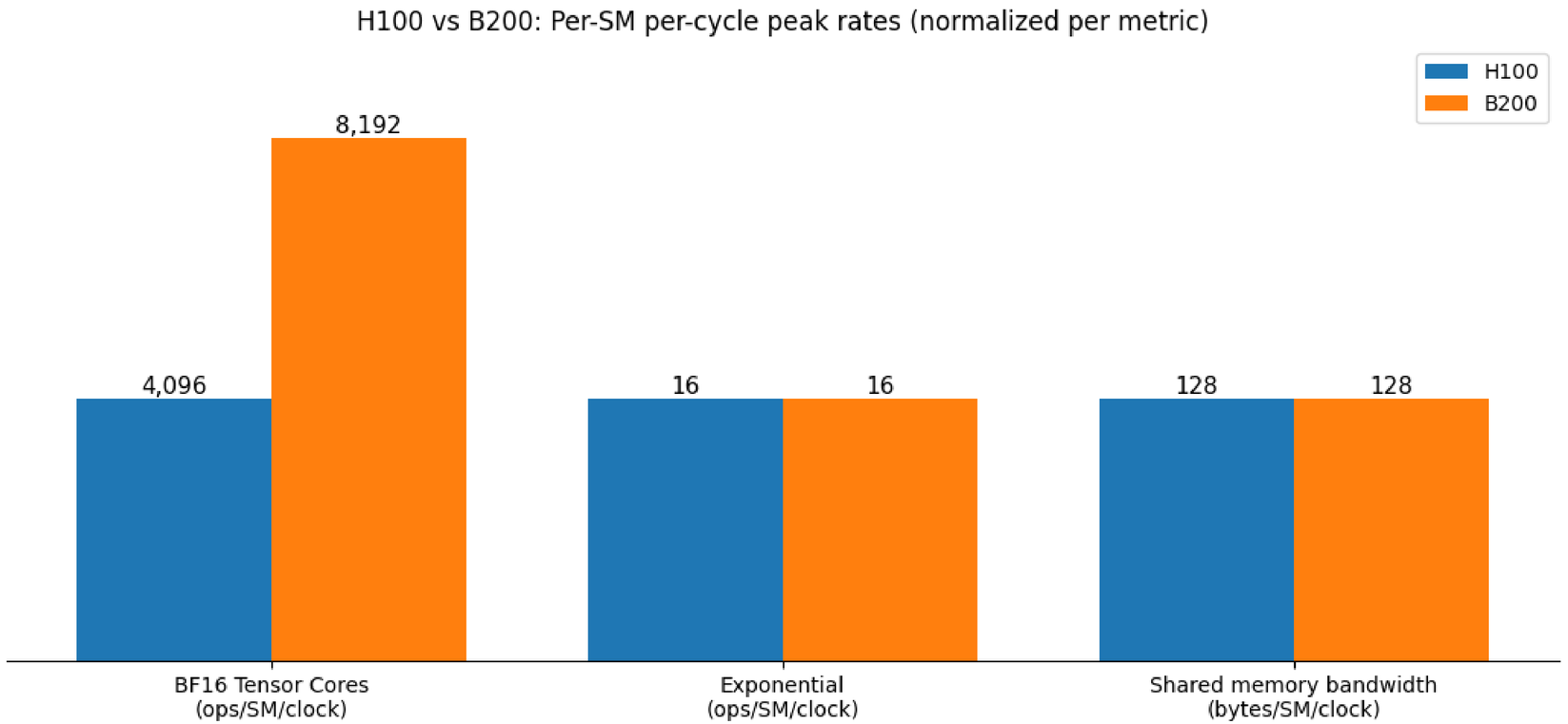
With causal mask

CUDA tool kit 12.8
Triton 3.1
cuDNN 9.6

FlashAttention 4: Algo-Kernel Co-design for Asymmetric Hardware Scaling

Aka The Rise of Asymmetric Scaling in Blackwell GPUs

Blackwell matmul units get faster, but exponential units and shared memory stay the same speed

