CUTLASS and FlashAttention-3

Jay Shah, Colfax Research
https://research.colfax-intl.com/

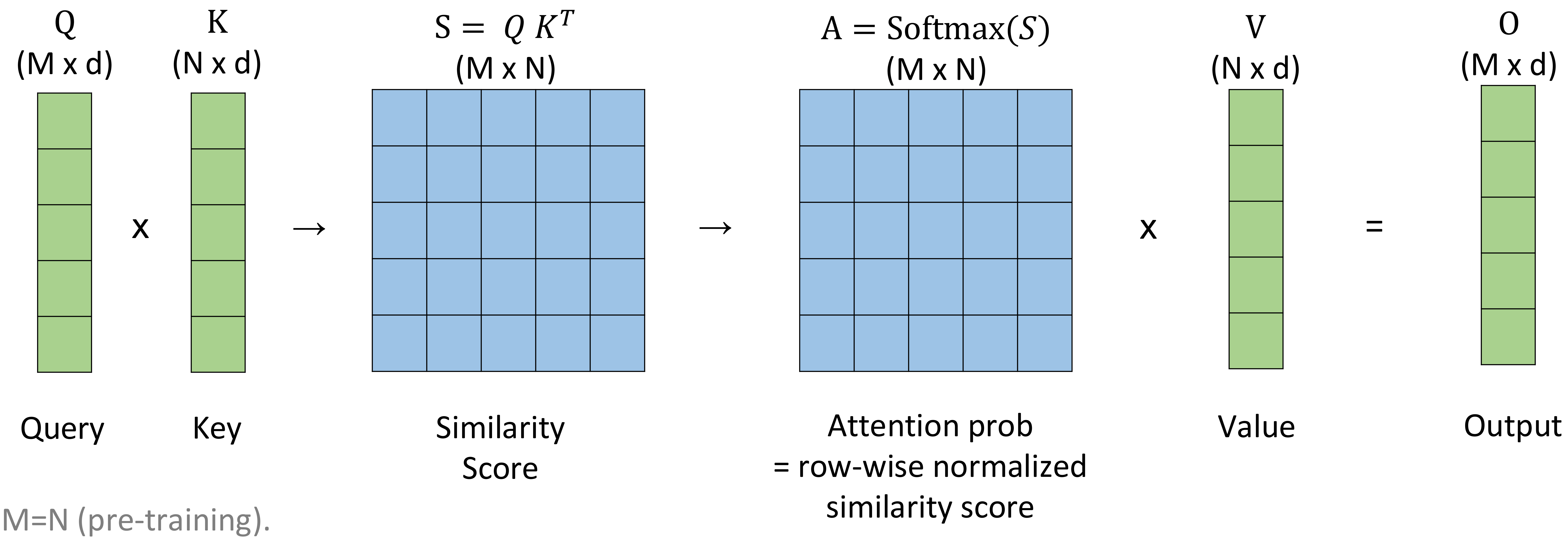
Outline of talk:

1. Recap of attention and Flash Attention.

2. High-level overview of the FlashAttention-3 algorithm.

3. Translating the algorithm into working code built on CUTLASS.

Background: Attention Mechanism



Suppose M=N (pre-training).
Typical sequence length N: 1K – 16K
Head dimension d: 64 – 256

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

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

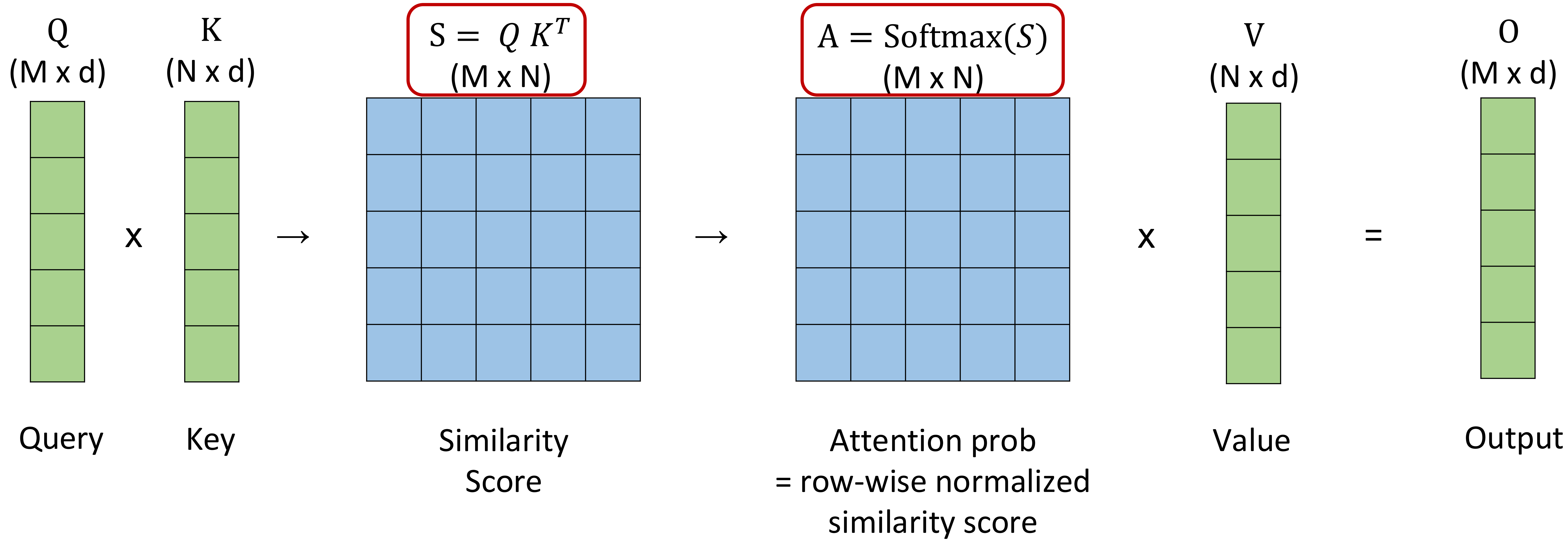
Attention scales quadratically in sequence length N

Naïve (Standard) Attention Algorithm

Algorithm 1 Standard Attention

Load Q and K by blocks from HBM.
Compute $S = (1/\sqrt{d})QK^T$ (GEMM-I).
Write S to HBM.
Read S from HBM.
Compute $S = S - \text{rowmax}(S)$.
Compute $P = \text{softmax}(S)$.
Write P to HBM.
Load P and V by blocks from HBM.
Compute $O = PV$ (GEMM-II).
Write O to HBM.

Naïve Attention is Bottlenecked by Memory Reads/Writes



The biggest cost is in moving the bits!

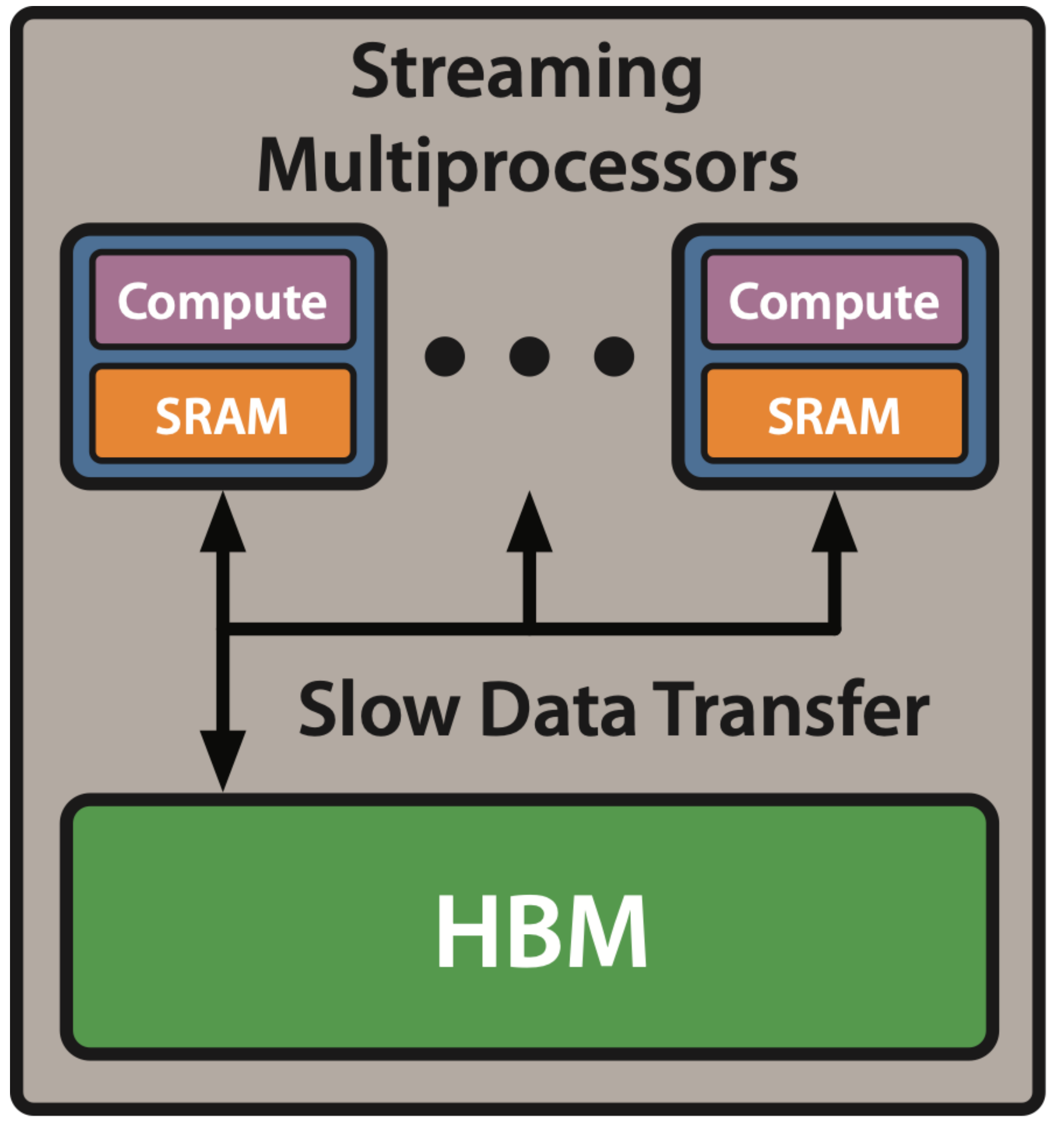
Standard implementation requires repeated R/W from slow GPU memory

