FFPA: Efficient Flash Prefill Attention for Large Head Dimensions via Split-D

DefTruth[†] Butterfingrz[†,‡]

[†]xlite-dev team[*] [‡]Shanghai University

June 2026

Abstract

Attention mechanisms with large head dimensions ($D > 256$) are increasingly common in large multimodal models (e.g., Gemma4-31B), yet standard FlashAttention kernels—optimized for $D \leq 256$—suffer from shared memory exhaustion and register pressure when D grows beyond 256. We present FFPA (Faster Flash Prefill Attention), which introduces **Split-D**, a head-dimension chunking strategy that decomposes $QK^\top$ and PV into sub-operations over D-slices. Split-D reduces SRAM complexity from $O(B_c \times D)$ to $O(B_c \times D_{\text{chunk}}) \approx O(1)$, keeping active register pressure bounded at $O(D_{\text{chunk}})$. The forward pass uses Split-D universally with no precision issues, suitable for both training and inference on any GPU. For the backward pass, two hardware-driven strategies are provided: an fp32 gradient buffer for compute-limited SM≤89 GPUs, and a Hopper-specialized CuTe DSL kernel with full-D WGMMA forward and recompute-based backward for SM≥90 GPUs. Across H200, H800, H20, RTX 5090, and L20, FFPA achieves **1.5–6.1**× speedup over PyTorch SDPA for $D \in [320, 1024]$, reaching **426 TFLOPS** on H200, and **1.4–1.5**× end-to-end training throughput on Gemma4-31B. Code is available at `https://github.com/xlite-dev/ffpa-attn`.

1 Introduction

The attention mechanism [1] is the computational heart of modern transformer models. While most large language models use moderate head dimensions ($D = 64$ or 128), a growing class of architectures—notably large multimodal models such as Gemma4-31B—employ *large* head dimensions of $D = 512$. In these regimes, attention accounts for a dominant fraction of end-to-end inference and training time.

FlashAttention [2–4] revolutionized attention computation by fusing the $QK^\top$–softmax–PV pipeline into a single GPU kernel operating on tiles in SRAM, avoiding $O(N^2)$ writes to high-bandwidth memory (HBM). The core tiling strategy partitions the sequence dimension N into blocks of size B_r (query rows) and B_c (key/value rows), while keeping the *full* head dimension D as the inner dimension of each matrix multiply. This design is optimal when D is small enough that intermediate score tiles and output accumulators fit in shared memory and the register file. However, when $D > 256$, the shared memory footprint of a full-D key/value tile grows to $O(B_c \times D)$, exceeding the SRAM budget and forcing repeated global memory accesses. Simultaneously, the register footprint of the output accumulator ($B_r \times D$) exceeds the GPU register budget, causing spilling to local memory.

We observe that the large-D bottleneck can be addressed by applying tiling *recursively* to the head dimension itself. Our key insight is that the online-softmax recurrence is *independent* of how the $QK^\top$ matmul is computed internally: as long as the final $[B_r, B_c]$ score tile is correctly accumulated, the softmax statistics m_i and ℓ_i remain valid regardless of whether the dot product was computed in one full-D operation or accumulated over multiple D-chunks. This leads to **Split-D**, a head-dimension chunking strategy that we integrate into a complete FlashAttention-style tiled kernel.

A critical challenge arising from Split-D is numerical precision in the **backward pass**. When gradients are accumulated across multiple D-chunks, the bf16 load-modify-store round-trip introduces precision loss proportional to the number of contributing tiles. The forward pass is unaffected—online softmax with fp32 accumulation keeps accuracy within standard mixed-precision bounds, making FFPA equally suitable for inference on any GPU. For the backward pass, we address precision with a **hardware-driven strategy**: on compute-limited SM≤89 GPUs

[*] `https://github.com/xlite-dev`

(e.g., L20 at 119.5 dense FP16 TFLOPS), we use an fp32 gradient buffer that trades IO bandwidth for full-precision accumulation; on compute-rich SM≥90 GPUs (e.g., H200 at ∼989 dense FP16 TFLOPS), we adopt a recompute-based approach that re-materializes the softmax and its gradient across D-chunks, exploiting surplus tensor core throughput to eliminate bf16 round-trip errors.