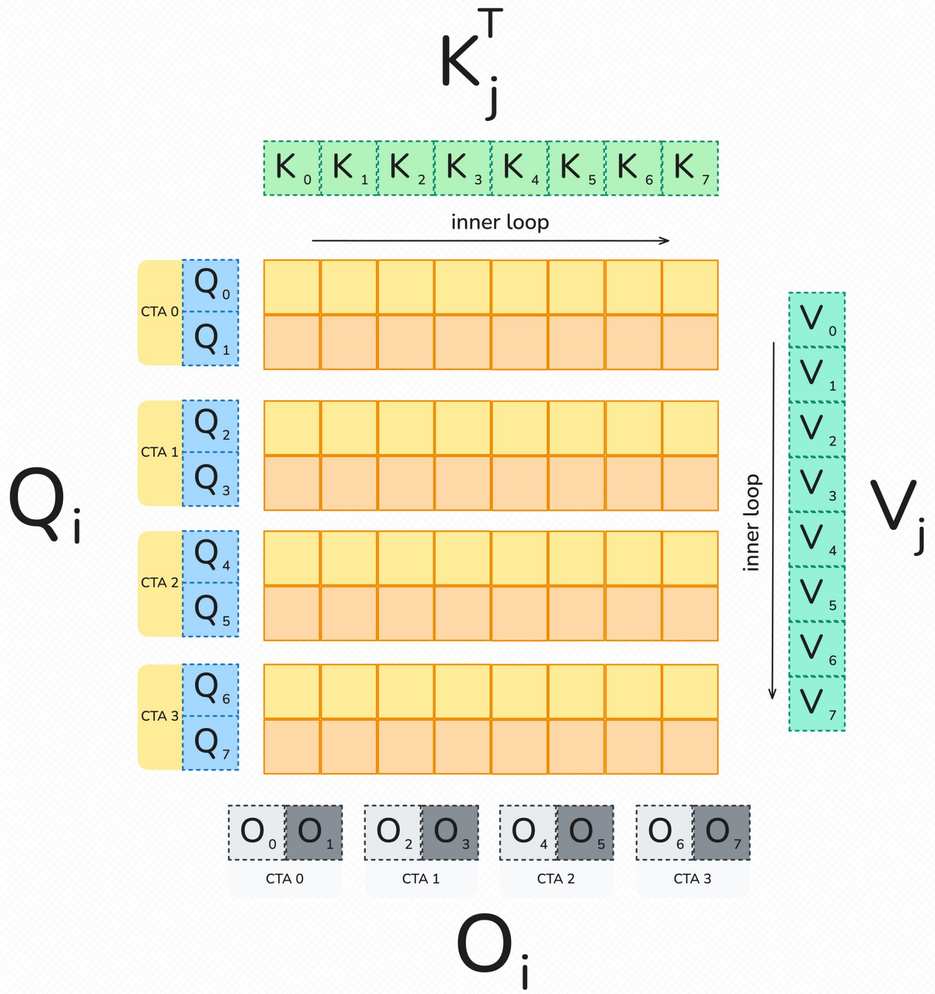


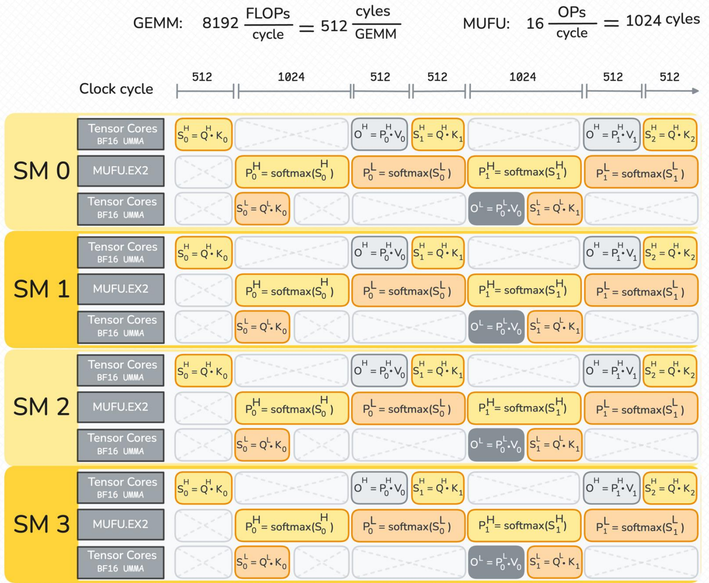
Attn fwd bottlenecked by **exponential**, bwd bottlenecked by **smem** traffic!

Ted Zadouri*, Markus Hoehnerbach*, Jay Shah*, Timmy Liu, Vijay Thakkar, Tri Dao.
FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. MLSys 2026.

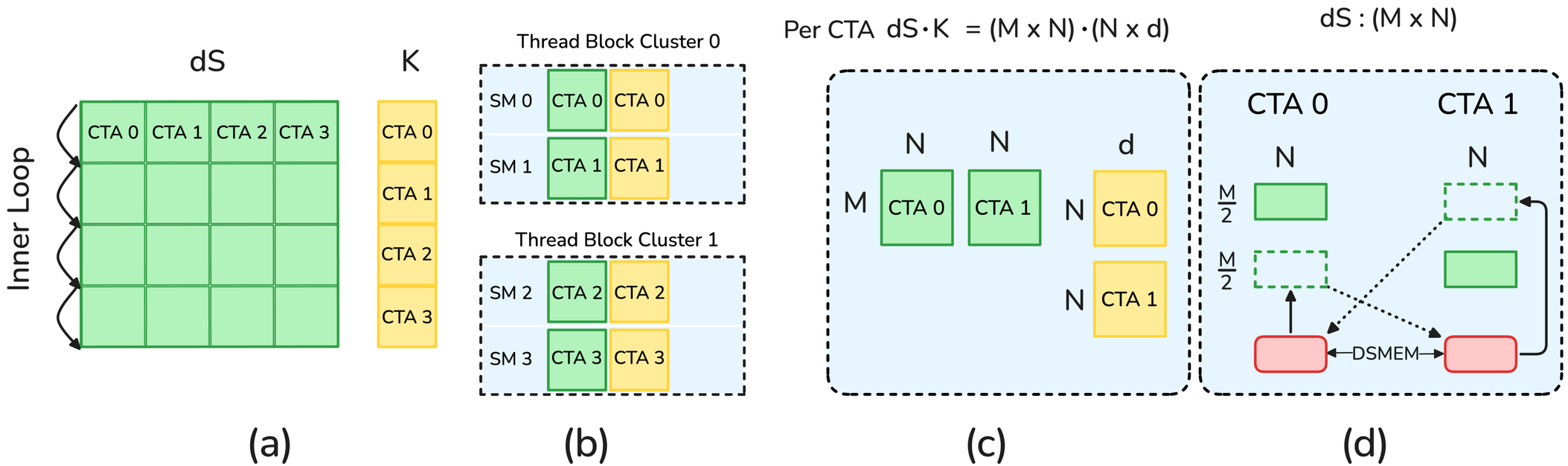
Forward Pass: Redesigned Algorithm to Go Beyond Hardware Limits