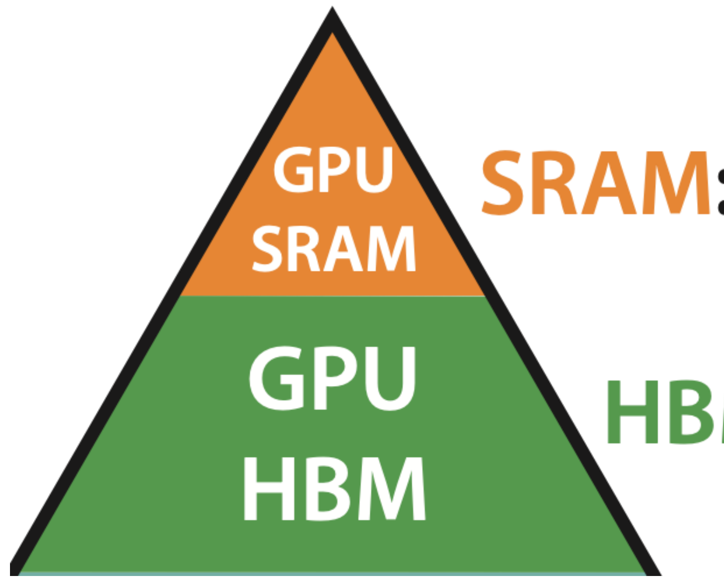
Background: GPU Compute Model & Memory Hierarchy

2. Data moved to compute units & SRAM for computation

1. Inputs start out in HBM (GPU memory)

3. Output written back to HBM





SRAM: 19 TB/s (20 MB)

HBM: 1.5 TB/s (40 GB)

[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: Tiling and Recomputation

Challenges:

(1) Don’t materialize scores matrix to HBM, and compute softmax normalization without access to full input.

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

Approaches:

(1) Kernel fusion with tiling and online softmax: for a given tile of Q, load KV block by block and accumulate values of corresponding tile of O.

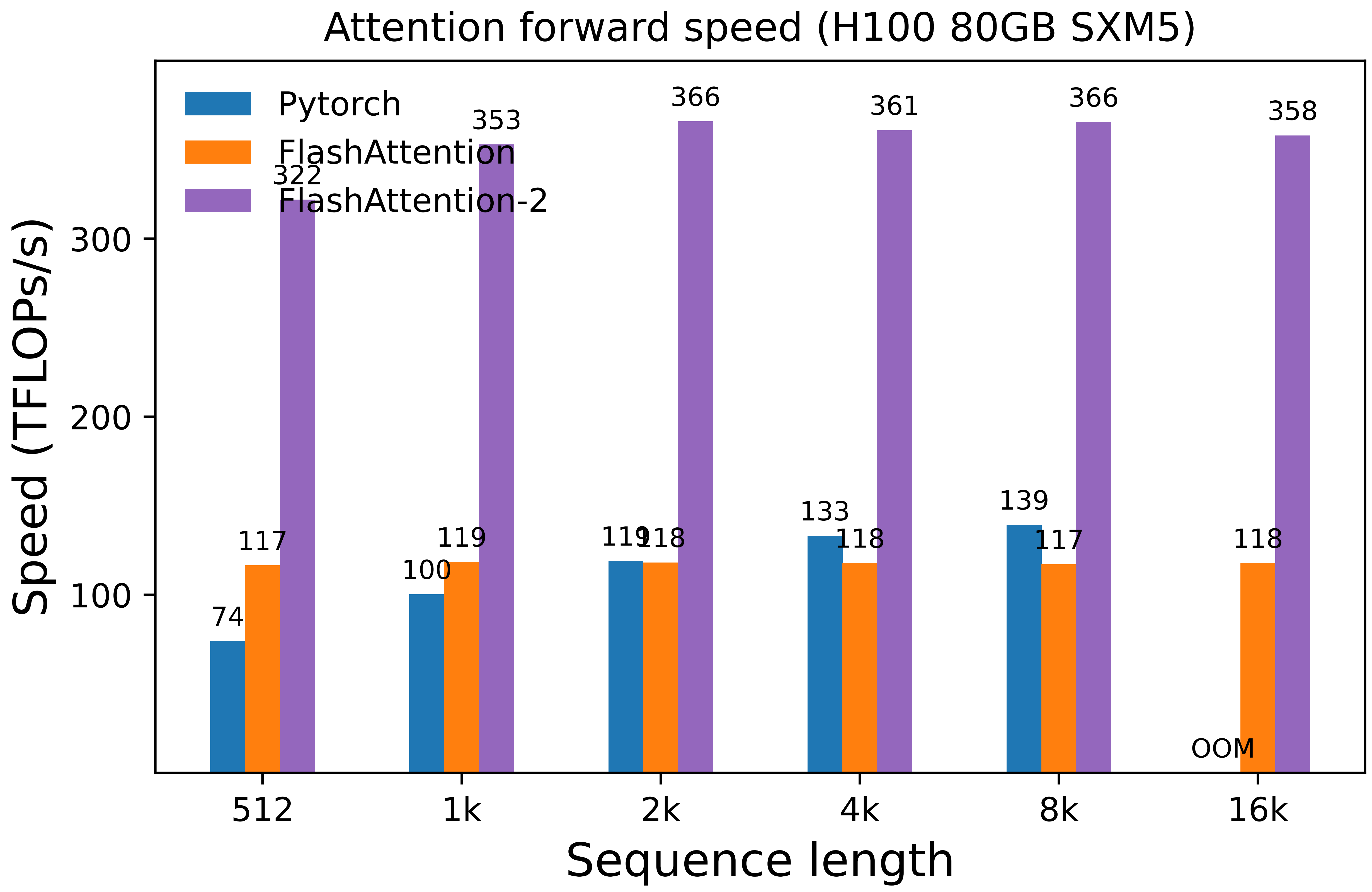
(2) Recomputation: Don’t store attn. matrix from forward, recompute it in the backward.

FlashAttention-2 Algorithm

Algorithm 2 FlashAttention-2 (FMHA)

for i **in** range(tiles of Q) **do**
    Load $bM \times d$ tile Q_i from HBM to SMEM.
    Initialize $bM \times d$ accumulator $O_i = (0)$.
    Initialize $bM \times 2$ rowmax $m_i = (-\infty)$ and $bM \times 1$ rowsum $\Sigma_i = (0)$.
    for j **in** range(tiles of K) **do**
        Load $bN \times d$ tile K_j from HBM to SMEM.
        Compute $S_{ij} = (1/\sqrt{d})(Q_i K_j^T)$ (SS-GEMM-I).
        Update rowmax $m_i = (m_i^{\text{new}}, m_i^{\text{old}})$, tracking rowmax at steps j and $j-1$.
        Compute $\widetilde{P}_{ij} = \exp(S_{ij} - m_i^{\text{new}})$.
        Update rowsum $\Sigma_i = \exp(m_i^{\text{old}} - m_i^{\text{new}})\Sigma_i + \text{rowsum}(\widetilde{P}_{ij})$.
        Load $bN \times d$ tile V_j from HBM to SMEM.
        Compute $O_i = \exp(m_i^{\text{old}} - m_i^{\text{new}})O_i + \widetilde{P}_{ij}V_j$ (RS-GEMM-II).
    end for
    Compute $O_i = (1/\Sigma_i)O_i$.
    Write O_i to HBM.
end for

Challenge: Optimizing FlashAttention for Modern Hardware - H100



FA2 only gets to 35-40% utilization (no WGMMA, no TMA)

