FlashAttention-4: Algorithm and Kernel Pipelining Co-Design for Asymmetric Hardware Scaling

Ted Zadouri[*1,6] Markus Hoehnerbach[*2] Jay Shah[*3]
Timmy Liu[4] Vijay Thakkar[2,5] Tri Dao[1,6]
[1]Princeton University [2]Meta [3]Colfax Research [4]NVIDIA [5]Georgia Tech [6]Together AI

March 5, 2026

Abstract

Attention, as a core layer of the ubiquitous Transformer architecture, is the bottleneck for large language models and long-context applications. While FlashAttention-3 optimized attention for Hopper GPUs through asynchronous execution and warp specialization, it primarily targets the H100 architecture. The AI industry has rapidly transitioned to deploying Blackwell-based systems such as the B200 and GB200, which exhibit fundamentally different performance characteristics due to asymmetric hardware scaling: tensor core throughput doubles while other functional units (shared memory bandwidth, exponential units) scale more slowly or remain unchanged. We develop several techniques to address these shifting bottlenecks on Blackwell GPUs: (1) redesigned pipelines that exploit fully asynchronous MMA operations and larger tile sizes, (2) software-emulated exponential and conditional softmax rescaling that reduces non-matmul operations, and (3) leveraging tensor memory and the 2-CTA MMA mode to reduce shared memory traffic and atomic adds in the backward pass. We demonstrate that our method, FlashAttention-4, achieves up to 1.3× speedup over cuDNN 9.13 and 2.7× over Triton on B200 GPUs with BF16, reaching up to 1613 TFLOPs/s (71% utilization). Beyond algorithmic innovations, we implement FlashAttention-4 entirely in CuTe-DSL embedded in Python, achieving 20-30× faster compile times compared to traditional C++ template-based approaches while maintaining full expressivity.

1 Introduction

The Transformer architecture [27] continues to serve as the primary backbone for nearly all AI applications, from large language models [2] to vision [8] and multimodal systems. For Transformers, the attention mechanism constitutes the primary computational bottleneck, with self-attention scores computed between queries and keys exhibiting quadratic scaling in sequence length. Scaling attention to longer contexts unlocks new capabilities such as reasoning over multiple documents [10, 24], modeling entire codebases [22], and processing high-resolution videos [3, 11]. Meanwhile, accelerator hardware continues to evolve rapidly [19], with each generation delivering substantially higher peak compute throughput. However, this evolution is asymmetric: while matrix multiplication units scale aggressively, other functional units such as memory bandwidth and specialized compute units scale more slowly, creating increasingly unbalanced hardware pipelines that demand careful algorithmic co-design.

This has generated sustained interest in making attention faster through algorithmic innovations that deeply integrate knowledge of GPU hardware characteristics. Dao et al. [6] introduced

[*]Equal contribution

FlashAttention, which eliminates intermediate reads/writes to slow global memory through novel tiling and kernel fusion. Dao [5] restructured this as FlashAttention-2 to parallelize over the sequence length dimension, improving GPU occupancy. Shah et al. [23] further adapted the algorithm for Hopper GPUs as FlashAttention-3, exploiting asynchronous execution through warp specialization and incorporating FP8 support. Recent work has also explored low-precision attention: SageAttention [13] achieves speedups through INT8 quantization, SageAttention2 [12] extends this with INT4/FP8 quantization, and SageAttention3 [14] demonstrates FP4 quantization on Blackwell consumer GPUs. However, these approaches primarily target consumer GPUs, while most AI compute is deployed on datacenter GPUs. Meanwhile, FlashAttention-3 primarily targets the NVIDIA Hopper H100 architecture, while the AI industry has rapidly transitioned to deploying Blackwell-based datacenter systems [19] such as the B200 and GB200, which represent a new generation of GPUs with fundamentally different performance characteristics.

A critical trend in accelerator evolution is the asymmetric scaling of hardware units. Although Blackwell B200 doubles the tensor core throughput compared to Hopper H100 (2.25 PFLOPS vs. 1 PFLOPS for FP16/BF16), other functional units (shared memory bandwidth, exponential units, and integer/floating point ALUs) scale more slowly or remain unchanged. As a result, non-MMA resources emerge as bottlenecks. Our roofline analysis (Sections 3.1 and 3.2) reveals that for typical attention workloads on Blackwell, surprisingly, shared memory traffic and exponential operations now dominate execution time, exceeding MMA compute by 25-60%. Additionally, Blackwell introduces new architectural features: 256 KB of tensor memory (TMEM) per SM for storing intermediate tensor core results, 128×128 MMA tiles (double the area of Hopper's 64×128), and fully asynchronous tensor core operations that write directly to TMEM. Simply porting existing attention algorithms to this new hardware either leaves significant performance on the table or is impossible due to lack of forward compatibility for Hopper MMA instructions.

To this end, we propose FlashAttention-4, which co-designs the algorithm and kernel implementation to address the shifting bottlenecks in modern GPU architectures. Rather than treating hardware as a uniform compute resource, we explicitly identify and mitigate bottlenecks in non-matmul units through algorithmic innovations:

1. **Redesigned pipeline for maximum overlap:** We develop new software pipelines for both forward and backward passes that exploit Blackwell's fully asynchronous MMA operations and larger tile sizes to maximize overlap between tensor cores, softmax computation, and memory operations.
2. **Exponential unit bottleneck mitigation:** For the forward pass, we implement software-emulated exponential functions using polynomial approximation on FMA units, increasing exponential throughput. We also introduce conditional softmax rescaling that skips unnecessary rescaling operations.
3. **Shared memory traffic reduction:** For the backward pass, we leverage tensor memory to store more intermediate results, reducing shared memory traffic. We also leverage Blackwell's 2-CTA MMA mode, so each CTA stages and loads half of operand B to further reduce shared memory traffic, which we exploit to restructure the dQ step to halve the number of atomic reductions. We also implement a deterministic execution mode with minimal performance overhead, enabling reproducible training for reinforcement learning applications.
4. **Improved scheduling and resource allocation:** We develop new CTA scheduling strategies and register allocation schemes tailored to Blackwell's resource constraints and larger tile sizes.

Beyond algorithmic innovations, we implement FlashAttention-4 entirely in CuTe-DSL embedded in Python, achieving 20-30× faster compile times compared to traditional C++ template-

based approaches while maintaining full expressivity. This framework significantly improves developer productivity and lowers the barrier to entry, enabling researchers to rapidly prototype and deploy new attention variants without deep expertise in C++ template metaprogramming.

To empirically validate our method, we benchmark FlashAttention-4 on the B200 GPU and show that (1) BF16 achieves up to 1.3× speedup over cuDNN and 2.7× over the Triton implementation, (2) we achieve near-peak utilization on the shifted bottleneck resources, reaching up to ∼1600 TFLOPS (71% theoretical max), and (3) for large sequence lengths, FlashAttention-4 outperforms alternative attention implementations.

We open source FlashAttention-4 with a permissive license and are working to integrate it with popular libraries to benefit the largest number of researchers and developers. The code is available at `https://github.com/Dao-AILab/flash-attention/tree/main/flash_attn/cute`

2 Background

2.1 Multi-Head Attention

Let $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ be the query, key and value input sequences associated to a single head, where N is the sequence length and d is the head dimension. The attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$ is computed as:

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

where softmax is applied row-wise and $\alpha = 1/\sqrt{d}$ is the scaling factor. In practice, we subtract $\mathrm{rowmax}(\mathbf{S})$ from $\mathbf{S}$ for numerical stability. For multi-head attention (MHA), each head has its own set of projections, and this computation parallelizes across multiple heads and batches.

Given output grad $\mathbf{dO} \in \mathbb{R}^{N \times d}$, the backward computes:

$$\begin{aligned}\mathbf{dV} &= \mathbf{P}^\top \mathbf{dO}, \quad \mathbf{dP} = \mathbf{dO}\mathbf{V}^\top,\\ \mathbf{dS} &= \mathrm{dsoftmax}(\mathbf{dP}),\\ \mathbf{dQ} &= \alpha \mathbf{dS}\mathbf{K}, \quad \mathbf{dK} = \alpha \mathbf{dS}^\top \mathbf{Q},\end{aligned}$$

where $\mathrm{dsoftmax}(\mathbf{dP})$ denotes row-wise softmax gradient $\mathbf{d}s = (\mathrm{diag}(p) - pp^\top)\mathbf{d}p$ for $p = \mathrm{softmax}(s)$.