Algo changes:
1. Pingpong pipeline of query tiles
2. Exp emulation with polynomials (Chebyshev!)
3. New variant of online softmax to skip 90% of rescaling



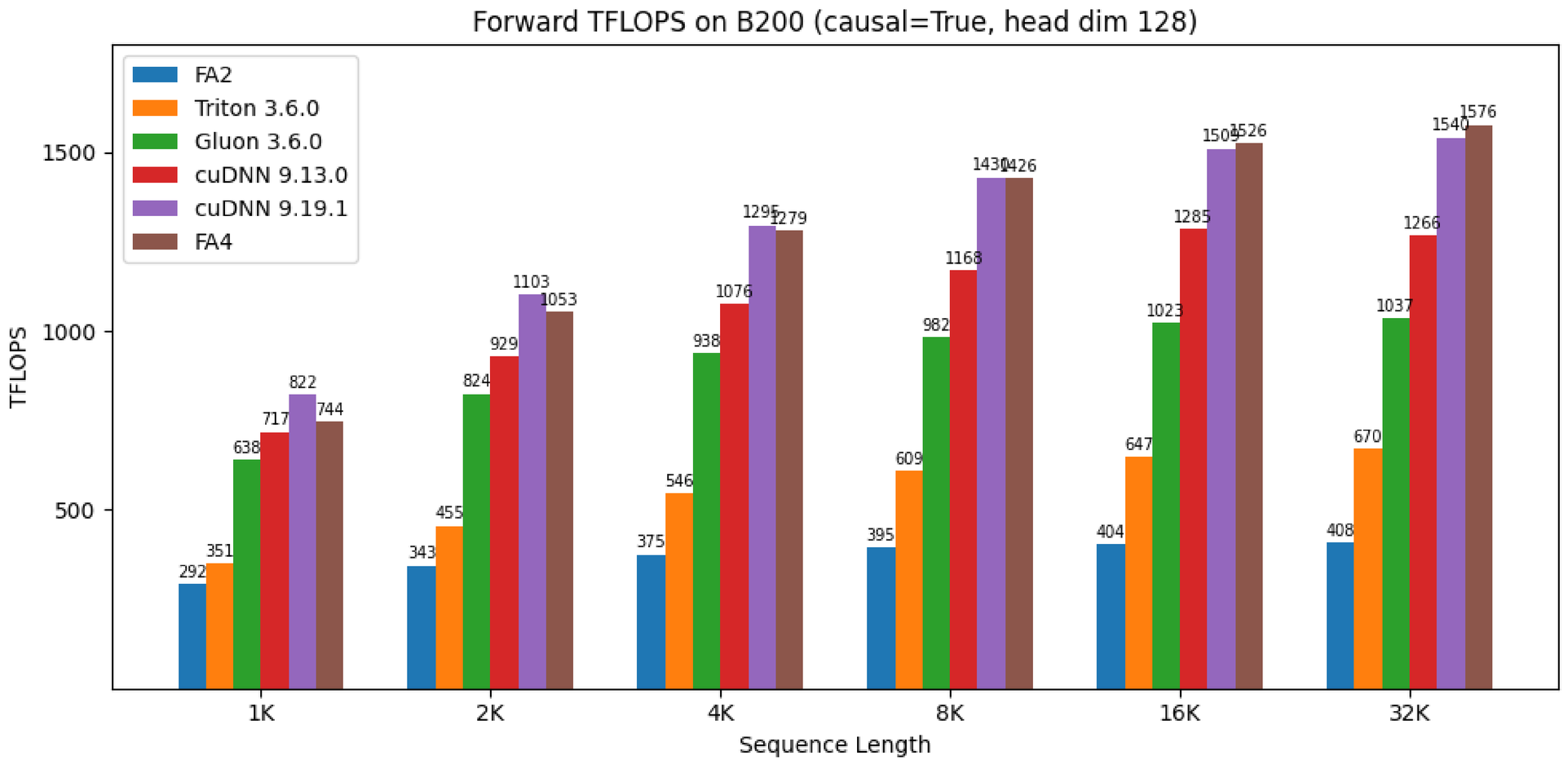


Backward Pass: New 2CTA MMA Instructions to Reduce Smem Bandwidth



Careful synchronization and data movement for 2 CTAs to cooperate