FlashAttention-3: Optimizing FlashAttention for H100 GPU

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

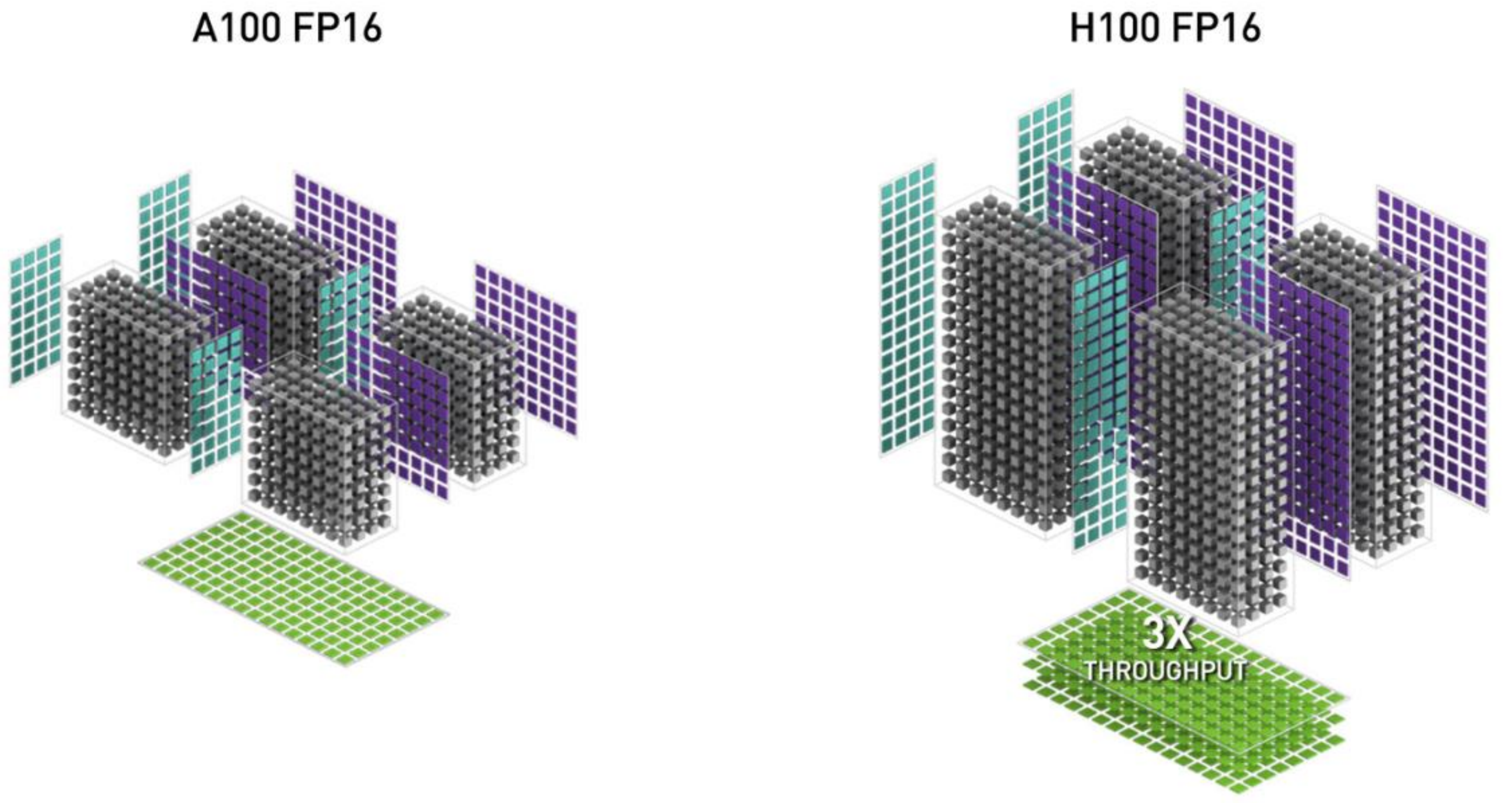
1. **New instructions** on H100:
   - **WGMMA**: higher throughput MMA primitive, async,
     collectively executed by a warpgroup (= 4 contiguous warps)
   - **TMA**: faster loading from gmem <-> smem, async, saves registers

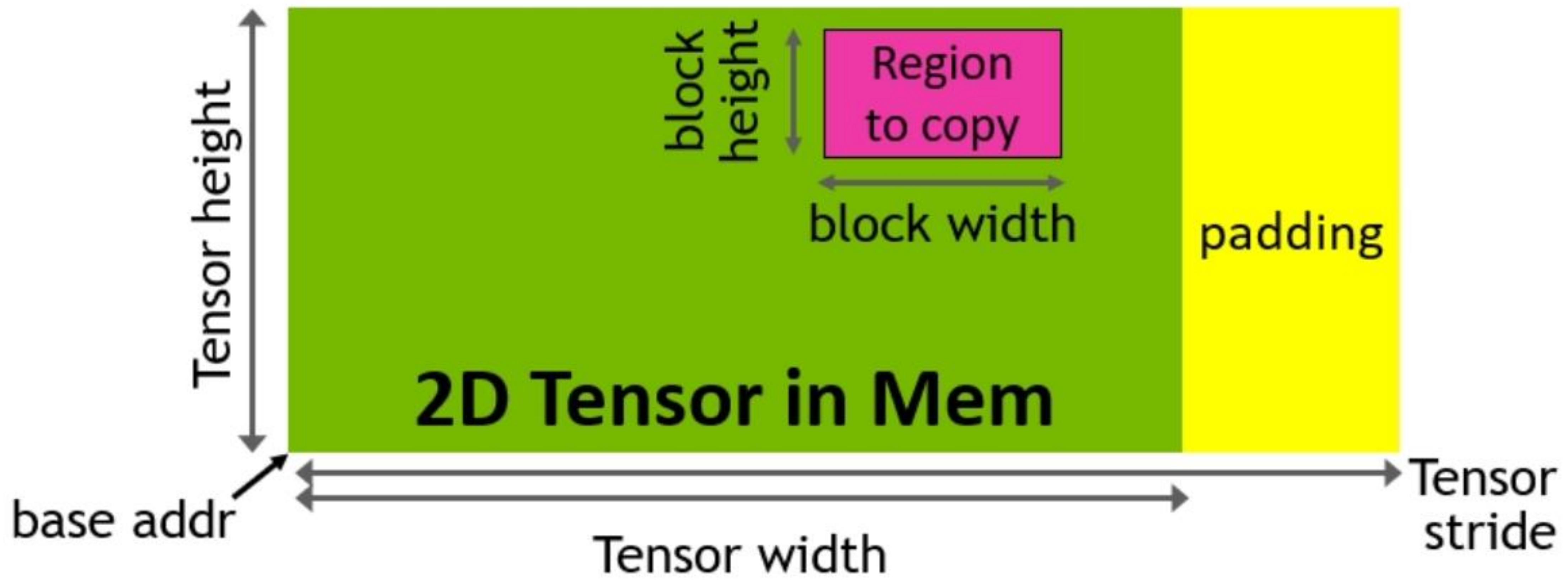
2. **Asynchrony**
   - Builds on asynchronous wgmma, TMA, transaction barrier
   - Inter-warpgroup overlapping: warp-specialization, pingpong
   - Intra-warpgroup overlapping: softmax and async matmul

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

Upshot: **1.6-3x** speedup

New Instructions: WGMMA & TMA





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

TMA: accelerate gmem -> smem, saves registers

WGMMA and TMA integrate into a producer-consumer warp-specialized pipelined design.
Use warpgroup-wide register reallocation to give consumers greater share of registers.

FlashAttention-3 Algorithm: CTA View

Algorithm 1 FLASHATTENTION-3 forward pass **without** intra-consumer overlapping – CTA view

Require: Matrices $\mathbf{Q}_i \in \mathbb{R}^{B_r \times d}$ and $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, key block size B_c with $T_c = \lceil \frac{N}{B_c} \rceil$.

Initialize pipeline object to manage barrier synchronization with s-stage circular SMEM buffer.
if in producer warpgroup **then**
    Deallocate predetermined number of registers.
    Issue load $\mathbf{Q}_i$ from HBM to shared memory.
    Upon completion, commit to notify consumer of the load of $\mathbf{Q}_i$.
    for $0 \le j < T_c$ **do**
        Wait for the $(j\%s)$th stage of the buffer to be consumed.
        Issue loads of $\mathbf{K}_j, \mathbf{V}_j$ from HBM to shared memory at the $(j\%s)$th stage of the buffer.
        Upon completion, commit to notify consumers of the loads of $\mathbf{K}_j, \mathbf{V}_j$.
    end for
else
    Reallocate predetermined number of registers as function of number of consumer warps.
    On-chip, initialize $\mathbf{O}_i = (0) \in \mathbb{R}^{B_r \times d}$ and $\ell_i, m_i = (0), (-\infty) \in \mathbb{R}^{B_r}$.
    Wait for $\mathbf{Q}_i$ to be loaded in shared memory.
    for $0 \le j < T_c$ **do**
        Wait for $\mathbf{K}_j$ to be loaded in shared memory.
        Compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T$ (SS-GEMM). Commit and wait.
        Store $m_i^{\text{old}} = m_i$ and compute $m_i = \max(m_i^{\text{old}}, \text{rowmax}(\mathbf{S}_i^{(j)}))$.
        Compute $\widetilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i)$ and $\ell_i = \exp(m_i^{\text{old}} - m_i)\ell_i + \text{rowsum}(\widetilde{\mathbf{P}}_i^{(j)})$.
        Wait for $\mathbf{V}_j$ to be loaded in shared memory.
        Compute $\mathbf{O}_i = \text{diag}(\exp(m_i^{\text{old}} - m_i))\mathbf{O}_i + \widetilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$ (RS-GEMM). Commit and wait.
        Release the $(j\%s)$th stage of the buffer for the producer.
    end for
    Compute $\mathbf{O}_i = \text{diag}(\ell_i)^{-1}\mathbf{O}_i$ and $L_i = m_i + \log(\ell_i)$.
    Write $\mathbf{O}_i$ and L_i to HBM as the ith block of $\mathbf{O}$ and L.
end if

Asynchrony: Overlapping GEMM and Softmax

Why overlapping?