We make the following contributions:

1. **Split-D tiling**: A head-dimension chunking strategy that *solves* the shared memory bottleneck ($O(1)$ SRAM) and *reduces* Q/K/V register pressure to $O(D_{\text{chunk}})$; the forward O accumulator remains $O(D)$ fp32 (§3.1).
2. **Generic forward/backward kernels** (CUDA & Triton, SM≥80) with a hardware-driven precision strategy: fp32 gradient buffer for SM≤89, where IO overhead is cheaper than recompute.
3. **Hopper-specialized kernels** (CuTe DSL, SM≥90, $D = 512$): full-D $QK^\top$ via a single WGMMA instruction in the forward pass, and a recompute-based backward with fp32 precision channels that eliminate bf16 round-trip loss.
4. **Multi-backend system** spanning CUDA, Triton, and CuTe DSL with automatic dispatch based on GPU architecture and head dimension.
5. **Comprehensive evaluation** across many GPU architectures (H200, H800, H20, RTX 5090, L20) and an end-to-end training verification on Gemma4-31B, achieving **1.5–6.1**× micro-benchmark speedup and **1.4–1.5**× training throughput improvement.

2 Background

2.1 Multi-Head Attention

Given query, key, and value matrices $Q, K, V \in \mathbb{R}^{N\times D}$ for a single attention head, the standard scaled dot-product attention computes:

$$S = \alpha\, QK^\top, \quad \alpha = 1/\sqrt{D}, \tag{1}$$
$$P = \mathrm{softmax}(S), \tag{2}$$
$$O = PV, \tag{3}$$

where softmax is applied row-wise. In practice, a causal mask or additive bias may be applied to S before the softmax. The backward pass computes gradients dQ, dK, dV from the output gradient dO:

$$dP = dO\, V^\top - \mathrm{diag}(\mathrm{rowsum}(dO \odot O))\, P, \tag{4}$$
$$dS = P \odot dP, \tag{5}$$
$$dQ = \alpha\, dS\, K, \quad dK = \alpha\, dS^\top Q, \quad dV = P^\top dO. \tag{6}$$

2.2 FlashAttention and Online Softmax

FlashAttention [2] avoids materializing the $N \times N$ attention matrix in HBM by tiling over the sequence dimension. For a query block of B_r rows, the algorithm streams $T_c = \lceil N/B_c \rceil$ key/value blocks through SRAM, maintaining running statistics $m_i, \ell_i \in \mathbb{R}^{B_r}$ that track the row-wise max and sum of $\exp(S_{i(j)})$ across all KV blocks $j = 0, \ldots, T_c - 1$:

$$m_i^{(j+1)} = \max(m_i^{(j)}, \mathrm{rowmax}(S_{i(j)})), \tag{7}$$
$$\ell_i^{(j+1)} = \ell_i^{(j)} \exp(m_i^{(j)} - m_i^{(j+1)}) + \mathrm{rowsum}\big(\exp(S_{i(j)} - m_i^{(j+1)})\big). \tag{8}$$

The output accumulator is rescaled at each step: $O_i^{(j+1)} = \mathrm{diag}\big(\exp(m_i^{(j)} - m_i^{(j+1)})\big)\, O_i^{(j)} + P_{i(j)} V_j$.

FlashAttention-2 [3] improved work partitioning by assigning each program a query block and optimizing the loop order to reduce shared memory accesses. FlashAttention-3 [4] introduced warp-specialized producer-consumer pipelining and FP8 support for Hopper GPUs.

2.3 The Large Head-Dimension Bottleneck

Standard FlashAttention treats D as a *fused* dimension in every matrix multiply. This creates two bottlenecks when D is large:

Shared memory (SMEM). Each KV block requires loading a full-D key tile of size $B_c \times D$ and a full-D value tile of the same size. At $D = 512$ and $B_c = 64$ with fp16 storage, this demands 64 KB per tile, and with double-buffering the SMEM footprint can exceed the typical 128–228 KB budget on modern GPUs. The standard workaround—reducing B_c—increases the number of HBM round-trips, negating the original purpose of tiling.