2.2 GPU Hardware Characteristics and Execution Model

We describe the aspects of the GPU's execution model relevant for FlashAttention-4, with a focus on the NVIDIA Blackwell architecture (B200 & GB200). We highlight key differences from the prior Hopper architecture that motivate the optimizations in FlashAttention-4.

Memory hierarchy: The GPU's memories are organized as a hierarchy of data locales, with capacity inversely related to bandwidth. Global memory (GMEM), also known as HBM, is the off-chip DRAM that is accessible to all streaming multiprocessors (SMs). Data from GMEM are transparently cached in an on-chip L2 cache. Next, each SM contains a small, programmer-managed, highly banked cache called shared memory (SMEM) on the chip. Lastly, there is the register file within each SM.

Blackwell introduces a new memory level called *tensor memory* (TMEM), a 256 KB on-chip memory per SM specifically designed for storing intermediate results of tensor core operations. Unlike shared memory, TMEM is warp-synchronous and tightly coupled with the tensor cores, enabling the matrix multiply-accumulate (MMA) units to write outputs directly to TMEM without

consuming registers. This alleviates the extreme register pressure that plagued Hopper kernels and enables larger tile sizes. TMEM is allocated in 32-column (16 KB) granules and requires explicit programmer management for allocation, deallocation, and data movement.

Thread hierarchy: The GPU's programming model is organized around logical groupings of execution units called threads. From the finest to the coarsest level, the thread hierarchy is comprised of threads, warps (32 threads), warpgroups (4 contiguous warps), threadblocks (i.e. cooperative thread arrays or CTAs), threadblock clusters, and grids. Threads in the same CTA are co-scheduled on the same SM, and CTAs in the same cluster are co-scheduled on the same GPC. SMEM is directly addressable by all threads within a CTA, whereas each thread has at most 256 registers (RMEM) private to itself.

Tensor cores and increased asynchrony: Blackwell features fifth-generation tensor cores that operate on tiles significantly larger than in previous architectures. Each MMA tensor core instruction processes $128 \times N$ tiles (typically $N = 128$ or 256), compared to $64 \times N$ on Hopper. Crucially, Blackwell MMAs write their output directly to TMEM asynchronously, whereas Hopper MMAs write to registers. This full asynchrony enables better overlap between computation and other operations, as the MMA units no longer block on register writeback.

Hardware support for asynchrony allows for warp-specialized kernels, where the warps of a CTA are divided into producer or consumer roles that only ever issue data movement or computation [1].

2-CTA tensor core: Blackwell supports a 2-CTA tensor core MMA mode in which a CTA pair within the same thread block cluster cooperatively executes a single MMA, allowing the operation to read and write tensor memory from both CTAs. One thread in the pair initiates the MMA, but the peer CTA must be launched and remain active while the operation is in flight. Compared to single CTA MMAs that limit the dimension M to 128, the paired mode supports M = 128 or 256 by partitioning the A tile and the accumulator across the pair in dimension M and partitioning the B tile across the two CTAs in dimension N so that each CTA stages only half of B in its own shared memory while the hardware consumes the combined B tile during the multiply. This reduces redundant shared memory capacity and bandwidth, but since these operations touch tensor memory across the CTA pair, kernels must launch CTAs in fixed pairs and use a consistent 2-CTA mode for tensor memory and tensor core operations throughout the kernel.

Shifting bottlenecks: A key trend reflected in Blackwell is that tensor core throughput scales faster than other functional units. Blackwell doubles the FP16/BF16 tensor core throughput compared to Hopper (2.25 PFLOPS [19] vs 1 PFLOPS [17] per GPU), but shared memory bandwidth and exponential unit throughput remain unchanged or scale more slowly. This imbalance shifts the performance bottleneck away from matrix multiplication toward shared memory traffic and non-matmul operations like softmax. As our roofline analysis in Sections 3.1 and 3.2 shows, this requires careful kernel design to maximize overlap between MMA operations and these bottleneck resources.

The throughput of several hardware components on B200 (and GB200) is listed below.

1. Tensor cores: The BF16 MMA has a throughput of 8192 ops / clock / SM, doubled from 4096 ops / clock / SM of Hopper. This can be derived from the theoretical maximum FLOPS: 2.25 PFLOPS / 1850 Mhz clock speed / 148 SMs = 8192 ops / clock / SM.
2. Exponential unit. The multifunction unit (MUFU) on B200 and GB200 can perform 16 ops / clock / SM, the same as Hopper [18]. We note that B300 and GB300 GPUs have doubled the exponential throughput to 32 ops / clock / SM, though these GPUs are not yet widely available at the time of writing.

3. SMEM: the read throughput is 128 bytes / clock / SM, the same as Hopper, as measured by microbenchmarking [15].

We see that the MMA throughput on Blackwell has doubled compared to Hopper, but other hardware units do not necessarily get faster at the same rate. This reflects a broader trend in accelerator design: increasing the throughput of the most important components (typically matrix multiply units) to get higher performance under similar power / silicon area constraint.

3 Algorithm

3.1 Attention forward pass

We first do a roofline analysis to show the bottlenecks of attention forward pass, which motivates our new pipeline design, as well as changes in the FlashAttention algorithm to increase the throughput of the exponential unit and avoid most of the softmax rescaling steps.

3.1.1 Feeds and Speeds

We provide intuition for our kernel design and optimizations by first analyzing the roofline, based on the throughput of the matmul units (tensor cores), shared memory (smem), and exponential unit. We note that this is a simplified analysis that does not consider all resources in the GPU (e.g., floating point math, register bandwidth, L2 bandwidth). Nevertheless, it can identify bottlenecks.

Let the shape of the tile along the length dimension of the sequence of $\mathbf{Q}$ and $\mathbf{K}$ be $M \times N$, and let the head dimension be d. We analyze the compute and memory traffic requirements to identify the performance bottleneck.

MMA compute. The forward pass performs two matrix multiply-accumulate (MMA) operations per iteration: $\mathbf{QK}^\top$ (computing $M \times N$ output from $M \times d$ and $d \times N$ inputs) and $\mathbf{PV}$ (computing $M \times d$ output from $M \times N$ and $N \times d$ inputs). Each MMA requires $2MNd$ floating-point operations. With a tensor core throughput of 8192 FLOPs per cycle, the total compute time is

$$T_{\text{MMA}} = \frac{4MNd}{8192} \text{ cycles.} \tag{1}$$

Shared memory traffic. Of the two MMAs, one is shared-shared (SS) where both operands are read from shared memory ($\mathbf{QK}^\top$), while the other is tensor-shared (TS) where operand A is read from tensor memory and operand B from shared memory ($\mathbf{PV}$). Since each MMA instruction operates on tiles of size 128×128, computing an $M \times N$ output requires $\lceil M/128 \rceil \times \lceil N/128 \rceil$ MMA instructions. Crucially, when multiple MMA instructions are needed, the shared memory operands are read multiple times.

For $\mathbf{QK}^\top$ (SS), computing the $M \times N$ output requires $\lceil M/128 \rceil \times \lceil N/128 \rceil$ MMA instructions, each reading a $128 \times d$ chunk of $\mathbf{Q}$ and a $d \times 128$ chunk of $\mathbf{K}^\top$ from shared memory. The total shared memory reads are $\lceil M/128 \rceil \times \lceil N/128 \rceil \times (128d + 128d) = \lceil M/128 \rceil \lceil N/128 \rceil \times 256d$ elements. For $\mathbf{PV}$ (TS), computing the $M \times d$ output requires $\lceil M/128 \rceil \times \lceil d/128 \rceil$ MMA instructions, each reading a $N \times 128$ chunk of $\mathbf{V}$ from shared memory, totaling $\lceil M/128 \rceil \times \lceil d/128 \rceil \times 128N$ elements. At 2 bytes per element (bf16) and 128 bytes per cycle bandwidth, the shared memory (T_{smem}) read time is

$$= 2\left\lceil \tfrac{M}{128} \right\rceil \left\lceil \tfrac{N}{128} \right\rceil 256d + 2\left\lceil \tfrac{M}{128} \right\rceil \left\lceil \tfrac{d}{128} \right\rceil 128N = \tfrac{3MNd}{8192} \text{ cycles} \tag{2}$$

(assuming M, N, d are multiples of 128).