Example: headdim 128, block size 128 x 192

FP16 WGMMA: 2 x 2 x 128 x 192 x 128 = 12.6 MFLOPS, 4096 FLOPS/cycle -> **3072 cycles**

MUFU.EX2: 128 x 192 = 24.6k OPS, 16 OPS/cycle -> **1536 cycles**

MUFU.EX2 takes 50% the cycles of WGMMA!

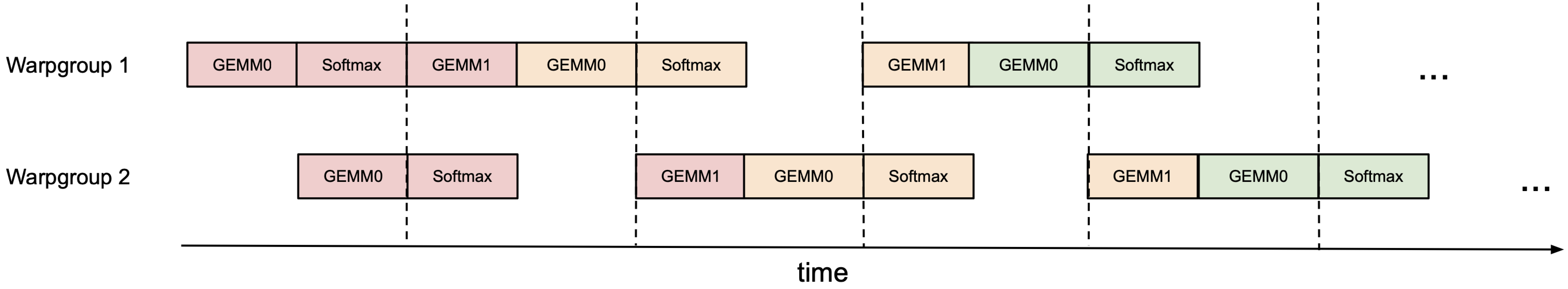
FP8 is even worse: WGMMA and EX2 both take 1536 cycles.

We want to be doing EX2 while tensor cores are busy with WGMMA.

Inter-warpgroup Overlapping of GEMM and Softmax

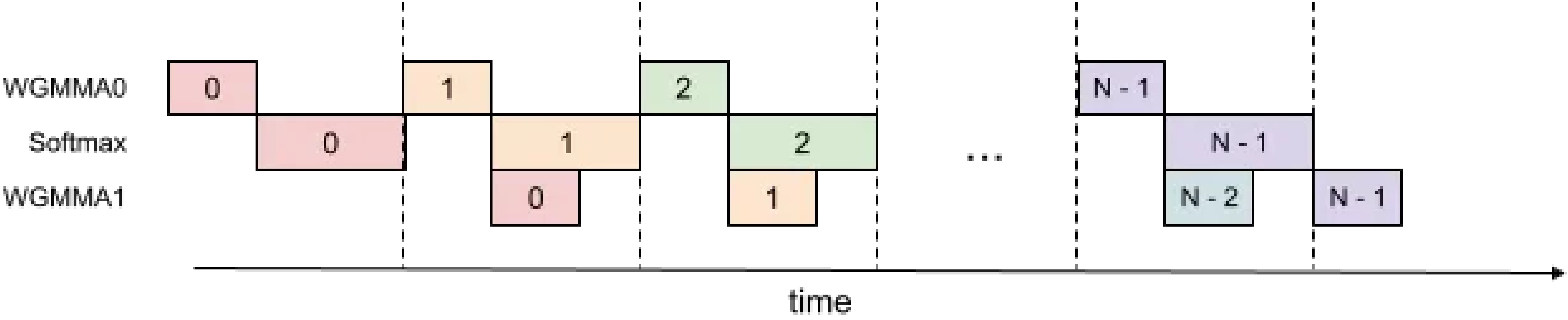
Easy solution: leave it to the warp schedulers!

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



2-stage pipelining: 640 TFLOPS -> 670 TFLOPS (but higher register pressure)

Note: Corrected image from talk.

FlashAttention-3 Algorithm: Consumer View

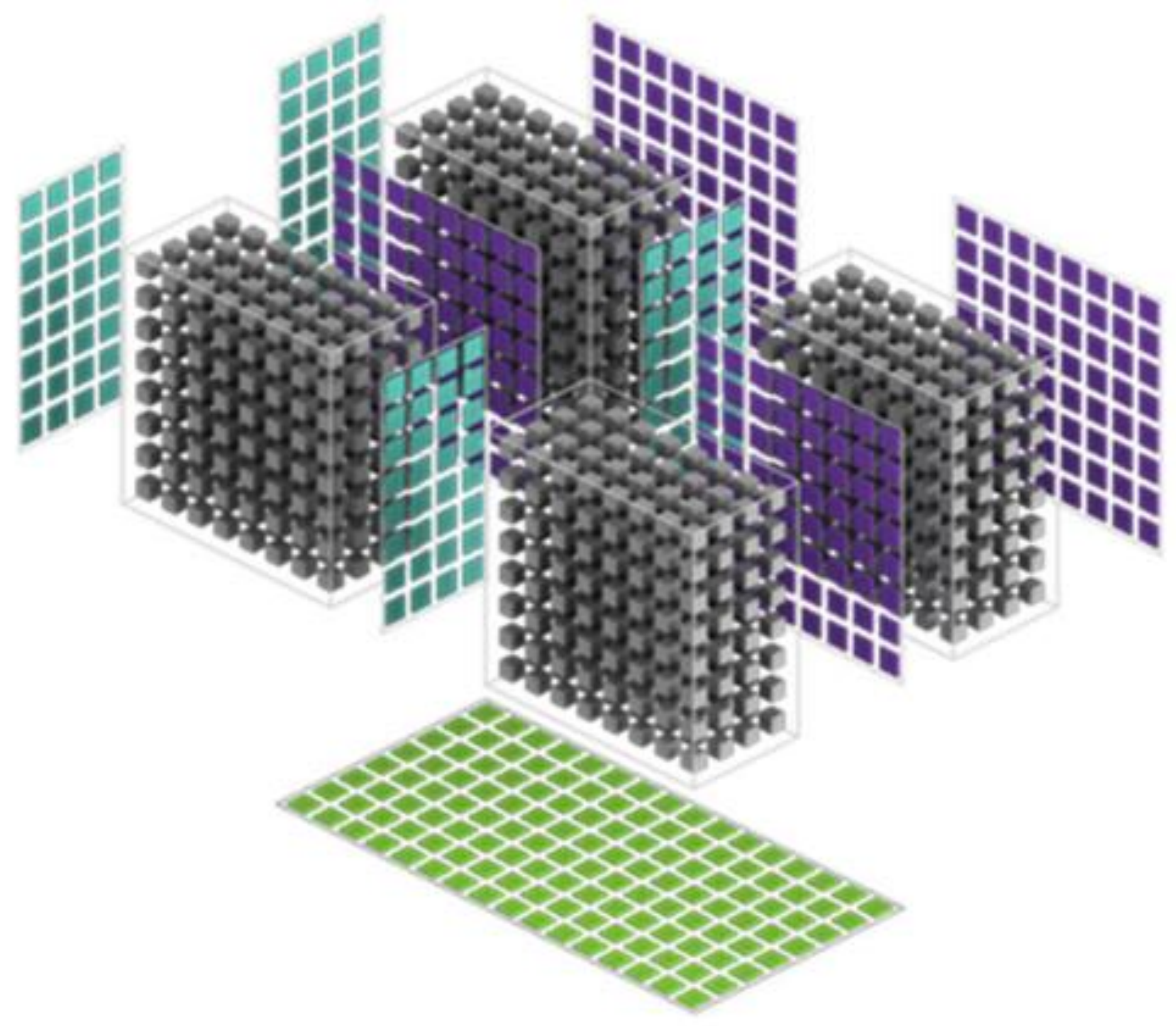
Algorithm 2 FLASHATTENTION-3 consumer warpgroup forward pass

Require: Matrices $\mathbf{Q}_i \in \mathbb{R}^{B_r \times d}$ and $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, key block size B_c with $T_c = \lceil \frac{N}{B_c} \rceil$.