FlashAttention 4 Perf: Forward Pass

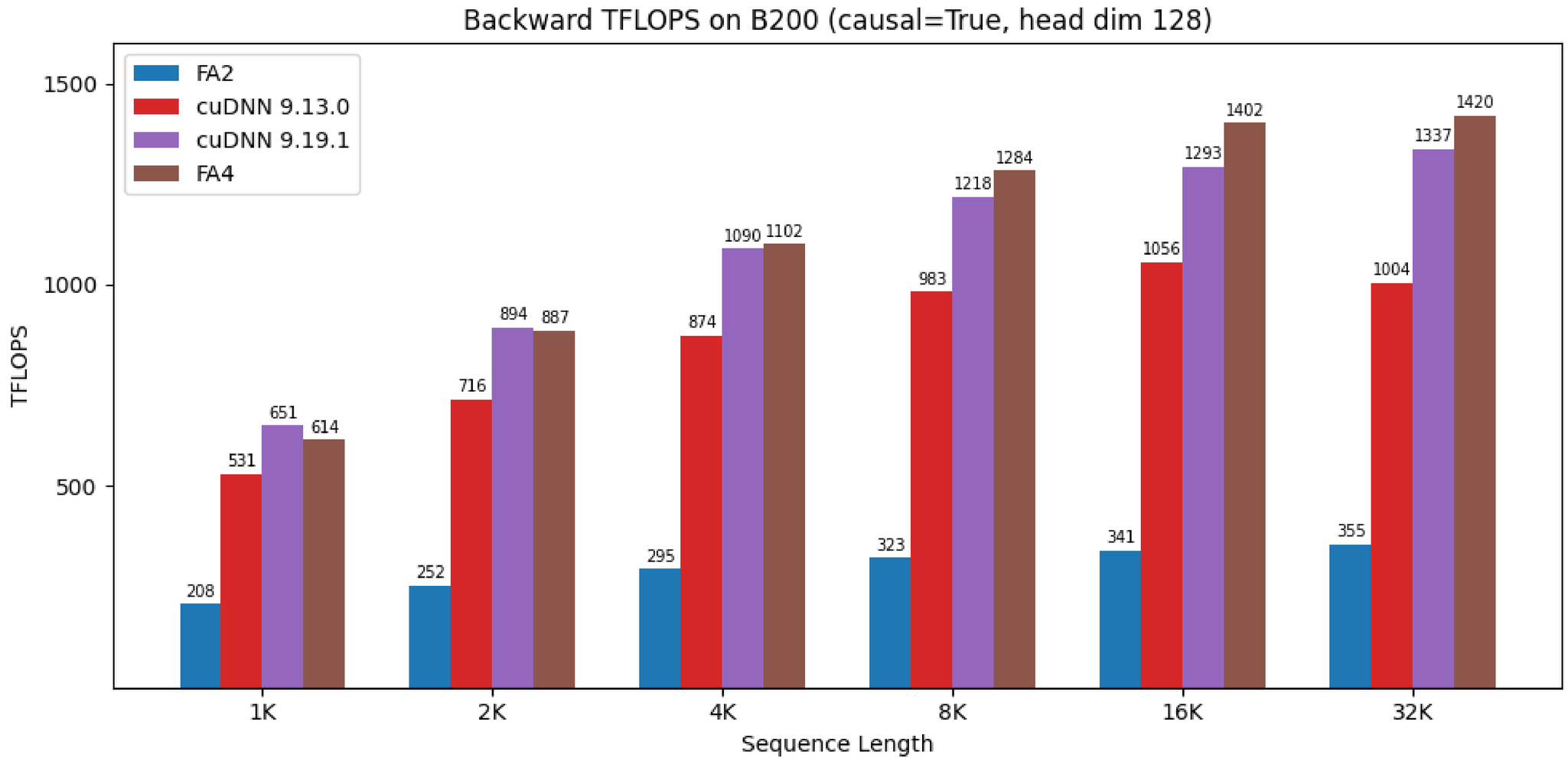


Collab

Newer cuDNN versions now incorporate many FA4 optimizations

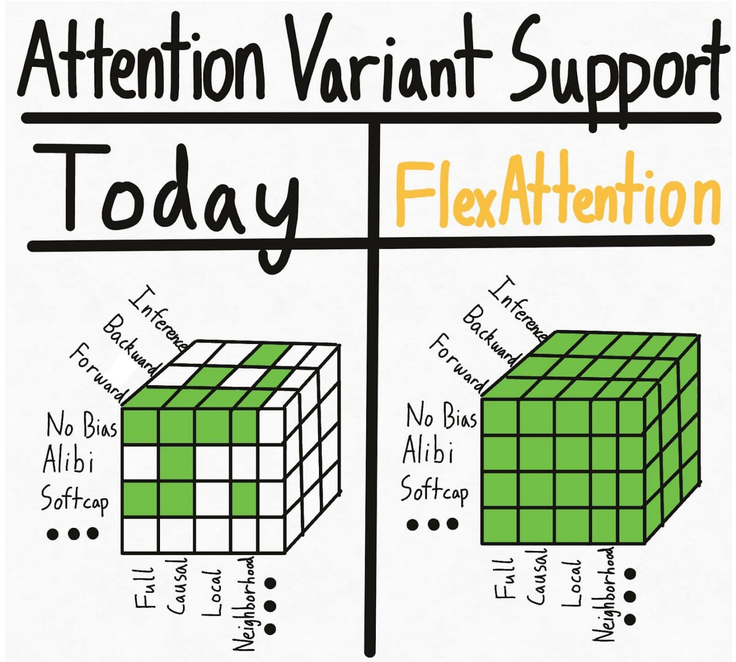
Attn as optimized as matmul (up to 1600 TFLOPS), despite new bottlenecks, thanks to insights from algos + hardware understanding!