Exponential unit. The exponential unit computes elementwise operations required for the softmax computation. The forward pass requires exponential operations on $M \times N$ values (corresponding to the attention matrix $\mathbf{S}$). With a throughput of 16 operations per cycle, the exponential unit requires

$$T_{\text{exp}} = \frac{MN}{16} \text{ cycles.} \tag{3}$$

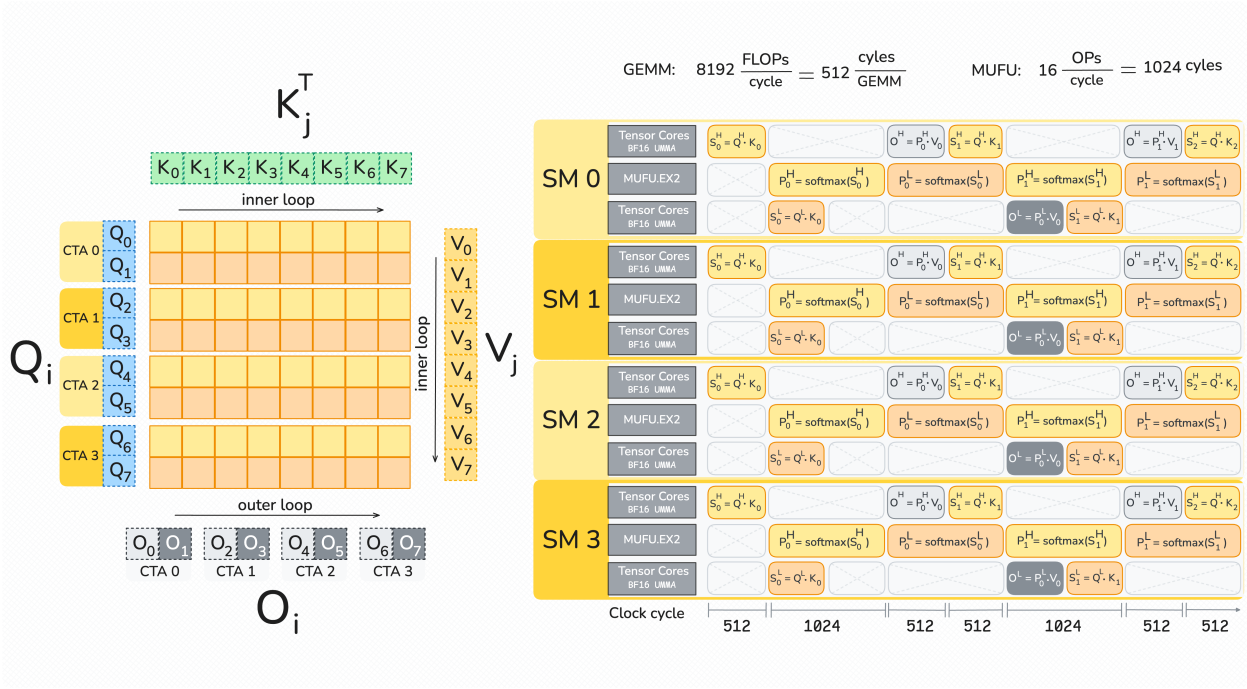

Figure 1: FlashAttention-4 forward pipeline. The superscript ^{H} denotes the matrices corresponding to the "high" Q tile, and superscript ^{L} denotes matrices corresponding to the "low" Q tile. Each Q tile corresponds to 128 query tokens.

Resource	128^3	256×128^2
MMA compute	**1024**	**2048**
Shared memory	768	1536
Exponential unit	**1024**	**2048**

Table 1: Roofline analysis (cycles) for the attention forward pass. For both tile sizes, MMA compute and exponential unit are the primary bottlenecks.

Table 1 summarizes the analysis for two typical tile configurations. For $M = N = d = 128$, the resources are well-balanced, with shared memory (768 cycles) being slightly lower than both MMA compute and exponential unit (both 1024 cycles). For the larger tile size $M = 256, N = d = 128$, the shared memory traffic increases to 1536 cycles due to reading MMA operands multiple times, while MMA compute and exponential unit double to 2048 cycles. This analysis motivates our kernel design to (1) use large tile sizes and maximize overlap between MMA operations and softmax computations (2) increase the throughput of exponential by using other hardware units (3) reduce the time of unnecessary non-matmul operations.

3.1.2 New pipeline to overlap matmul and softmax

Since the Blackwell architecture doubled the tensor core flops again, taking care to overlap softmax and tensor core operations is even more crucial than on Hopper. We follow a ping-pong schedule similar to FA-3, where two tiles of the output are computed per thread block. While one tile's tensor core operations are executed, the other tile computes softmax. While Hopper tensor cores hold the accumulator in registers, with four threads per row in an interleaved pattern, Blackwell tensor cores hold their accumulators in tensor memory. Additionally, a single accumulator tile on

Blackwell is 128 by 128 elements large, where Hopper's tile size was 64 by 128.

The natural way to distribute work across these tiles is then to have two warpgroups of 128 threads each, with each thread processing an entire row. This eliminates the need for inter-warp shuffles to reduce the row max, and for multiple statistics registers per thread. Just like with FA-3, we explicitly synchronize the two softmax warpgroups to not overlap in their critical section, which is the part of exponential computation. Each softmax warpgroup proceeds by first loading the entire row into registers, then computing the maximum, then computing the softmax (i.e., subtract the max, rescale, exponentiate, convert to input precision), and finally computing the row sum.

Another difference from FA-3 is that since we transfer **P** via tensor memory rather than register file, we can decouple the rescaling of the output to a separate "correction" warpgroup and thus take it out of the critical path.

Several tensor memory partitionings are possible to achieve this pipeline overlap. All must allocate two tiles worth of output, leaving (at head dimension 128) half the tensor memory to store **S** and **P**. That memory can store two copies of **S** or four copies of **P** (assuming the input of the FP16 or BF16 tensor core). This leaves us with roughly two partitioning options for the remaining tensor memory: one tile of **S** and two tiles of **P**, or two tiles of **S** that overlap with **P**. We choose the latter because it allows us to start our software pipeline by immediately computing two **S** tiles. It also leaves some tensor memory to communicate rescale statistics to the correction warpgroup.

One issue of the larger Blackwell tile sizes and the chosen thread assignment is that, unless we re-load from tensor memory, we must hold an entire row of 128 elements in register. Given that we use two softmax warpgroups, one correction warpgroup, and one warpgroup to drive tensor cores and TMA units, assigning sufficient registers to softmax and preventing register spills is critical. For BF16 input data types, we need to hold 128 registers for the input, and potentially 64 registers for the output (plus miscellaneous and temporary registers). To reduce register pressure, we stage out storing **P**: The first three quarters are stored once (and trigger the corresponding MMA operations), and the last quarter is stored separately.

3.1.3 Emulation of the exponential function

Exponential throughput bottleneck. On modern GPUs, the exponential function is computed by the multi-function unit (MUFU), which has significantly lower throughput than the tensor cores used for matrix multiplication. On B200 and GB200 GPUs, MUFU provides 16 operations per clock per SM, compared to 8192 operations per clock per SM for matrix multiplication. Since softmax computation requires many exponential evaluations, this disparity makes the exponential function a critical bottleneck in attention kernels.

Software emulation via polynomial approximation. To increase exponential throughput, we implement a software emulation of 2^x using floating-point FMA units, which can operate in parallel with MUFU. We use the classical range reduction technique (Cody-Waite) and then the polynomial approximation [16]. The key insight is to decompose the exponential computation:

$$2^x = 2^{\lfloor x \rfloor} \, 2^{x - \lfloor x \rfloor} \tag{4}$$

where $\lfloor x \rfloor$ is the integer part and $x - \lfloor x \rfloor \in [0, 1)$ is the fractional part.

The integer part $2^{\lfloor x \rfloor}$ can be computed efficiently using bit manipulation of the IEEE 754 floating-point representation. Since the exponent field directly represents powers of two, computing $2^{\lfloor x \rfloor}$ amounts to a shift and add operation on the exponent bits, which can be done using integer ALU instructions.