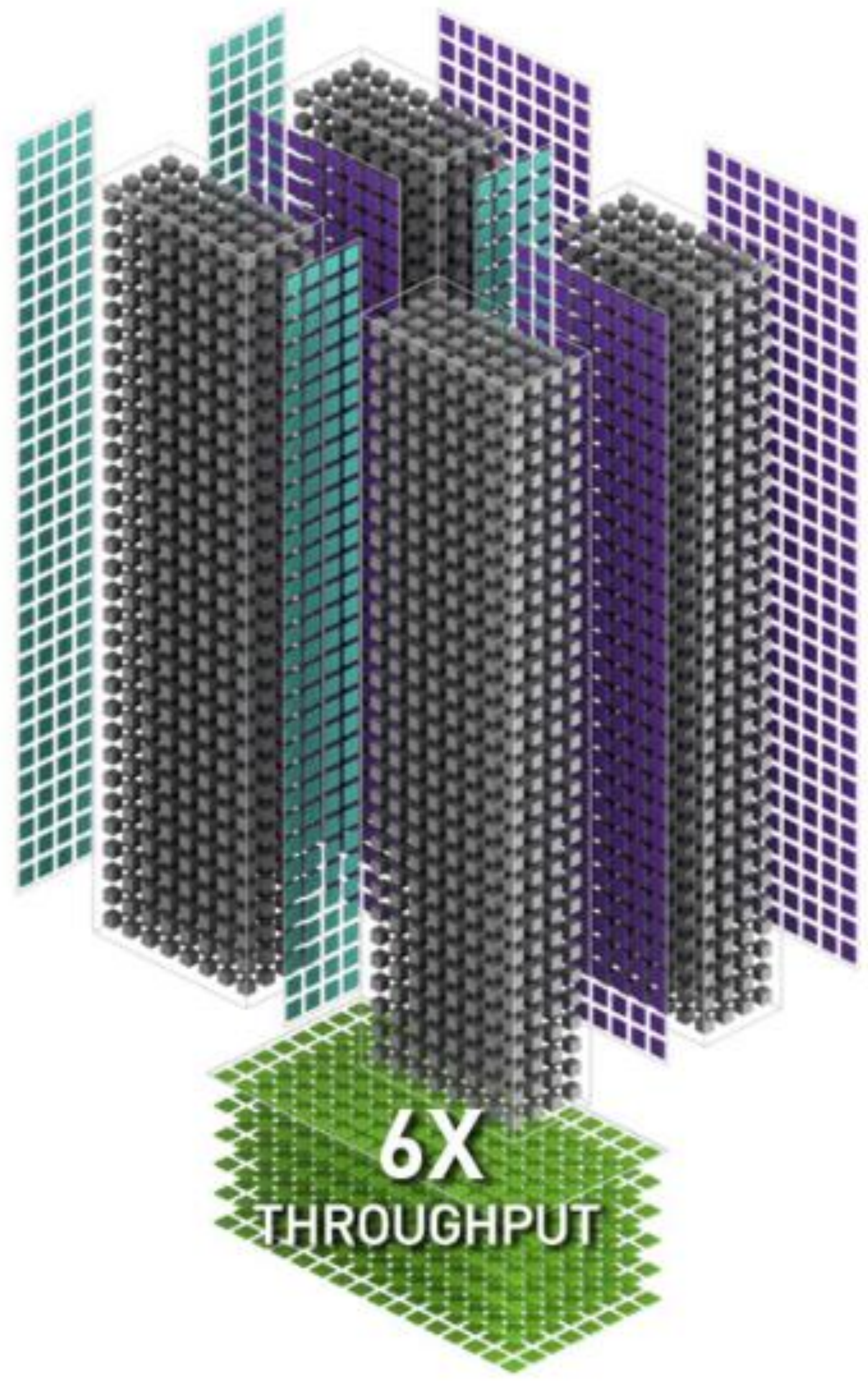
Reallocate predetermined number of registers as function of number of consumer warps.
On-chip, initialize $\mathbf{O}_i = (0) \in \mathbb{R}^{B_r \times d}$ and $\ell_i, m_i = (0), (-\infty) \in \mathbb{R}^{B_r}$.
Wait for $\mathbf{Q}_i$ and $\mathbf{K}_0$ to be loaded in shared memory.
Compute $\mathbf{S}_{\text{cur}} = \mathbf{Q}_i \mathbf{K}_0^T$ using WGMMA. Commit and wait.
Release the 0th stage of the buffer for $\mathbf{K}$.
Compute $m_i, \tilde{\mathbf{P}}_{\text{cur}}$ and ℓ_i based on $\mathbf{S}_{\text{cur}}$, and rescale $\mathbf{O}_i$.
for $1 \le j < T_c - 1$ **do**
    Wait for $\mathbf{K}_j$ to be loaded in shared memory.
    Compute $\mathbf{S}_{\text{next}} = \mathbf{Q}_i \mathbf{K}_j^T$ using WGMMA. Commit but do not wait.
    Wait for $\mathbf{V}_{j-1}$ to be loaded in shared memory.
    Compute $\mathbf{O}_i = \mathbf{O}_i + \tilde{\mathbf{P}}_{\text{cur}} \mathbf{V}_{j-1}$ using WGMMA. Commit but do not wait.
    Wait for the WGMMA $\mathbf{Q}_i \mathbf{K}_j^T$.
    Compute $m_i, \tilde{\mathbf{P}}_{\text{next}}$ and ℓ_i based on $\mathbf{S}_{\text{next}}$.
    Wait for the WGMMA $\tilde{\mathbf{P}}_{\text{cur}} \mathbf{V}_{j-1}$ and then rescale $\mathbf{O}_i$
    Release the $(j\%s)$th, resp. $(j-1\%s)$th stage of the buffer for $\mathbf{K}$, resp. $\mathbf{V}$.
    Copy $\mathbf{S}_{\text{next}}$ to $\mathbf{S}_{\text{cur}}$.
end for
Wait for $\mathbf{V}_{T_c-1}$ to be loaded in shared memory.
Compute $\mathbf{O}_i = \mathbf{O}_i + \tilde{\mathbf{P}}_{\text{last}} \mathbf{V}_{T_c-1}$ using WGMMA. Commit and wait.
Rescale $\mathbf{O}_i$ based on m_i. Compute L_i based on m_i and ℓ_i.
Epilogue: Write $\mathbf{O}_i$ and L_i to HBM as the i-th block of $\mathbf{O}$ and L.

Low-precision: FP8

A100 FP16



H100 FP8



FP8 doubles WGMMA throughput, but trade off accuracy

Layout conformance challenges with FP8

FP8 WGMMA: requires operand SMEM tensors to be memory-contiguous
in the inner dimension (k-major)

It is standard for QKV to be memory-contiguous in the head dimension (BSHD).
- Note: TMA can't change the contiguous mode.

For gemm0 (Q.K^T), this is fine as is. For gemm1 (P.V), we need to transpose V.

Solution: In-kernel transpose of V in the producer warpgroup.
Uses LDSM/STSM instructions with custom layouts and byte permute in-between.

Layout conformance challenges with FP8, ctd.

We also need to reshape layout of scores accumulator for gemm1.
Why? FP32 accumulator layout differs from FP8 operand A layout.

T0 {d0, d1}	T1 {d0, d1}	T2 {d0, d1}	T3 {d0, d1}	T0 {d4, d5}	T1 {d4, d5}	T2 {d4, d5}	T3 {d4, d5}
T0 {d2, d3}	T1 {d2, d3}	T2 {d2, d3}	T3 {d2, d3}	T0 {d6, d7}	T1 {d6, d7}	T2 {d6, d7}	T3 {d6, d7}

Figure 3: FP32 accumulator register WGMMA layout – rows 0 and 8, threads 0-3, entries 0-7.

T0 {a0, a1}	T0 {a2, a3}	T1 {a0, a1}	T1 {a2, a3}	T2 {a0, a1}	T2 {a2, a3}	T3 {a0, a1}	T3 {a2, a3}
T0 {a4, a5}	T0 {a6, a7}	T1 {a4, a5}	T1 {a6, a7}	T2 {a4, a5}	T2 {a6, a7}	T3 {a4, a5}	T3 {a6, a7}

Figure 4: FP8 operand A register WGMMA layout – rows 0 and 8, threads 0-3, entries 0-7.

Note: Can use in-kernel "transpose" to write out row permutation of V
such that we can avoid shuffle instructions for the reshape (but not byte permute).

Persistent Kernels in FlashAttention

Idea: Decouple *physical* CTAs from *logical* work tiles, launching fixed number of CTAs.
Can then overlap epilogue of current work tile with prologue loads of next work tile.