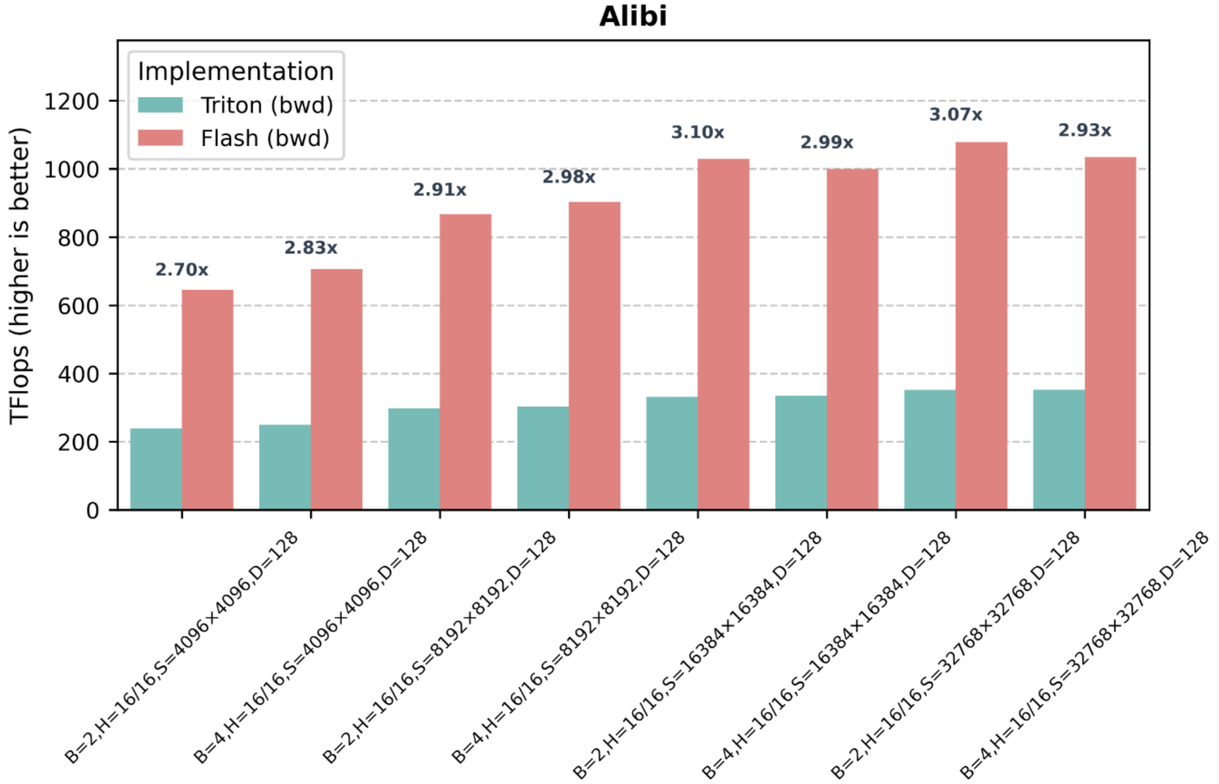
FlashAttention 4 Perf: Backward Pass



4x speedup compared to FA2 baseline with no Blackwell optimizations

Written in Cute-DSL, huge FlexAttention speedup



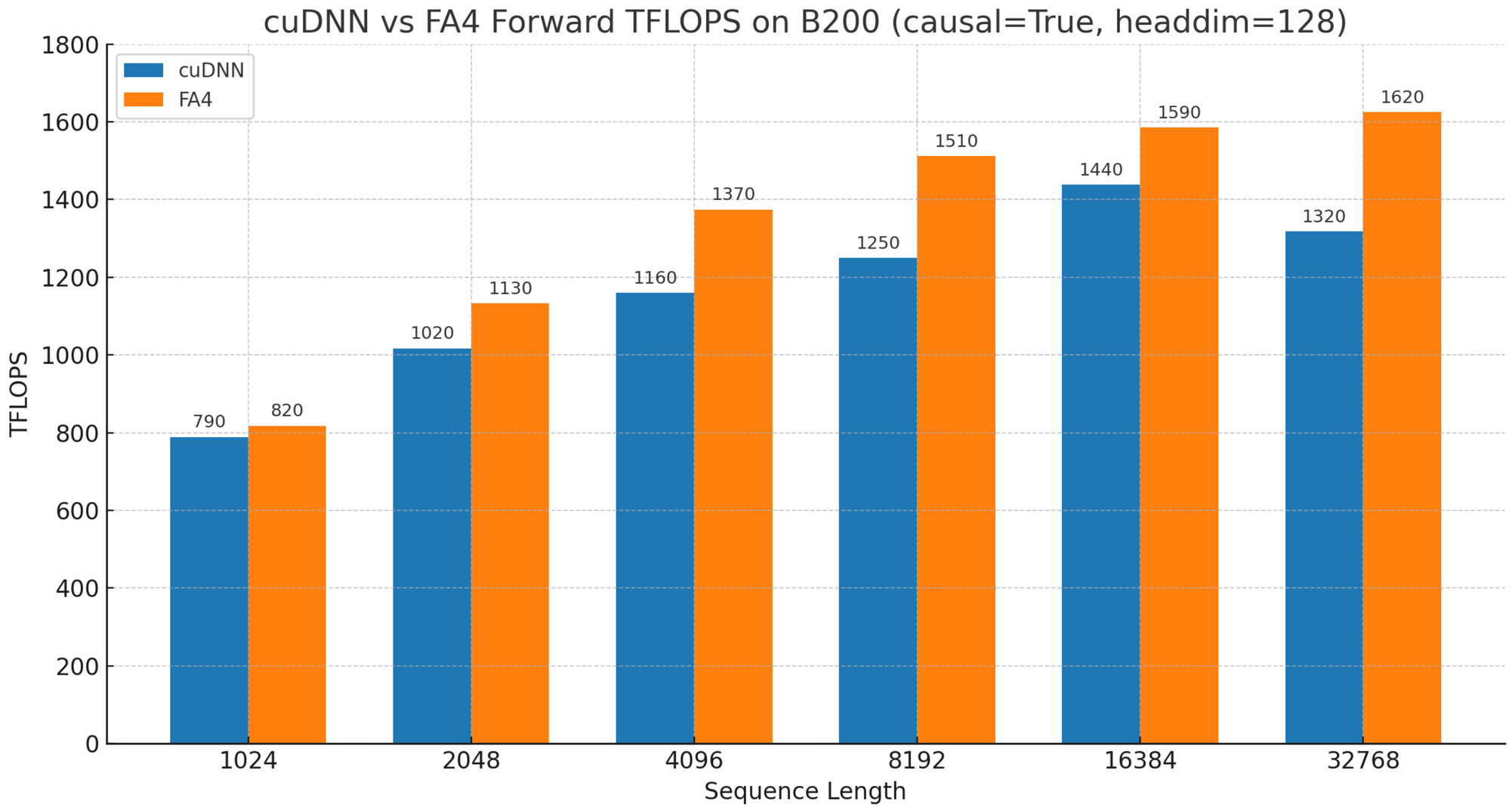


Abstractions from FA4 (block sparse, masking) + Python-embedded DSL

-> **2.7-3.0x** FlexAttn speedup across the board