For the fractional part, we approximate $2^{x_{\text{frac}}}$ where $x_{\text{frac}} \in [0, 1)$ using a polynomial:

$$2^{x_{\text{frac}}} \approx \sum_{i=0}^{n} p_i \, x_{\text{frac}}^i \tag{5}$$

	FP32 vs FP64		BF16 vs FP64	
Method	Max rel err	Mean rel err	Max rel err	Mean rel err
Ideal (FP64→BF16)	—	—	3.89×10^{-3}	1.41×10^{-3}
Hardware `MUFU.EX2`	1.41×10^{-7}	3.04×10^{-8}	3.89×10^{-3}	1.41×10^{-3}
Degree 3	8.77×10^{-5}	5.43×10^{-5}	3.90×10^{-3}	1.41×10^{-3}
Degree 4	3.05×10^{-6}	1.84×10^{-6}	3.89×10^{-3}	1.41×10^{-3}
Degree 5	1.44×10^{-7}	5.48×10^{-8}	3.89×10^{-3}	1.41×10^{-3}

Table 2: Accuracy of 2^x polynomial emulation on $[0, 1)$, measured against FP64 reference on 4M random inputs. FP32 columns measure the raw polynomial output; BF16 columns measure after rounding to BF16. The BF16 quantization error dominates for all degrees ≥ 3.

with $p_0 = 1.0$ and the remaining coefficients chosen to minimize the relative approximation error over $[0, 1)$, calculated using the Sollya software package [4]. The polynomial evaluation uses Horner's method with FMA instructions, achieving high throughput.

The complete algorithm proceeds as follows:

1. Clamp x to be at least -127 to avoid underflow
2. Compute $\lfloor x \rfloor$ using round-down mode: add $2^{23} + 2^{22}$ to x (forcing the fractional bits into the mantissa), then subtract it back with round-down mode
3. Compute fractional part: $x_{\text{frac}} = x - \lfloor x \rfloor$
4. Evaluate polynomial to get $2^{x_{\text{frac}}}$
5. Combine integer and fractional parts: shift $\lfloor x \rfloor$ into the exponent field and add the mantissa bits of $2^{x_{\text{frac}}}$

By distributing exponential computations across both MUFU and FMA units, this approach effectively increases the exponential throughput, alleviating a key bottleneck in attention computation.

Partial emulation. Although polynomial emulation increases exponential throughput, it comes at a cost: additional registers (to hold intermediate values and coefficients), higher register bandwidth consumption, and longer latency compared to the MUFU instruction. Using emulation for all exponential evaluations would increase register pressure and could cause spills that negate the throughput benefit. Instead, we apply emulation to only a subset of the entries in each softmax row (10–25%), with the remaining entries computed via hardware `MUFU.EX2`. The exact fraction is tuned empirically based on the ratio of MMA and exponential throughput for a given tile configuration.

Numerical accuracy. Table 2 compares the accuracy of polynomial approximations of different degrees against the hardware `MUFU.EX2` instruction, measured on 4M random inputs in $[0, 1)$. We report two metrics: the FP32-level error (before any quantization) and the BF16-level error (after rounding the FP32 output to BF16), both measured against a FP64 reference.

At the FP32 level, the degree-3 polynomial has a maximum relative error of 8.8×10^{-5}, roughly $600\times$ higher than hardware. However, after rounding to BF16, the errors become nearly indistinguishable: the quantization error of BF16 ($\sim 3.9 \times 10^{-3}$) dominates the polynomial approximation error for all degrees ≥ 3. The degree-3 polynomial matches hardware to within 1 BF16 ULP on 99% of inputs, which is sufficient for attention computation where the softmax output is consumed with BF16 precision. Higher-degree polynomials close the FP32 gap: degree 5 matches hardware to within $2\times$ in maximum relative error, at the cost of two additional FMA instructions per evaluation.

3.1.4 Skipping online softmax rescaling

FlashAttention online softmax. FlashAttention computes attention $\text{softmax}(QK^\top)V$ in blocks to minimize memory traffic. For numerical stability, the algorithm maintains running statistics as it processes blocks. When computing block j, let $S_j = QK_j^\top$ be the attention scores for that block. The online softmax algorithm tracks:

$$\begin{aligned} m_j &= \max(m_{j-1}, \text{rowmax}(S_j)) \\ \ell_j &= e^{m_{j-1}-m_j}\ell_{j-1} + \text{rowsum}(e^{S_j - m_j}) \end{aligned}$$

where m_j is the running max and ℓ_j is the running sum of exponentials (normalizer). The intermediate output O_j is updated as: $O_j = e^{m_{j-1}-m_j}O_{j-1} + e^{S_j-m_j}V_j$. The rescaling factor $e^{m_{j-1}-m_j}$ ensures numerical stability by renormalizing previous results when larger values are encountered.

Conditional rescaling. The step $e^{m_{j-1}-m_j}O_{j-1}$ requires a vector multiplication. We make two simple observations:

1. Rescaling is only necessary when $m_j > m_{j-1}$, i.e., when new larger values are found.
2. We can tolerate some "slack" in the rescaling: only rescale when $m_j - m_{j-1} > \tau$, where τ is a threshold (typically set to $\log_2(256) = 8.0$, corresponding to a rescaling factor of 256.0). As long as we keep track of the statistics (the total scaling we have done), we can still get the true denominator at the end to get the right final output.

In FlashAttention-4, we modify the algorithm as:

$$O_j = \begin{cases} e^{m_{j-1}-m_j}O_{j-1} + e^{S_j-m_j}V_j & \text{if } m_j - m_{j-1} > \tau \\ O_{j-1} + e^{S_j-m_{j-1}}V_j & \text{otherwise} \end{cases} \tag{6}$$

When $m_j - m_{j-1} \le \tau$, we skip updating m and continue using m_{j-1}. This maintains the correctness because at the end of the computation, all accumulated values are renormalized by the true maximum m_{final} and the final normalizer ℓ_{final}:

$$\text{Output} = \frac{1}{\ell_{\text{final}}}O_{\text{final}}$$

This modification significantly reduces the number of rescaling operations while maintaining numerical accuracy, as the final normalization step corrects any small deviations introduced by skipping intermediate rescaling.

In practice, to avoid warp divergence, we rescale when any of the threads in the warp needs rescaling.

3.2 Attention backward pass

3.2.1 Feeds and Speeds

Similar to the forward pass, we first provide intuition for our kernel design and optimizations by analyzing the roofline, based on the throughput of the matmul units (tensor cores), shared memory (smem), and exponential unit.

Let the tile shape along the sequence length dimension of **Q** and **K** be $M \times N$, and let the head dimension be d. We analyze the compute and memory traffic requirements to identify the performance bottleneck. Unlike for the forward pass, we make the assumption that $M = N = d = 128$ to simplify the formulas for the smem cycle count, although we retain variable names for clarity.

MMA compute. The backward pass performs five matrix multiply-accumulate (MMA) operations per iteration. Each MMA involves an $M \times N$ matrix, an $M \times d$ matrix, and an $d \times N$ matrix (varying which one is the output matrix), requiring $2MNd$ floating-point operations. With a tensor

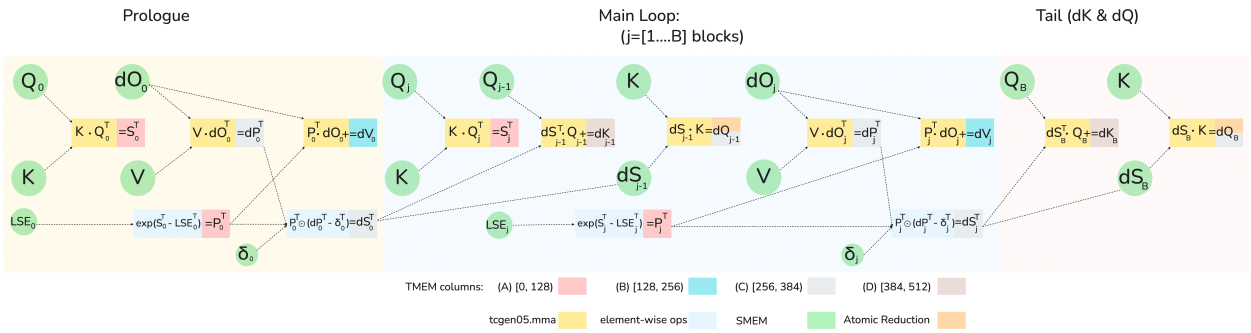

Figure 2: FlashAttention-4 backward computation graph (5 MMA operations + 2 elementwise operations), showing the 1-CTA MMA mode software pipeline order across the prologue, main loop, and tail.