Register file. The output accumulator $O_i \in \mathbb{R}^{B_r \times D}$ alone occupies $B_r \times D$ registers. With $B_r = 128$ and $D = 512$, this is 65,536 float values (256 KB), far exceeding the register file of any current GPU (even Hopper's 256 KB is shared across all warps in an SM). The compiler spills these tensors to local memory, causing significant latency.

Numerical precision. Large head dimensions amplify gradient accumulation errors in the backward pass. In standard FlashAttention, each KV block contributes one partial update to dK, dV, and dQ. When these gradients are stored in bf16 (7-bit mantissa), every partial update incurs a bf16 load-modify-store round-trip: the previous partial gradient is loaded from HBM in bf16, the current tile's contribution is added in fp32, and the result is truncated back to bf16 for storage. A single round-trip introduces an error of approximately $\epsilon_{\text{bf16}} = 2^{-7} \approx 7.8 \times 10^{-3}$ relative to the stored value. With $T_c = \lceil N/B_c \rceil$ contributing KV blocks, the accumulated error can exceed 10^{-2} in practice—approaching the scale of the gradients themselves in causal attention, where contributions near the diagonal are sparse and highly non-uniform. At $D = 512$, $N = 8192$, and $B_c = 64$, this amounts to $T_c = 128$ bf16 round-trips per gradient element. This precision challenge is specific to large head dimensions because small-D attention ($D \leq 256$) can use SDPA's fused backward, which avoids explicit per-block gradient storage entirely.

Standard FlashAttention's backward is typically fused through cuDNN or SDPA and does not materialize per-block partial gradients—it accumulates them internally in fp32. For large D, however, the Split-D backward must explicitly accumulate dK, dV, and dQ across D-chunks and KV blocks, making gradient storage precision a first-order concern. Section 3.4 details our hardware-driven solutions.

Split-D addresses the **SMEM bottleneck** directly by reducing the per-tile SRAM requirement to $O(B_c \times D_{\text{chunk}})$, and **reduces** Q/K/V register pressure to $O(D_{\text{chunk}})$ by loading only D_{chunk}-sized slices at a time. The output accumulator, however, must be maintained in fp32 across all D-slices to support online softmax rescaling, and therefore still requires $O(D)$ register storage in the forward pass—the primary obstacle to scaling Split-D to even larger head dimensions.

3 FFPA: Algorithm

FFPA provides a two-tier algorithm family whose choice of backward precision strategy is determined by hardware compute capability. Section 3.1 establishes the Split-D principle. Section 3.2 describes the generic algorithm for SM≥80 GPUs (CUDA and Triton backends). Section 3.3 presents the Hopper-specialized CuTe DSL variant.

3.1 Split-D: Principle and Correctness

Split-D decomposes each D-dimensional matrix multiply into a sequence of smaller multiplies over head-dimension chunks of size D_{qk} (for $QK^\top$) and D_v (for PV).

3.1.1 SMEM Analysis

Split-D fundamentally changes shared memory usage compared to standard FlashAttention: only D_{chunk}-sized tiles reside in SMEM at any time in the forward pass, and K/V tiles can be persisted across iterations in the backward pass. Table 1 summarizes the SMEM and register complexity for all components across both passes.

Forward pass. In the QK phase, only a single D_{qk}-sized chunk of K ($B_c \times D_{\text{qk}}$ elements) resides in SMEM at a time, compared to $B_c \times D$ elements in standard FlashAttention. In the PV phase, each V-group loads a D_v-sized chunk of V ($B_c \times D_v$ elements) into SMEM and immediately multiplies it with the score tile P. Critically, the online softmax rescaling $O_i^{(v)} \leftarrow \alpha_{\text{rescale}} \odot O_i^{(v)} + P v_{\text{tile}}$ (Step 18 in Algorithm 1) operates entirely on registers: the N_v output accumulators $O_i^{(v)}$ are kept in fp32 registers and never spilled to SMEM. The updated $O_i^{(v)}$ is written back to HBM only once, after the final normalization (Step 22). Thus SMEM complexity for the forward pass is $O(B_c \times \max(D_{\text{qk}}, D_v)) \approx O(1)$ with respect to D, fully delivering on the Split-D guarantee.