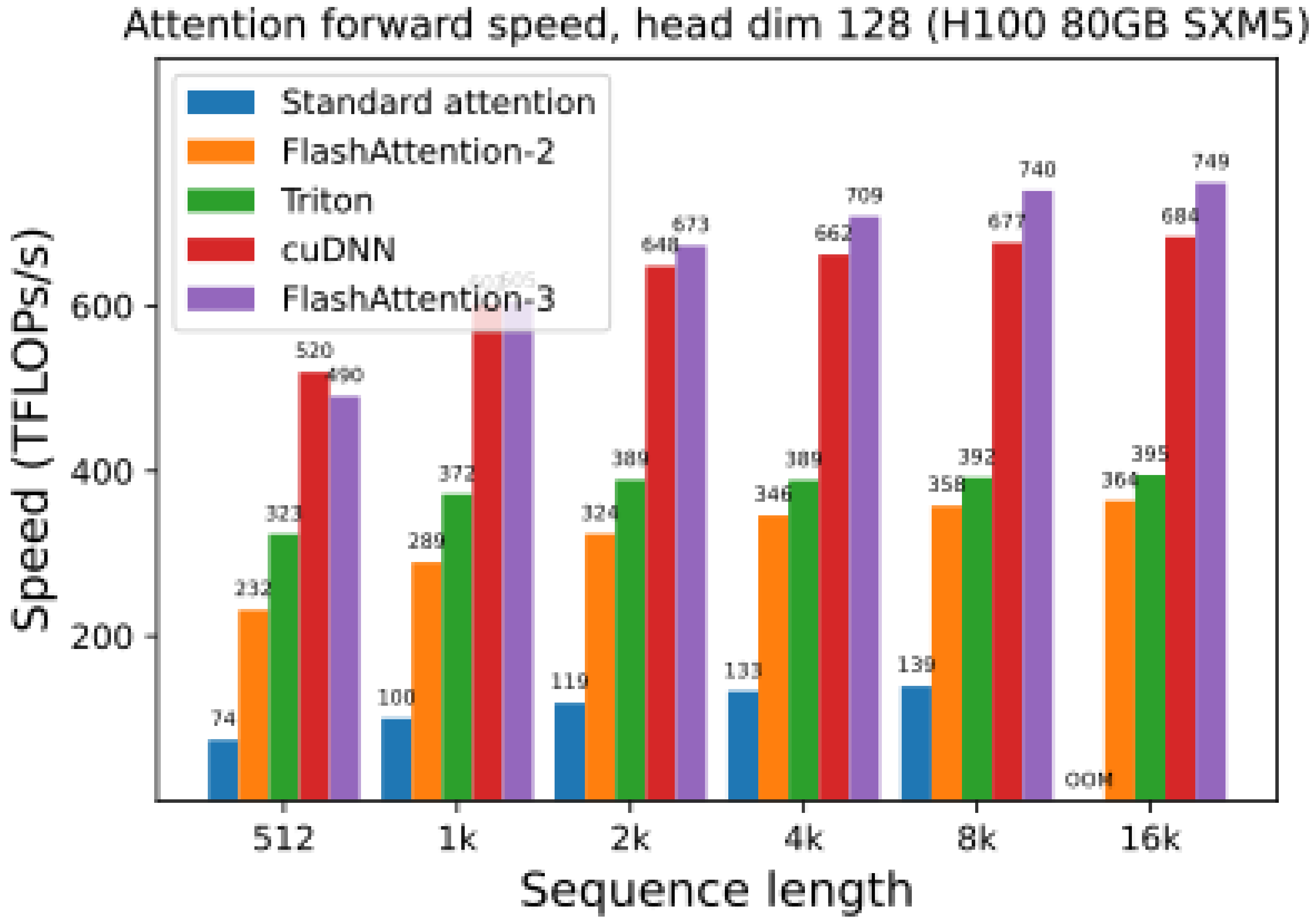
Example: Seqlen = 4096, Heads = 8, Batch = 4. Fix BlockM = 128, so mblocks = 32.
Have 32*8*4 = 1024 work tiles to process in the kernel.

Without persistent kernel, launch CUDA grid with dims = (32, 8, 4), so 1024 CTAs.
1-to-1 mapping of CTAs with work tiles.

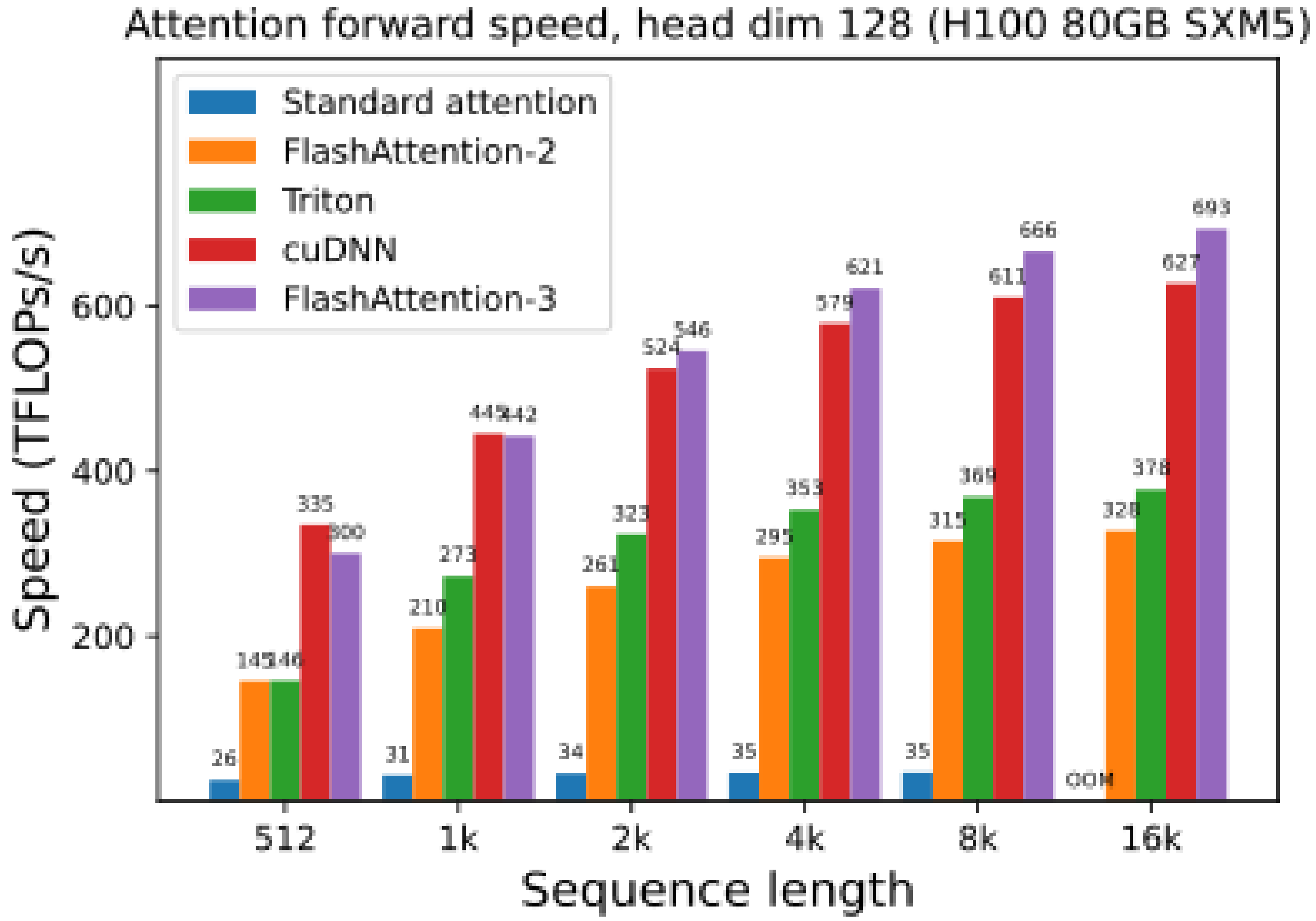
With persistent kernel on H100 SXM5 GPU, launch 132 CTAs = num SMs.
Each CTA runs over a fraction of the 1024 work tiles.

Can dynamically allocate work tiles to next available CTA in persistent kernel.
Helps with load balancing when doing causal masking.