core throughput of 8192 FLOPs per cycle, the total compute time is

$$T_{\text{MMA}} = \frac{10MNd}{8192} \text{ cycles.} \tag{7}$$

Shared memory traffic. Of the five MMAs, three of them – $\mathbf{S}^\top = \mathbf{K}\mathbf{Q}^\top$, $\mathbf{dP}^\top = \mathbf{V}\mathbf{dO}^\top$, and $\mathbf{dQ} = \mathbf{dS}\mathbf{K}$ – are shared-shared (SS) operations where both operands are read from shared memory, while two – $\mathbf{dV} = \mathbf{P}^\top\mathbf{dO}$ and $\mathbf{dK} = \mathbf{dS}^\top\mathbf{Q}$ – are tensor-shared (TS) operations where operand A is read from tensor memory and operand B from shared memory. The SS MMAs read in total $2Md + 3Nd + MN$ elements from shared memory, while the TS MMAs read in total $2Md$ elements from shared memory. At a shared memory bandwidth of 128 bytes per cycle and with each element being 2 bytes (bf16), this contributes

$$T_{\text{smem,MMA}} = \frac{4Md + 3Nd + MN}{64} \text{ cycles.} \tag{8}$$

Additionally, the algorithm writes the intermediate gradient $\mathbf{dS}$ of size $M \times N$ to shared memory in bf16, requiring $2MN$ bytes or $MN/64$ cycles. The gradient $\mathbf{dQ}$ of size $M \times d$ is written in fp32 (4 bytes per element) to shared memory, then read back via TMA for the reduction, totaling $8Md$ bytes of shared memory traffic or $Md/16$ cycles.

The total shared memory access time (T_{smem}) is therefore

$$\frac{4Md + 3Nd + MN}{64} + \frac{MN}{64} + \frac{Md}{16} \text{ cycles.} \tag{9}$$

Exponential unit. The exponential unit computes elementwise operations (exponentials, logarithms, and related nonlinear functions) required for the softmax and its gradient. The backward pass requires exponential operations on the $M \times N$ values (corresponding to the attention matrix $\mathbf{S}$ and related terms). With a throughput of 16 operations per cycle, the exponential unit requires

$$T_{\text{exp}} = \frac{MN}{16} \text{ cycles.} \tag{10}$$

Table 3 summarizes the analysis for the typical tile configuration $M = N = d = 128$. The shared memory traffic time of 3328 cycles exceeds both the MMA compute time (2560 cycles) and the exponential unit time (1024 cycles), indicating that shared memory bandwidth is the primary bottleneck, though less severely than global memory traffic would suggest. This motivates our kernel design to maximize overlap between MMA operations and other computations to hide shared memory latency.

3.2.2 New pipeline to overlap matmul and softmax

The backward pass in flash attention performs five MMA operations, corresponding to recomputing $\mathbf{S}$, and the two gradient computations induced by $\mathbf{QK}$ (which yields $\mathbf{dQ}$ and $\mathbf{dK}$) and $\mathbf{PV}$ (which

Resource	Cycles ($N = d = 128$) **1-CTA** ($M = 128$)	Cycles ($N = d = 128$) **2-CTA** ($M = 256$)
MMA compute	2560	2560
Shared memory (MMA operands)	2048	1536
Shared memory (**dS** write)	256	256
Shared memory (**dS** *DSMEM*)	0	384
Shared memory (**dQ** write + read)	1024	512
Total shared memory	**3328**	**2688**
Exponential unit	1024	1024

Table 3: Roofline analysis for the attention backward pass with $M = N = d = 128$. Shared memory traffic is the bottleneck, exceeding MMA compute time by approximately 30%. In the 2-CTA setting with $M = 256$ and $N = d = 128$ (with an exception to the **dQ** mma, with $M = N = 128$ and $d = 256$), shared memory traffic exceeds MMA compute time by approximately 5%.

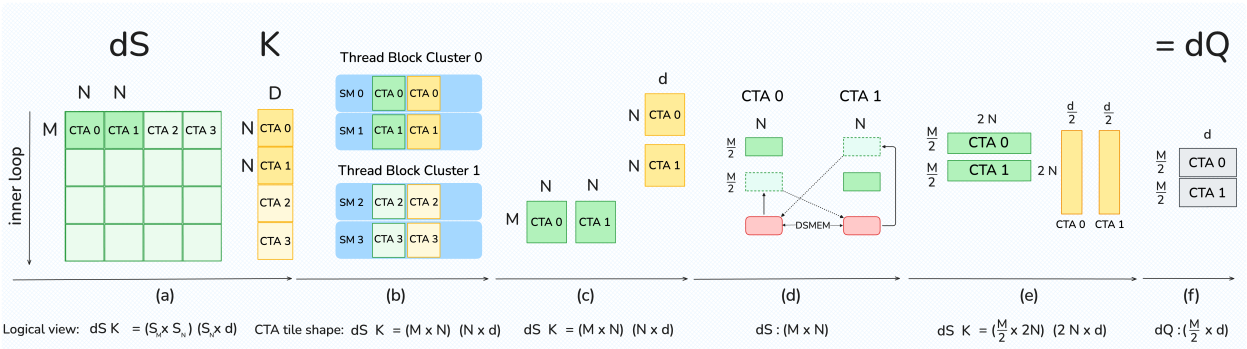

Figure 3: In the 2-CTA backward dQ step, the CTA pair uses DSMEM to exchange half of the dS tile so each CTA forms an $(\frac{M}{2} \times 2N)$ operand and can run a CTA-pair UMMA with a doubled reduction.

yields **dP** and **dV**) respectively. In FA-3, accumulators are stored in registers and registers are a limited resource. This imposes significant ordering constraints that—in effect—serialize the compute graph, i.e., compute $\mathbf{S}, \mathbf{dP}, \mathbf{dV}, \mathbf{dQ}, \mathbf{dK}$, with only the TMA load running significantly out of turn. In addition to that, the algorithm is similar: It iterates along the KV sequence length dimension and computes values transposed w.r.t. the forward pass, since that is the layout that the **dV** and **dK** gradient calculations require to read one of their operands from tensor memory. **dQ** accumulates through the atomics.

In FA-4, TMEM enables additional schedules compared to FA-3 that provide significant overlap between MMA and non-MMA operations. Specifically, just as with the forward pass, we are trying to hide the latency of the softmax calculation. In FA-3, the softmax computation overlaps with the MMA for **dP**. From the previous section, we know that on Blackwell we need at least two MMA operations to run concurrently.

We achieve this by using the **dQ** and **dK** MMA's of the previous iteration. This requires careful management of shared memory and tensor memory resources between loading, MMA, compute, and reduction operations. In particular, note that we do not have sufficient tensor memory to fit five accumulator tiles. At most, four tiles of 128 by 128 elements can fit and **dV** and **dK** accumulate, and thus cannot share their space. In our implementation, we have **S** and **P** share one of the tmem blocks (at offset 0) and we have **dP**, **dS** and **dQ** share the other one. We demonstrate the computational graph of the FA-4 bwd in Figure 2.

3.2.3 2-CTA backward pass: Reducing shared memory traffic and global atomic adds

Even with improved pipelining and with two of the ten GEMM operands resident in tensor memory, shared memory bandwidth still dominates the backward pass. Across the five GEMMs, the remaining eight BF16 operands are loaded from shared memory to feed the tensor cores, and this shared memory traffic incurs about 30% more cycles than the tensor core compute. To further mitigate this bottleneck, we use the 2-CTA MMA mode introduced by Blackwell, in which the output accumulator is partitioned in the M dimension. With an MMA tile shape of $M = 256$ and $N = K = 128$, the two CTAs act as a single larger tile: each CTA loads and stages half of operand B and keeps only its own accumulator slice.