Driss Guessous, Markus Hoehnerbach, Ted Zadouri, Jay Shah, Tri Dao, and others. Bringing muscle to FlexAttention. 2026.

FlashAttention 4 on Blackwell



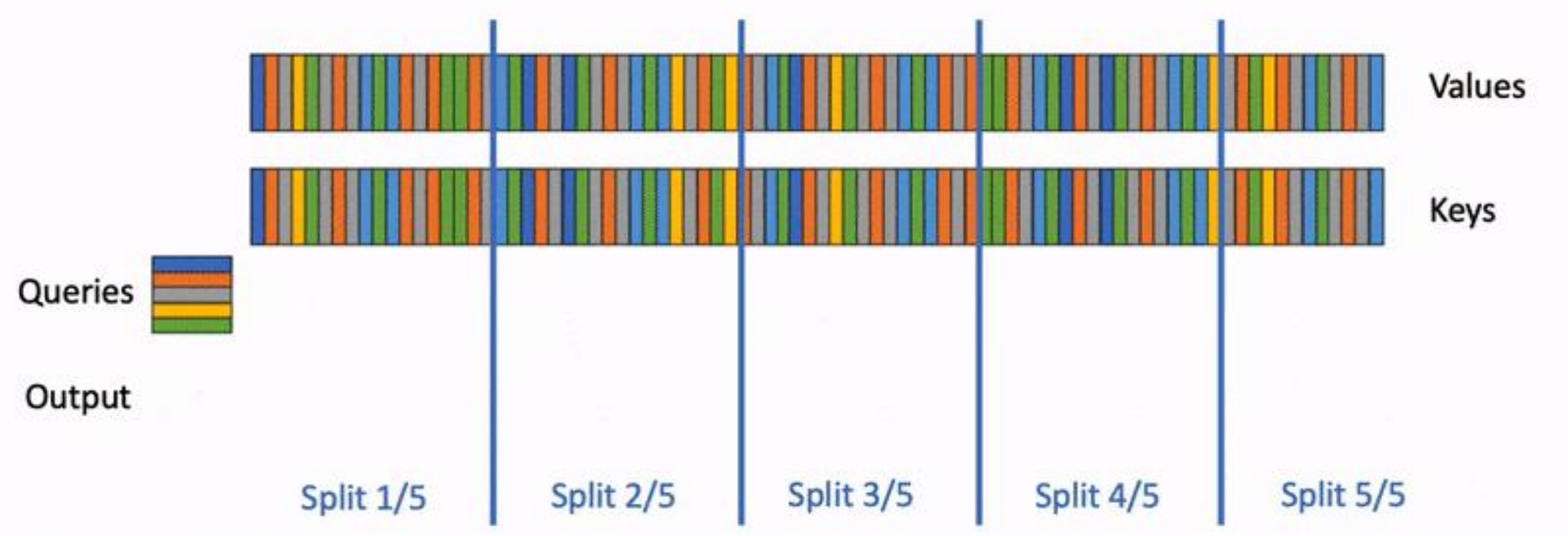
cuDNN 9.11.0
CTK 13.0

https://github.com/Dao-AILab/flash-attention/blob/main/flash_attn/cute/flash_fwd_sm100.py

How to optimize attention for inference?