BF16 Benchmark: 1.6-2.0x speedup

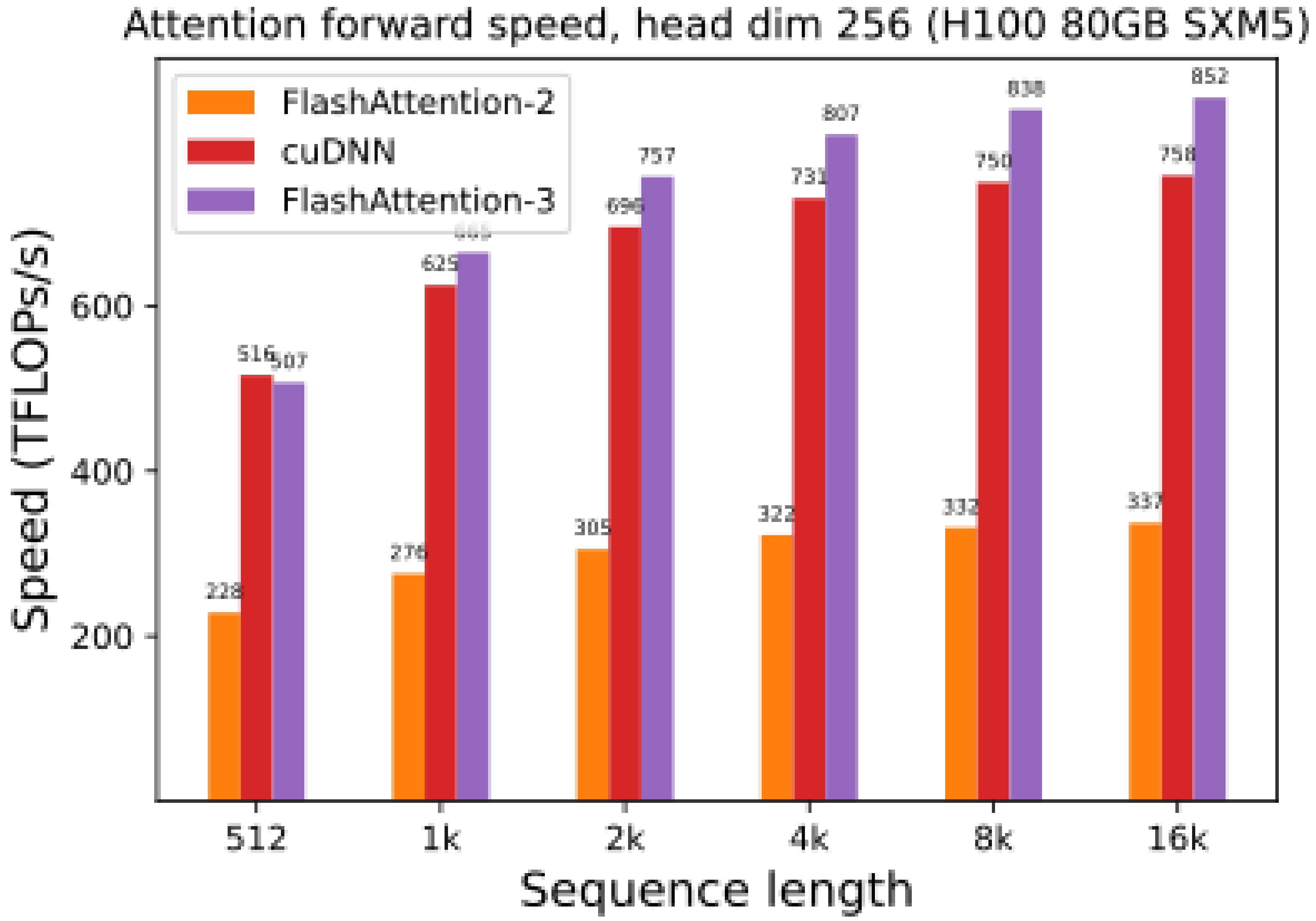


Without causal mask

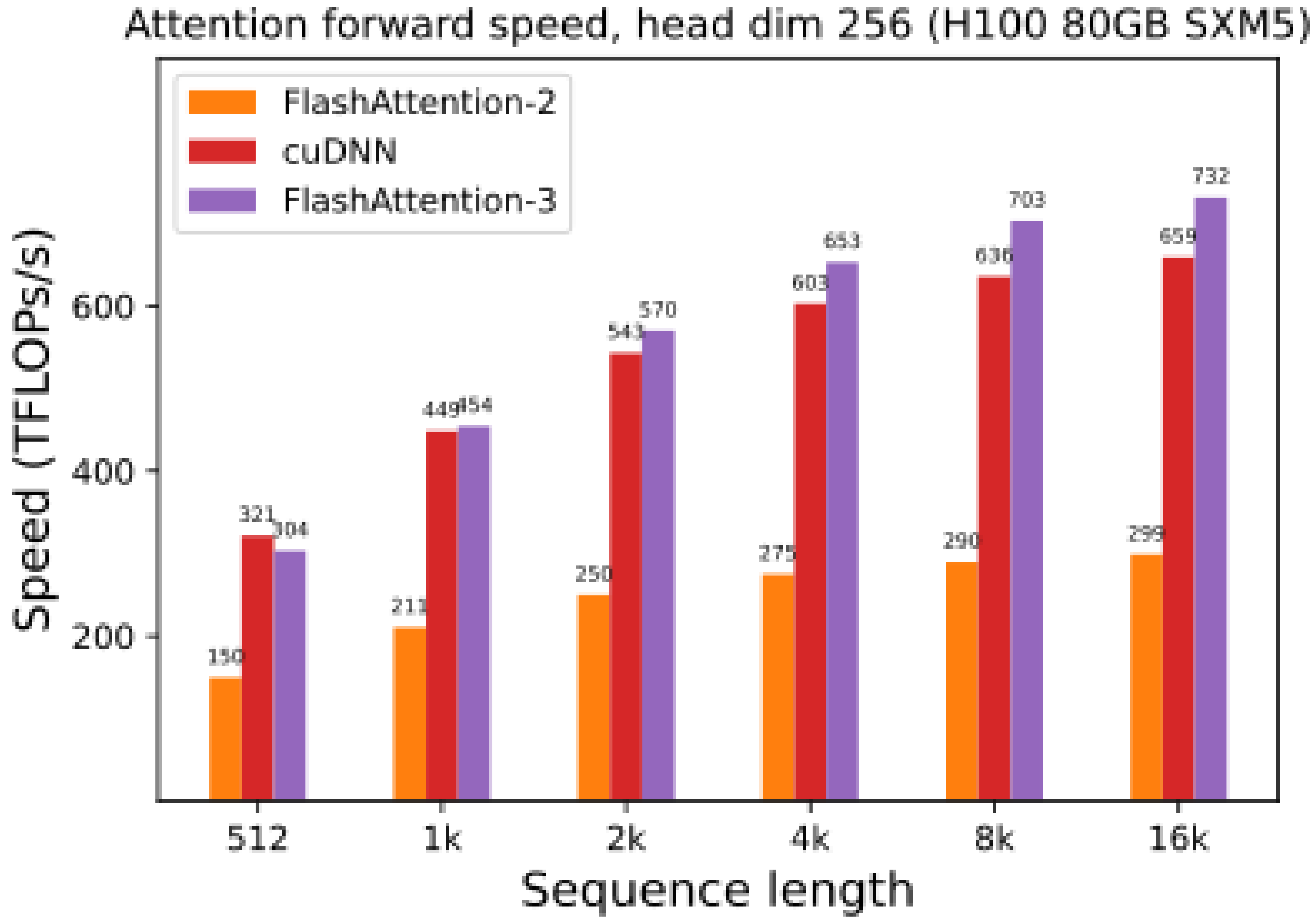


With causal mask

BF16 Benchmark: reach up to 850 TFLOPS

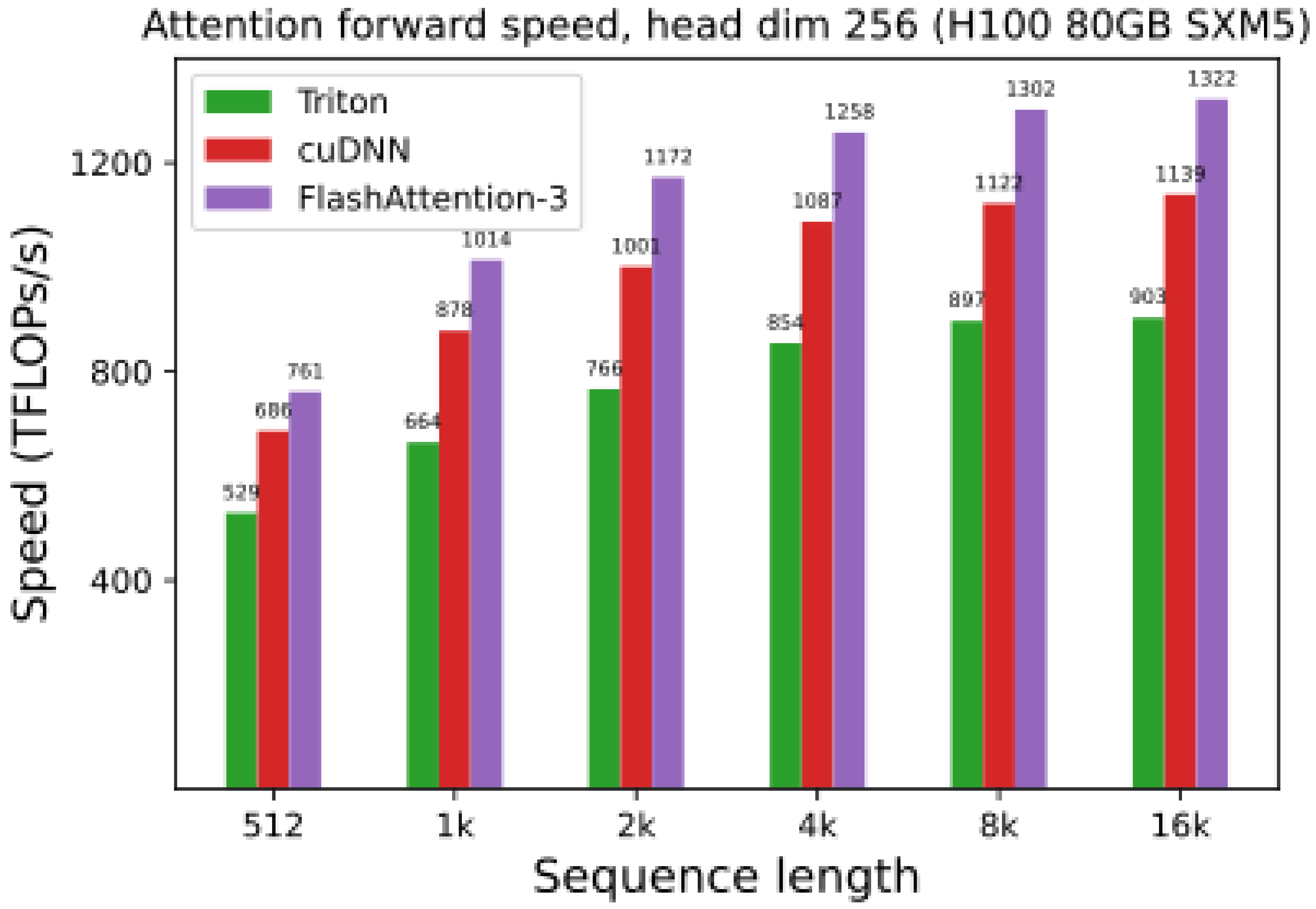


Without causal mask

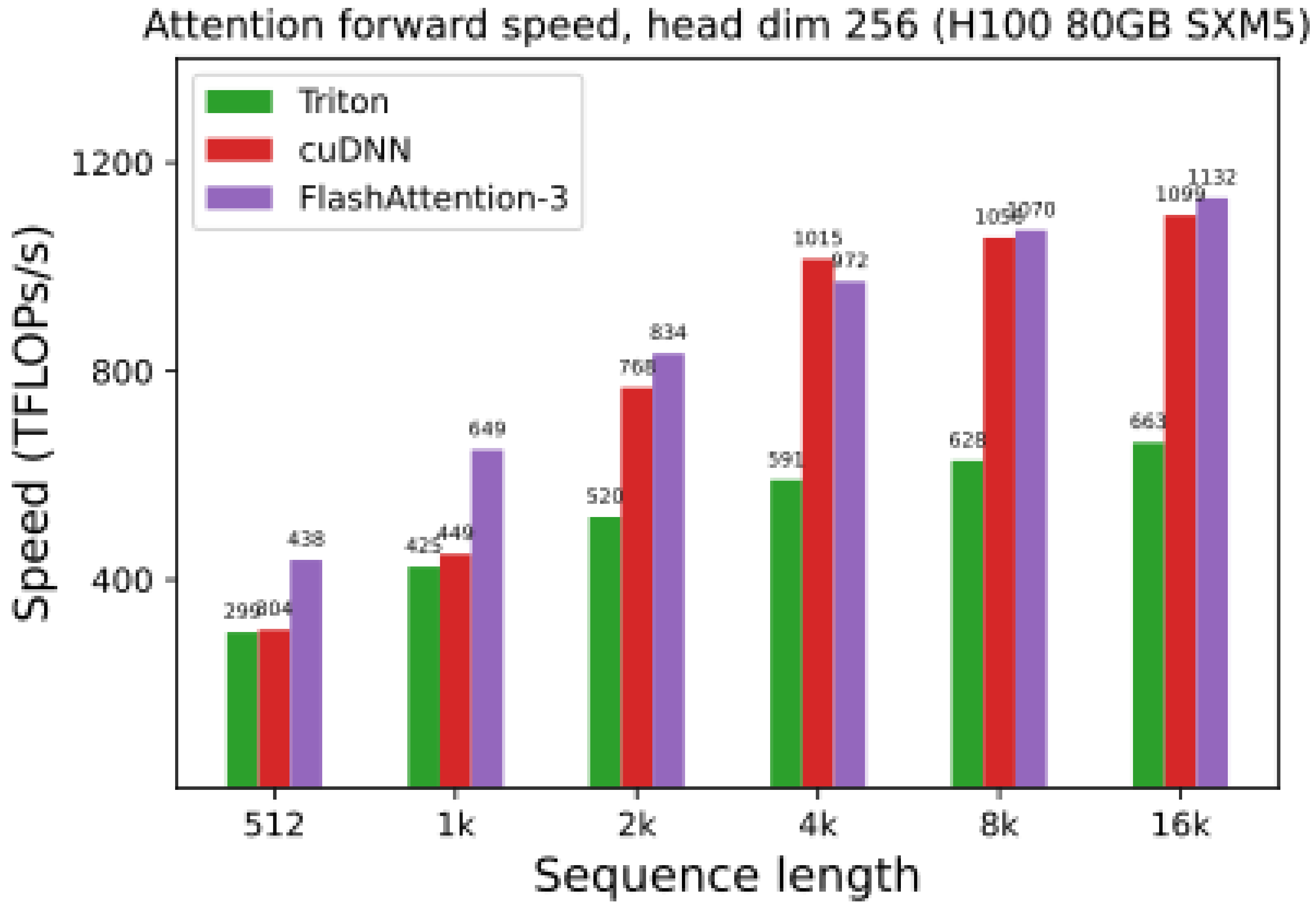


With causal mask

FP8 Benchmark: up to 1.3 PFLOPS

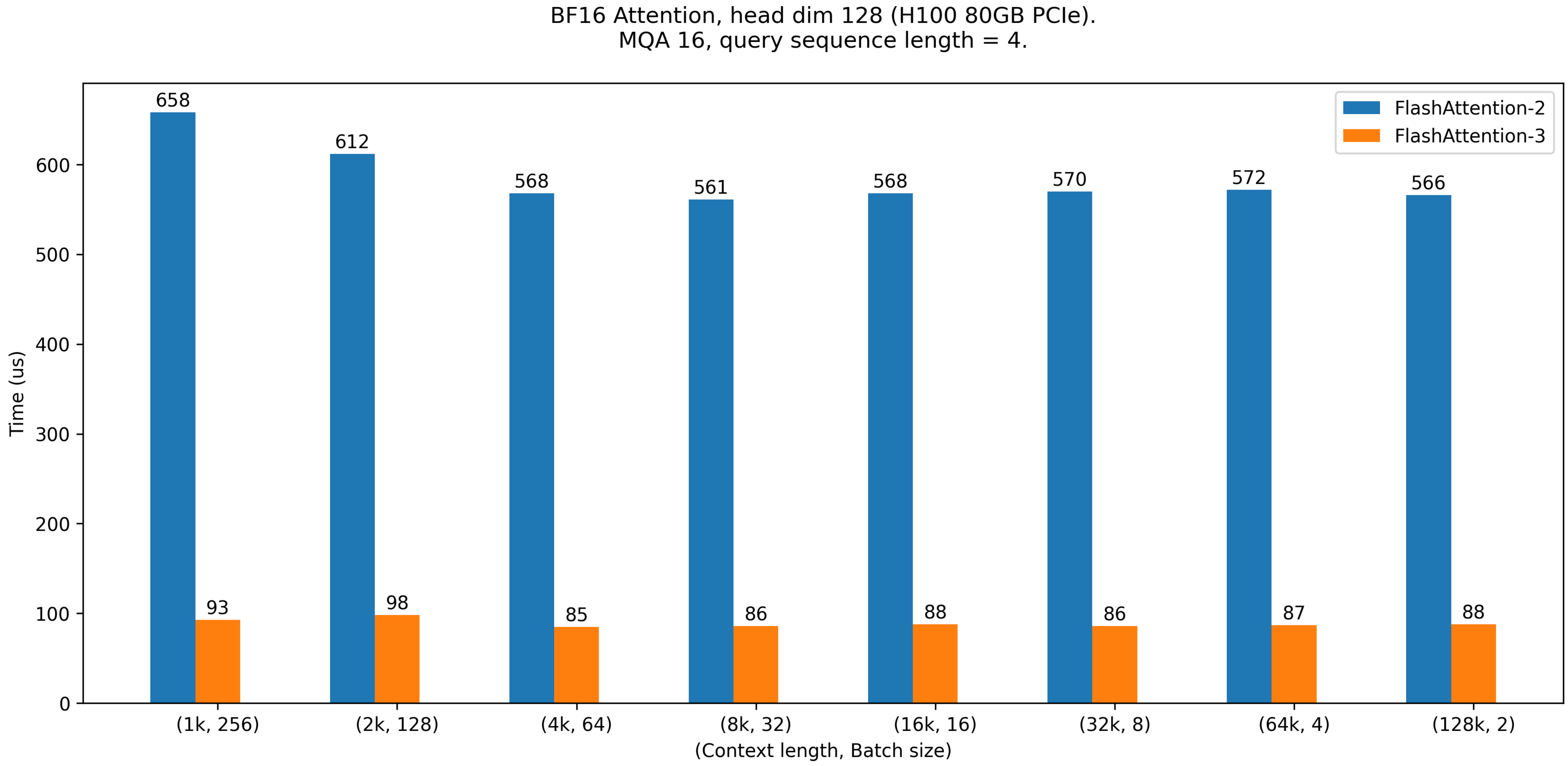


Without causal mask



With causal mask

BF16 Decode Benchmark for MQA. Lower is better!



FlashAttention-3 for Decoding Inference

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

Optimizations:
1) Split along the KV sequence length to occupy the GPU with enough work
- already in FlashAttention-2 as **Flash-Decoding**. Reuse the same algorithm.

2) **GQA packing**: Pack multiple query heads into a single query tile.

Summary – FlashAttention-3

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

Key algorithmic ideas: **asynchrony**, **low-precision**
- 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

Building FlashAttention-3 with CUTLASS

Overall structure has three classes:

1. CollectiveMainloop for load and mma.

2. CollectiveEpilogue for store.

3. TileScheduler to manage work loop for persistent kernel.

Each class has its own set of kernel parameters (`to_underlying_arguments`).

```
CollectiveMainloop collective_mainloop
CollectiveEpilogue collective_epilogue
if producer:
    reg_dealloc(LoadRegisterRequirement)
    work_tile = get_initial_work()
    while(work_tile.is_valid()):
        auto block_coord = work_tile_info.get_block_coord()
        collective_mainloop.load(block_coord)
        work_tile = get_next_work()
else:
    reg_alloc(MmaRegisterRequirement)
    work_tile = get_initial_work()
    while(work_tile.is_valid()):
        auto block_coord = work_tile_info.get_block_coord()
        Tensor tOrO = collective_mainloop.mma(block_coord)
        collective_epilogue.store(tOrO, block_coord)
        work_tile = get_next_work()
```