Shared memory traffic. With five GEMMs in the backward pass, we use MMA tile shape of $M = 256$ and $N = K = 128$, which roughly halves the shared memory traffic for operand B. In the FlashAttention backward pass, each CTA holds a fixed KV tile (parallelize the outer loop across N CTAs) and streams over M tiles in the inner loop. The accumulation of **dQ** is a reduction in the KV sequence in the outer loop, but the 2-CTA MMA only splits the output tile, not the reduction axis, and the reduction dimension of the **dQ** MMA is N, which is naturally split across the CTA pair. As a result, each CTA still needs the full reduction for the rows it owns. To address this conflict on the reduction axis, we use distributed shared memory (DSMEM) to exchange half of the dS between the two CTAs since they are in the same cluster. This approach repacks **dS** so that it is partitioned along the non-reduction axis, with each CTA owning its $\frac{M}{2}$ rows and holding the full $2N$ reduction. As a result, the per CTA **dQ** MMA tile shape is $(\frac{M}{2}, 2N)(2N, d)$ and accumulates a tile $(\frac{M}{2}, d)$ in tensor memory. In 2-CTA MMA mode, the MMAs for **S**, **dP**, **dV** and **dK** run with the tile $M = 256$, while **dQ** uses $M = 128$ but with double reduction $2N = 256$. We then reordered the software pipeline with respect to the 1-CTA variant to hide the DSMEM latency. We compute **dP** for the current tile before computing **dQ** for the tile of the previous iteration. The **dQ** tile is small enough to fit in TMEM alongside **P**, reusing the same TMEM region as **S**, so we no longer reuse the same TMEM region for **dP** and **dQ** as we do in the 1-CTA mode. With this new pipelining ordering, we can compute the element-wise **dS** of the current tile in parallel with the **dQ** MMA of the tile from the previous iteration. See Figure 3 for an illustration of how the **dQ** step is decomposed.

dQ atomic adds. A complementary benefit of this **dQ** decomposition is that it halves the number of global atomic reductions. Atomic updates introduce nondeterminism and are expensive as they occur in every iteration of the inner loop. Consequently, each CTA writes only half of the **dQ** tile and performs half as many global atomic reductions as the 1-CTA counterpart.

3.2.4 Deterministic backward pass

Our backward kernel introduces nondeterminism for the gradient computation due to inter-CTA reductions in global memory (affecting **dQ** in general, and **dK**/**dV** in the case of GQA). To ensure reproducibility and facilitate reliable debugging during training, we also provide a deterministic execution mode. The standard solution, which we also adopt, is to serialize the global reductions using a semaphore lock. Specifically, each CTA writing to a common **dQ** tile must acquire the lock according to a predefined order, perform its reduction, and then release the lock by incrementing the semaphore counter.

This lock-based approach impacts performance for two main reasons: (1) issuing the memory fence to ensure device-wide visibility of the semaphore write (required for correct acquire–release semantics), and (2) introducing stalls as each CTA waits for previous CTAs reducing on a common **dQ** tile to complete. In load-imbalanced situations, a naive choice of CTA order can severely degrade performance. In general, we do CTA swizzling in the head and batch dimensions to reduce stalls

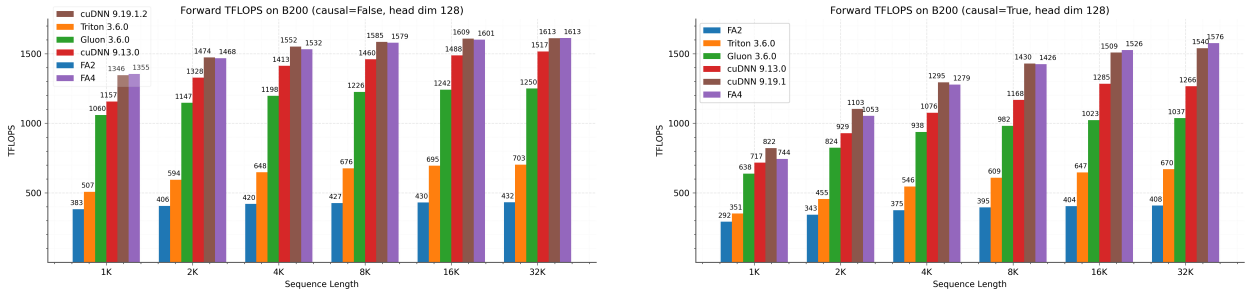

Figure 4: Forward pass TFLOPS on B200 (FP16/BF16) with head dimension 128. Left: non-causal attention. Right: causal attention. FA4 achieves 1.1-1.3× speedup over cuDNN 9.13.0 and 2.1-2.7× over Triton across sequence lengths. Since the release of our implementation, newer versions of cuDNN have incorporated many of the techniques described in this paper, yielding similar performance to FA4.

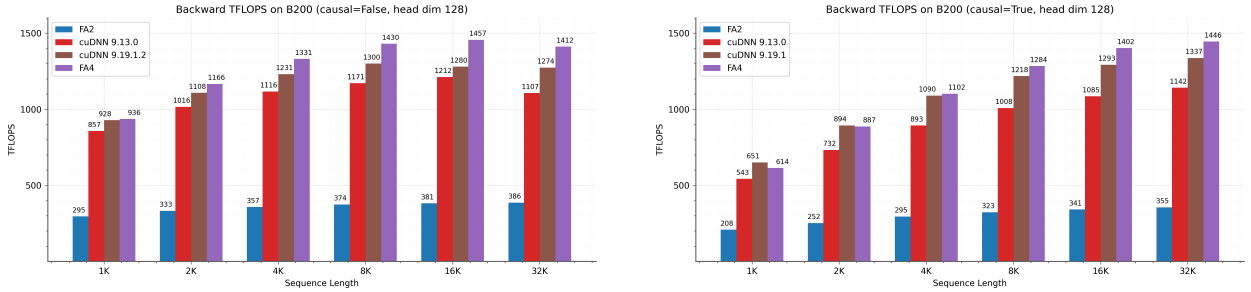

Figure 5: Backward pass TFLOPS on B200 (FP16/BF16) with head dimension 128. Left: non-causal attention. Right: causal attention.

(up to L2 cache capacity, cf. Section 3.3). For causal masking, we additionally launch KV blocks in descending order, traverse query blocks in ascending order starting from the diagonal, and order the **dQ** reductions by descending query block index. This "shortest-processing-time-first" (SPT) schedule ensures that no CTA is stalled on its first **dQ** write.

3.3 Scheduling

In many situations, such as with causal masking or variable sequence length (varlen), the attention kernel is naturally load-imbalanced – SMs are assigned worktiles whose mainloops differ in length, since some worktiles require more loads and MMAs than others. Furthermore, we can choose the order in which SMs process tiles, for instance by defining a preferred linearization of the grid coordinates. Abstracting away any specific features of attention, we can then apply general results on makespan minimization for identical parallel processors to our context. In particular, in FlashAttention-4, we use the classical idea of longest-processing-time-first (LPT) scheduling [9]. We emphasize that the way in which we apply this idea works across all GPU architectures and was also validated as an improvement to FlashAttention-3 on Hopper GPUs.

LPT for causal masking. The standard attention grid is given by (mblocks, heads, batches) and is computed in increasing order left-to-right. But scores are masked out above the diagonal, so for fixed head and batch, SMs end up inefficiently processing worktiles from shortest to longest. On the other hand, a naive LPT order is also suboptimal, since for different batches, mainloop KV loads won't hit in L2 cache, and loading all KV heads first can thrash the L2 cache if they exceed its capacity. Instead, we always process batches as the outermost dimension, and swizzle over heads. This means that we divide heads into sections that don't overflow L2 cache; the tile scheduler then traverses the grid by heads per section, mblocks in reverse order, sections, and finally batches. In

particular, for MQA or GQA, we always traverse all query heads per KV head before varying over mblocks. Empirically, we validate that this LPT order is highly effective; for example, for BF16 and head dimension 128 we obtain 4-8% FLOPS gain for MHA and 7-14% for MQA 8 as measured on an H200 GPU.

LPT for variable sequence length. For varlen, we also have to contend with load imbalance due to variation among batches. For instance, in decode workloads, different batches might attend to different amounts of context, and in mixed or continuous batching, some batches might be prefill while others are decode. The list of query and KV sequence lengths per batch is typically stored as attention metadata on device, and the standard varlen attention kernel reads these integers at runtime while processing batches in increasing order. However, the given batch order may be arbitrarily suboptimal with respect to load balancing – for example, we could have shorter square prefills followed by long-context decodes. To ameliorate this, we can enforce an LPT order by launching a preprocessing kernel to sort batches according to their maximum per-worktile execution time, writing out the additional metadata of a virtual to actual batch index mapping that will be subsequently read back into the attention kernel in order to traverse batches in sorted order. This metadata can be cached and thus results in no performance loss from sorting.

4 Language and Framework

We write FlashAttention-4 entirely in CuTe-DSL [21], embedded in Python, without any component in CUDA C++. The CuTe-DSL compiler takes the source code in Python, lowers to PTX, then uses the PTX compiler (ptxas) to finally produce the assembly code (SASS).