Optimizations for Decoding Inference: Old and New

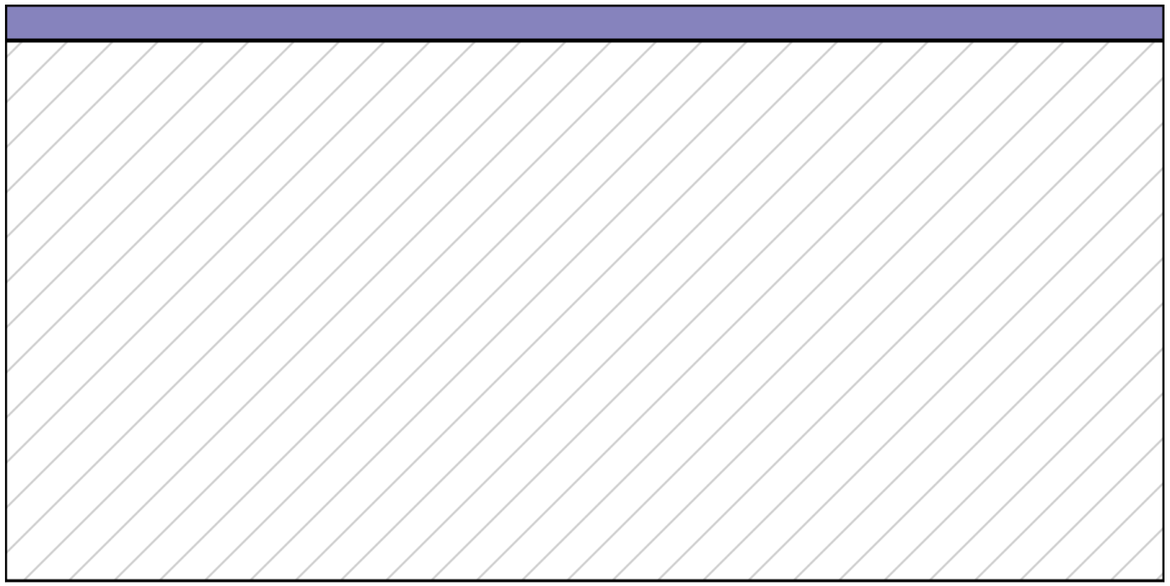
For decoding, query length is short (on the order of a few tokens), while context length is long (for example, 128k).



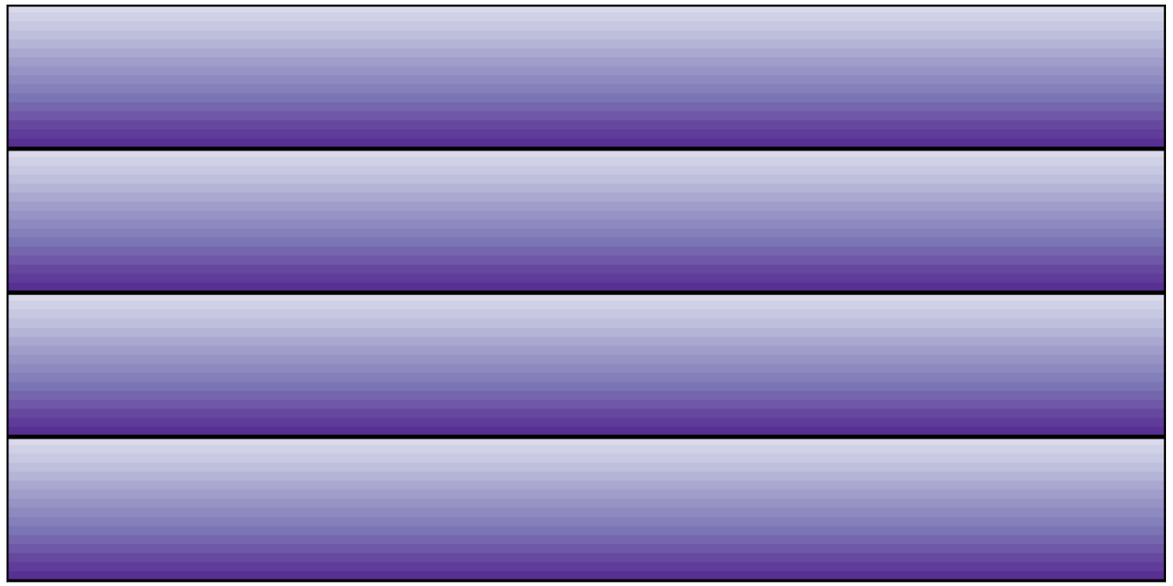
From FA-2, have **Flash Decoding**: split along the KV sequence length to occupy the GPU with enough work.