Backward pass. In the dKdV kernel, the K and V tiles can be loaded once and persisted in SMEM across all Q-block iterations, amortizing HBM traffic. The dK/dV gradient accumulation on SM<90 GPUs uses global load-modify-store, so no additional SMEM is needed beyond the working tiles. On SM≥90 GPUs, the dK/dV registers-local accumulators further eliminate global gradient traffic, while SMEM is used for the dual-precision P tile (bf16 + fp32 channels). SMEM complexity in the backward pass remains $O(B_c \times D_{\text{qkv}})$ for the persisted K/V tiles and $O(B_r \times B_c)$ for the score tile—both independent of the full D.

3.1.2 Register Analysis

Split-D reduces register pressure asymmetrically: the forward pass loads Q, K, and V in D_{chunk}-sized slices but must keep the O accumulator in fp32 across the full D dimension; the backward pass avoids full-D register accumulation through either global load-modify-store (SM<90) or D_{chunk}-sized fp32 register accumulators (SM≥90).

Forward pass. Q, K, and V are loaded in D_{chunk}-sized slices, so their register footprint is $O(D_{\text{chunk}})$. The output accumulator, however, spans the full head dimension: Algorithm 1 maintains $N_v = D/D_v$ accumulator groups $O_i^{(v)} \in \mathbb{R}^{B_r \times D_v}$ (Step 6), and all groups must be rescaled synchronously at each online-softmax update (Step 18). The combined O accumulator therefore consumes $O(B_r \times D)$ fp32 registers—the same as standard FlashAttention—and this $O(D)$ register pressure from the output accumulator is the *primary bottleneck* for scaling Split-D to head dimensions beyond ∼1024.

Backward pass (SM<90). The generic backward kernel (§3.2) uses a global-memory load-modify-store pattern to accumulate dK, dV, and dQ across D-chunks: each chunk loads the previous partial gradient from HBM, adds the current tile's contribution, and stores the updated gradient back. No full-D fp32 accumulators reside in registers; the active register set is $O(D_{\text{chunk}})$ for the Q/K/V/dO working tiles and $O(B_r \times B_c)$ for the S and dP score tiles.

Backward pass (SM≥90). The Hopper-specialized backward (§3.3) keeps dK and dV in D_{chunk}-sized fp32 register accumulators across all query blocks within a KV block, storing only once per D-chunk. Register pressure is $O(D_{\text{chunk}})$ for gradient accumulators plus $O(B_r \times B_c)$ for score tiles—still bounded independent of D.

Table 1: SMEM and register complexity. $D_{\text{qkv}} = \max(D_{\text{qk}}, D_v)$; "persisted" = loaded once and reused across all Q-blocks; "per chunk" = fresh D-slice per iteration. SM<90 uses global load-modify-store for gradients (no fp32 register accumulation).

Component	SMEM	Register	Note
Forward Pass (Split-D, generic)			
Q tile	$O(B_r \times D_{\text{qk}})$	$O(B_r \times D_{\text{qk}})$	per chunk
K tile	$O(B_c \times D_{\text{qk}})$	$O(B_c \times D_{\text{qk}})$	per chunk
V tile	$O(B_c \times D_v)$	$O(B_c \times D_v)$	per chunk
O acc	—	$O(B_r \times D)$ fp32	N_v groups, sync rescale
S tile	$O(B_r \times B_c)$	$O(B_r \times B_c)$	fp32 accumulation
Forward Pass (SM≥90, CuTe DSL)			
For $D \leq 512$: full-D QK via single WGMMA; SMEM and register usage same as standard FlashAttention. For $D > 512$: falls back to the generic Split-D path above.			
Backward Pass (Split-D, SM<90)			
Q tile	$O(B_r \times D_{\text{qk}})$	$O(B_r \times D_{\text{qk}})$	per chunk
K tile	$O(B_c \times D_{\text{qk}})$	$O(B_c \times D_{\text{qk}})$	persisted
V tile	$O(B_c \times D_v)$	$O(B_c \times D_v)$	persisted
dK/dV	—	$O(D_{\text{chunk}})$ per update	global L/M/S, no fp32 acc
dQ	—	$O(D_{\text{chunk}})$ per update	global L/M/S (or atomic-add)
S / dP	$O(B_r \times B_c)$	$O(B_r \times B_c)$	fp32 recompute
Backward Pass (Split-D, SM≥90)			
K tile	$O(B_c \times D_{\text{qk}})$	$O(B_c \times D_{\text{qk}})$	persisted
V tile	$O(B_c \times D_v)$	$O(B_c \times D_v)$	persisted
dK/dV	$O(B_c \times D_{\text{chunk}})$	$O(D_{\text{chunk}})$ fp32	register-local acc
S / dP	$O(B_r \times B_c)$	$O(B_r \times B_c)$	dual bf16 + fp32 channel