Full expressivity with clean abstractions. The CuTe-DSL programming model is isomorphic to CUTLASS C++, ensuring that FlashAttention-4 retains the full expressivity of low-level GPU programming while benefiting from the productivity gains of meta-programming in Python instead of C++ and fast JIT compilation times. CuTe-DSL provides direct access to PTX as an escape hatch, allowing developers to implement any functionality they need without framework limitations. For example, we leverage custom PTX sequences for operations not yet fully exposed in CuTe-DSL APIs (though these will be integrated in future releases), demonstrating that our framework does not constrain developers to a limited subset of GPU capabilities.

Fast compilation through JIT. Compile time has been a bottleneck in past FlashAttention implementations, due to complex C++ template metaprograms. By embedding CuTe-DSL in Python with just-in-time (JIT) compilation, FlashAttention-4 achieves faster build times compared to traditional C++ template-based approaches. As shown in Table 4, FlashAttention-4 reduces compile time by 20-30× compared to FlashAttention-3. This rapid iteration cycle significantly improves developer productivity, enabling faster experimentation and debugging during kernel development.

Method	Forward pass	Backward pass
FlashAttention-3	55s	45s
FlashAttention-4	2.5s	1.4s
Speedup	22×	32×

Table 4: Compile time for a single kernel: FA3 (C++ templates) and FA4 (CuTe-DSL). Typically FA2 and FA3 require precompiling hundreds of kernels for different attention variants.

Flexibility and accessibility. The Python based framework has already demonstrated its flexibility in practice: developers have successfully built FlexAttention and block-sparse attention

variants on top of FlashAttention-4 without modifying the core framework. By lowering the barrier to entry, our approach enables researchers and engineers with just a few months of GPU programming experience to contribute meaningful extensions without requiring deep expertise in C++ template metaprogramming. This accessibility accelerates innovation and allows the attention mechanism research community to more rapidly explore new algorithmic variants.

Our vision is to provide a comprehensive framework for building all kinds of attention variants with best-in-class performance. Rather than implementing each attention variant from scratch, FlashAttention-4 factors common functionality into independent, composable primitives. Operations such as block-sparse patterns, masking strategies, variable sequence length handling, and work scheduling are all exposed as orthogonal primitives that can be freely combined. This modular design ensures that optimizations and new features benefit all attention implementations built on the framework while still achieving the highest performance by compiling down to efficient GPU kernels.

5 Empirical Evaluation

We evaluate FlashAttention-4 efficiency compared to various open-source and closed-source baselines.

Benchmarking attention. We measure the runtime of FlashAttention-4 across different sequence lengths and head dimensions, comparing it to standard implementations in PyTorch, FlashAttention-2[1], Triton (which uses B200-specific instructions [26]), Gluon (a lower-level GPU programming language with finer control than Triton [25]), and cuDNN (a vendor's library optimized for B200 GPUs). We confirm that FlashAttention-4 is up to 1.3× faster than cuDNN 9.13 and up to 2.7× faster than Triton. FlashAttention-4 reaches up to 1613 TFLOPs/s, approximately 71% of the theoretical maximum TFLOPs/s on B200 GPUs.

Benchmark settings. We measure runtime on a B200 GPU for different settings (with/without causal mask, head dimensions 64, 128, and (192, 128)) for BF16 inputs. We vary the sequence length as 1k, 2k, ..., 32k, and set batch size so that the total number of tokens is 32k. We set the hidden dimension to 2048, and head dimension to be either 64 or 128 (i.e., 32 heads or 16 heads). For the (192, 128) configuration used in DeepSeek V3 [7], we use 16 heads with 192 query dimensions and 128 key/value dimensions. To calculate the FLOPs of the forward pass, we use $4 \cdot \text{seqlen}^2 \cdot \text{head dimension} \cdot \text{number of heads}$. With causal masking, we divide this number by 2 to account for the fact that approximately only half of the entries are calculated. To get the FLOPs of the backward pass, we multiply the forward pass FLOPs by 2.5 (since there are 2 matmuls in the forward pass and 5 matmuls in the backward pass, due to recomputation).

5.1 Forward pass

We report forward pass results in Figs. 4 and 6, showing that FlashAttention-4 is 1.1-1.3× faster than cuDNN 9.13 and 2.1-2.7× faster than Triton. For medium and long sequences (4k and above), FlashAttention-4 consistently outperforms all baselines across different head dimensions and causal masking settings. The gains are larger for the causal case, which we attribute to the longest-processing-time first (LPT) scheduler.

5.2 Backward pass

We report backward pass results in Fig. 5. FlashAttention-4 achieves consistent speedups across long sequence lengths and causal masking, demonstrating the effectiveness of our 2-CTA backward pass.

[1] FlashAttention-3 does not run on B200

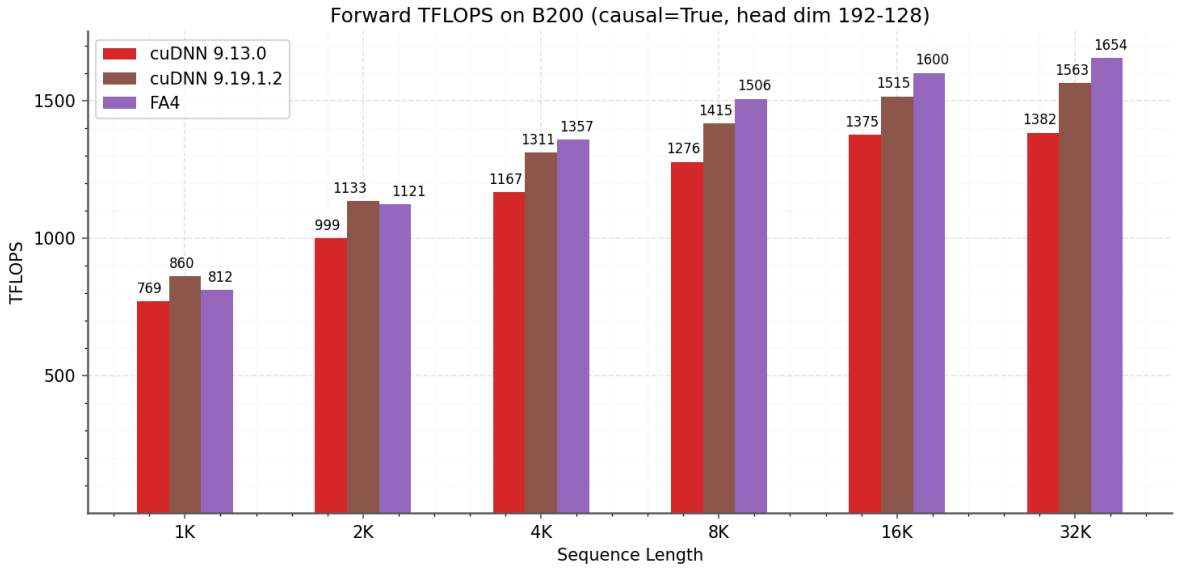

Figure 6: Forward pass TFLOPS comparison between cuDNN and FA4 on B200 (FP16/BF16) with head dimension (192, 128) for causal attention (typically used in DeepSeek V3 architecture)

We also show the performance of the deterministic backward pass in Fig. 7. Our careful swizzling and scheduling results in a much faster deterministic backward pass, getting up to 75% the speed of the nondeterministic backward pass of the 1-CTA backward pass.

6 Discussion and Conclusion

FlashAttention-4 addresses asymmetric hardware scaling, where tensor cores are so fast that the dominant bottlenecks shift to shared-memory traffic and exponential throughput, motivating algorithmic and kernel co-design to mitigate these limits. We redesign the pipeline around fully asynchronous MMA to overlap softmax with larger-tiled matmuls and introduce software-emulated exponential and conditional softmax rescaling to reduce non-matmul operations. We leverage tensor memory and 2-CTA MMA mode to reduce shared memory traffic. In addition, 2-CTA enables restructuring the global atomic accumulation, halving the number of global atomic adds. FlashAttention-4 is implemented entirely in CuTe-DSL embedded in Python, preserving low-level control while achieving 20-30× faster compile times than C++ template-based kernels. Although optimized for Blackwell GPUs, some of these algorithms can be extended to other accelerators as compute continues to outpace non-matmul units.