GQA Packing: compute for multiple query heads per KV head

WGMMA tile is 64 wide in the M dimension. This is wasted for short query length!
However, we can pack multiple query heads to fill out WGMMA tile for MQA/GQA.



Unpacked 64x128 Query Tile for a single head (4 query tokens)

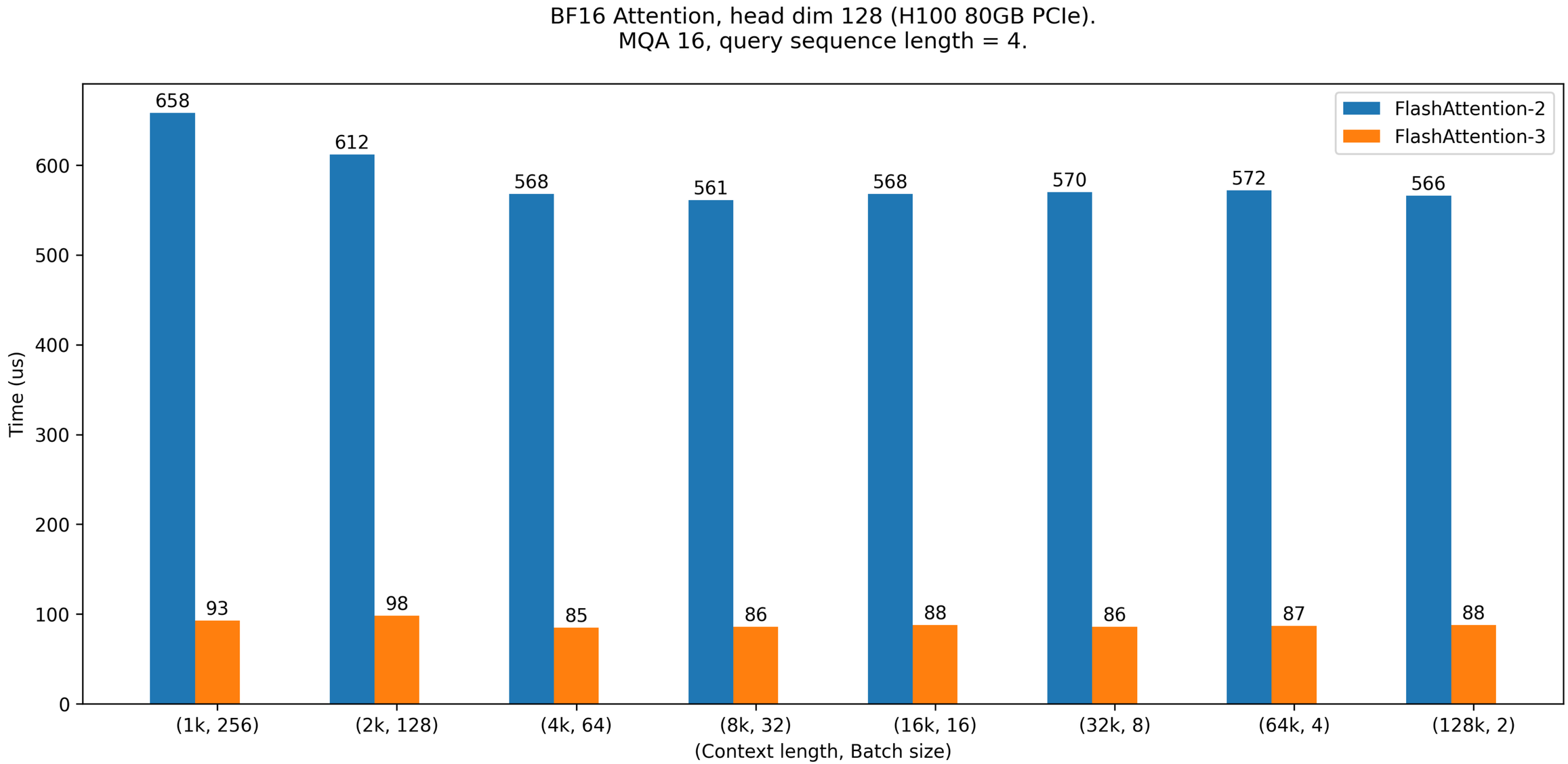


Packed 64x128 Query Tile (16 query heads, 4 query tokens)

FA-2 already did this for the case of **one** query token, which is a simple reshape.
In FA-3, we extend to the more involved case of arbitrary query length.

- Also benefits some compute-bound situations with tile quantization effects

BF16 Decode Benchmark for MQA. Lower is better!



Summary – FlashAttention

Fast and **accurate** attention optimized for modern hardware

Key algorithmic ideas: **asynchrony**, **low-precision**
- **Persistent kernels** with **LPT scheduling** for causal attention
- For inference: **Split KV** (Flash-Decoding) and **GQA packing**

Upshot: **faster** training, **better** models with **longer** sequences

Code:
https://github.com/Dao-AILab/flash-attention
https://github.com/NVIDIA/cutlass/tree/main/examples/77_blackwell_fmha