3.1.3 Correctness

The online softmax recurrence (Eqs. 7–8) depends only on the final $[B_r, B_c]$ score tile $S_{i(j)}$, not on how many partial dot products were summed to produce it. As long as $S_{i(j)} = \sum_c Q_i^{(c)} K_j^{(c)\top}$ is correctly accumulated across all D_{qk}-chunks before the softmax step, the statistics m_i and ℓ_i remain identical to the full-D computation. The same reasoning applies to the PV phase: the rescaling factor α_i and softmax probabilities $P_{i(j)}$ are computed once per KV block and reused for all V-groups.

3.1.4 HBM IO Analysis

FlashAttention-1 [2] established that the number of HBM accesses for standard attention is $\Theta(ND + N^2)$, while FlashAttention reduces this to $\Theta(N^2 D^2 M^{-1})$, where M is the SRAM capacity in elements. This result implies that FlashAttention is *not* universally preferable to standard MHA: when $D > \sqrt{M}$, the D^2/M factor exceeds 1 and FA's HBM cost surpasses that of unfused MHA. At $D = 512$ and $M \approx 114\text{K}$ (228 KB in bf16), we have $\sqrt{M} \approx 338$, so the regime $D > 256$ falls into this degradation zone.

Split-D resolves this by reducing the *effective* head dimension in the inner tiling loops. Below we derive the HBM cost of Split-D following the FA methodology and show that it corresponds to FlashAttention running at $D = D_{\text{chunk}} \ll \sqrt{M}$.

Standard MHA. Computing $S = QK^\top$, $P = \text{softmax}(S)$, and $O = PV$ without fusion requires three $N \times N$ matrix read/write rounds and two $N \times D$ passes:

$$H_{\text{MHA}} = 4N^2 + 4ND \ \text{(elements)}. \tag{9}$$

FA (standard tiling). Each of the N/B_r query-block programs reads K and V fully—ND elements each—across the inner KV-block loop. The per-program HBM cost is therefore $2ND + 2B_r D$. Multiplying by N/B_r programs and substituting the SRAM constraint $B_r \approx M/(4D)$ (double-buffered Q/K/V tiles each of size $B_r \times D$ compete for SRAM) yields the well-known result:

$$\begin{aligned} H_{\text{FA}} &= \frac{N}{B_r}\,(2ND + 2B_r D) \\ &= \frac{2N^2 D}{B_r} + 2ND \\ &= \Theta\!\left(\frac{N^2 D^2}{M}\right). \end{aligned} \tag{10}$$

Split-D (head-dimension chunking). In Split-D, every tile carries D_{chunk} elements along the head dimension instead of the full D. The outer product $\sum_c Q_i^{(c)} K_j^{(c)\top}$ over D/D_{qk} chunks and the output accumulation over D/D_v groups multiply the number of HBM accesses, but the per-access size shrinks by the same factor, so the element-level total cancels identically (Section 3.1.1, Table 1):