Acknowledgments

We thank Together AI, Meta, xAI, and Princeton Language and Intelligence (PLI) for compute support. We gratefully acknowledge the support of the Schmidt Sciences AI2050 fellowship, the Google ML and Systems Junior Faculty Awards, and the Google Research Scholar program. We want to further thank the following teams at Nvidia: CuDNN, TensorRT-LLM, and Cutlass teams for constant discussions, ideas, and feedback.

References

[1] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011.

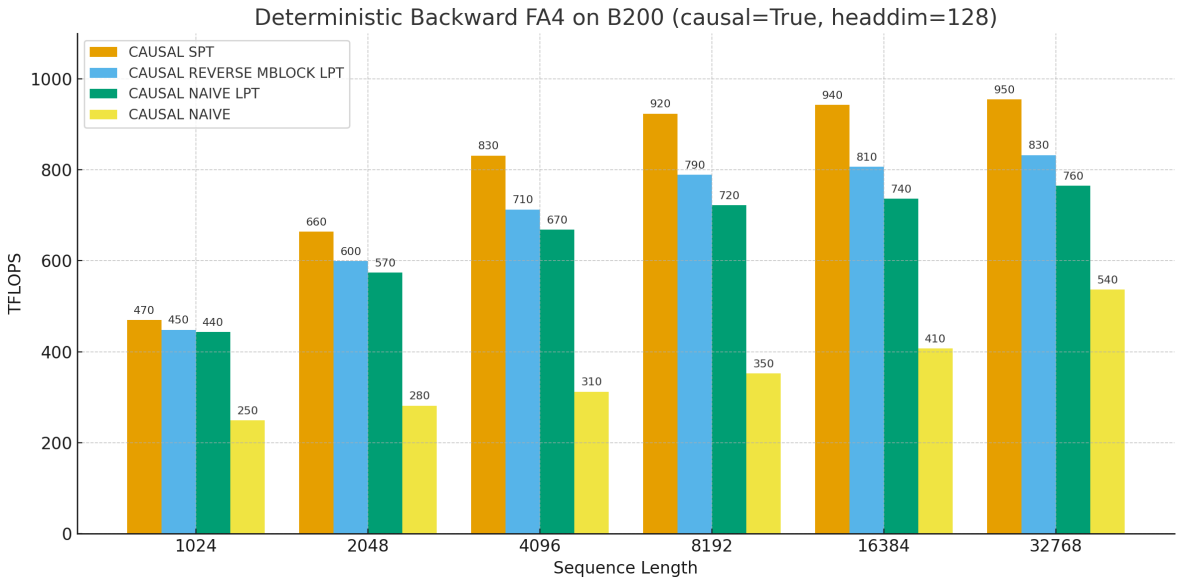

Figure 7: Ablations for Deterministic Backward pass on B200 (FP16/BF16) with head dimension 128. Causal attention – SPT, LPT with reverse mblock order, LPT, and naive with no batch/head swizzle.

Association for Computing Machinery. ISBN 9781450307710. doi: 10.1145/2063384.2063400. URL `https://doi.org/10.1145/2063384.2063400`.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] Richard J Chen, Chengkuan Chen, Yicong Li, Tiffany Y Chen, Andrew D Trister, Rahul G Krishnan, and Faisal Mahmood. Scaling vision transformers to gigapixel images via hierarchical self-supervised learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16144–16155, 2022.

[4] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In *Mathematical Software – ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31. Springer Berlin Heidelberg, 2010.

[5] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, 2023. URL `https://arxiv.org/abs/2307.08691`.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[7] DeepSeek-AI. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. doi: 10.1137/0117039.

[10] Mandy Guo, Joshua Ainslie, David Uthus, Santiago Ontanon, Jianmo Ni, Yun-Hsuan Sung, and Yinfei Yang. Longt5: Efficient text-to-text transformer for long sequences. *arXiv preprint arXiv:2112.07916*, 2021.

[11] Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J Fleet. Video diffusion models. *Advances in Neural Information Processing Systems*, 35:8633–8646, 2022.

[12] Jintao Lin, Mingye Luo, Jingwen Zeng, Jianfei Zhai, and Jun Hu. Sageattention2: Efficient attention with thorough outlier smoothing and per-thread int4 quantization. *arXiv preprint arXiv:2411.10958*, 2024.

[13] Jintao Lin, Jia Zhang, Haofeng Zheng, Jingwen Zeng, Jianfei Zhai, and Jun Hu. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. *arXiv preprint arXiv:2410.02367*, 2024.

[14] Jintao Lin, Mingye Luo, Jia Zhang, Zheng Xu, Haofeng Zheng, Jingwen Zeng, Jianfei Zhai, and Jun Hu. Sageattention3: Fast attention on blackwell gpus via fp4 quantization. *arXiv preprint arXiv:2505.11594*, 2025.

[15] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Hongyuan Liu, Qiang Wang, and Xiaowen Chu. Dissecting the nvidia hopper architecture through microbenchmarking and multiple level analysis. *arXiv preprint arXiv:2501.12084*, 2025.

[16] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2nd edition, 2018. ISBN 978-3-319-76525-9.

[17] NVIDIA. Nvidia H100 tensor core GPU architecture, 2022.

[18] NVIDIA. CUDA Programming Guide Version 12.4, 2024. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[19] NVIDIA. Nvidia Blackwell architecture technical brief, 2024. `https://nvdam.widen.net/s/xqt56dflgh/nvidia-blackwell-architecture-technical-brief`.

[20] NVIDIA. cudnn release notes. `https://docs.nvidia.com/deeplearning/cudnn/backend/latest/release-notes.html`, 2025.

[21] Nvidia. Nvidia cutlass documentation, 2025. URL `https://docs.nvidia.com/cutlass/media/docs/pythonDSL/cute_dsl_general/dsl_introduction.html`.

[22] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[23] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL `https://arxiv.org/abs/2407.08608`.

[24] Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, et al. Scrolls: Standardized comparison over long language sequences. *arXiv preprint arXiv:2201.03533*, 2022.

[25] Triton Team. Gluon: A lower-level gpu programming language. `https://github.com/triton-lang/triton/blob/main/python/tutorials/gluon/01-intro.py`, 2024.

[26] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

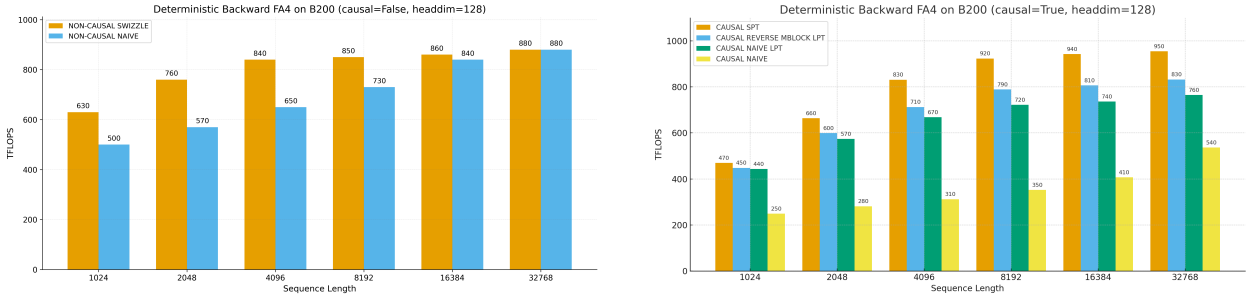

Figure 8: Ablations for Deterministic Backward pass on B200 with head dimension 128. Non-causal attention with batch/head swizzle versus naive. Right: causal attention – SPT, LPT with reverse mblock order, LPT, and naive with no batch/head swizzle.

A Additional Details on Experiments and Benchmarking

A.1 System and libraries

We benchmark the speed on a B100 180GB SXM6 (1000W). We warmup with 5 runs, then repeat the benchmarks 10 times, and take the average timing.

We generally used the latest versions of the libraries at the time of writing (March 2025). Specifically, we use:

- CUDA 13.1
- FlashAttention 2.8.3
- Triton 3.6
- PyTorch 2.10.0
- CuTe-DSL 4.4.1

For cuDNN, in the main paper, we compare to cuDNN 9.13 and the latest version cuDNN 9.19.1.2. Starting from version 9.13 and 9.14 [20], we have worked with the cuDNN team to incorporate some techniques from FlashAttention-4 into cuDNN so that our work can benefit as many practitioners as possible.

A.2 Backward Deterministic non-causal

For completeness, we also include performance numbers for the deterministic backward kernel without causal masking in Fig. 8, side by side with causal masking.