$$H_{\text{SplitD}} = \frac{N}{B_r}\Bigg[\underbrace{\frac{D}{D_{\text{qk}}} \cdot B_c D_{\text{qk}} \cdot \frac{N}{B_c}}_{\text{K: } \frac{D}{D_{\text{qk}}} \text{ chunks} \times B_c D_{\text{qk}} \text{ size} \times \frac{N}{B_c} \text{ KV-blocks}} + \underbrace{\frac{D}{D_v} \cdot B_c D_v \cdot \frac{N}{B_c}}_{\text{V: same structure}} + \underbrace{\frac{D}{D_{\text{qk}}} \cdot B_r D_{\text{qk}}}_{\text{Q: loaded per chunk}} + \underbrace{B_r D}_{\text{O: write once}} \Bigg]$$

$$= \frac{2N^2 D}{B_r} + 2ND. \tag{11}$$

The formula is structurally identical to Eq. 10. The difference lies entirely in the SRAM budget available to B_r. In FA, the dominant SRAM occupant is the full-D Q/K tile ($B_r \times D$), forcing $B_r \approx M/(4D)$. In Split-D, the dominant occupant is the D_{chunk}-wide tile ($B_r \times D_{\text{chunk}}$). With $D_{\text{chunk}} = 64 \ll D$, the tile is 8× narrower, allowing a proportionally larger B_r:

$$B_r^{\text{FA}} \approx \frac{M}{4d}, \qquad B_r^{\text{SplitD}} \approx \frac{M}{4D_{\text{chunk}}}. \tag{12}$$

The ratio is $B_r^{\text{SplitD}}/B_r^{\text{FA}} = D/D_{\text{chunk}} = 512/64 = 8\times$: Split-D supports 8× larger tile sizes along the sequence dimension because the per-tile memory footprint is 8× smaller along the head dimension.

Comparison. Eqs. 10 and 11 reveal a crucial fact: Split-D does *not* change the HBM formula—every thread block still reads the full K and V (ND elements each) to compute its output tile. What Split-D changes is the *number* of thread blocks. In FA, each thread block carries a full-D tile ($B_r \times D$) in SRAM, forcing $B_r \approx M/(4D) \approx 56$ and therefore $N/B_r \approx 146$ blocks along the query dimension. In Split-D, the SRAM occupant is the D_{chunk}-wide tile ($B_r \times D_{\text{chunk}}$), freeing SRAM to support $B_r \approx M/(4D_{\text{chunk}}) \approx 446$ and only $N/B_r \approx 18$ blocks—an 8× reduction in the number of thread blocks, each of which must traverse the same total K/V volume. Fewer blocks mean fewer total K/V passes through HBM, and the $D/D_{\text{chunk}} = 8\times$ factor carries through directly to the asymptotic HBM complexity:

$$H_{\text{FA}} = \Theta\left(\frac{N^2 D^2}{M}\right), \qquad H_{\text{SplitD}} = \Theta\left(\frac{N^2 D \cdot D_{\text{chunk}}}{M}\right). \tag{13}$$

For FA at $D = 512$, $D^2/M \approx 2.3$—every query-key interaction pair costs more than one HBM element, exceeding the N^2 baseline of standard MHA and confirming the degradation predicted by FA Theorem 2. Split-D lowers this ratio to $D \cdot D_{\text{chunk}}/M \approx 0.29$, restoring the HBM advantage by eliminating the D^2 penalty that large head dimensions impose on the FA tiling scheme.

In summary, Split-D inherits FA's HBM formula exactly but improves its worst-case behavior when $D > \sqrt{M}$: by replacing the per-block tile width D with D_{chunk}, it allows B_r to grow by 8×, reducing the total number of full-KV traversals by the same factor. The practical speedup observed in Section 4 is the direct consequence of this reduction, compounded in the backward pass where MHA must materialize $\Theta(N^2)$ intermediate matrices that Split-D avoids.

Table 2: HBM IO comparison (N=8192, D=512, M≈114K, bf16). H from Eq. 10 with $B_r \approx M/(4D_{\text{eff}})$ where D_{eff} is the per-tile head-dimension width.

Algorithm	D_{eff}	B_r	HBM (elements)	GB	D^2/M	$D \cdot D_{\text{chunk}}/M$
MHA	512	—	2.85×10^8	0.57	—	—
FA	512	56	1.23×10^9	2.46	2.30	—
Split-D	**64**	**446**	$\mathbf{1.54 \times 10^8}$	**0.31**	—	**0.29**

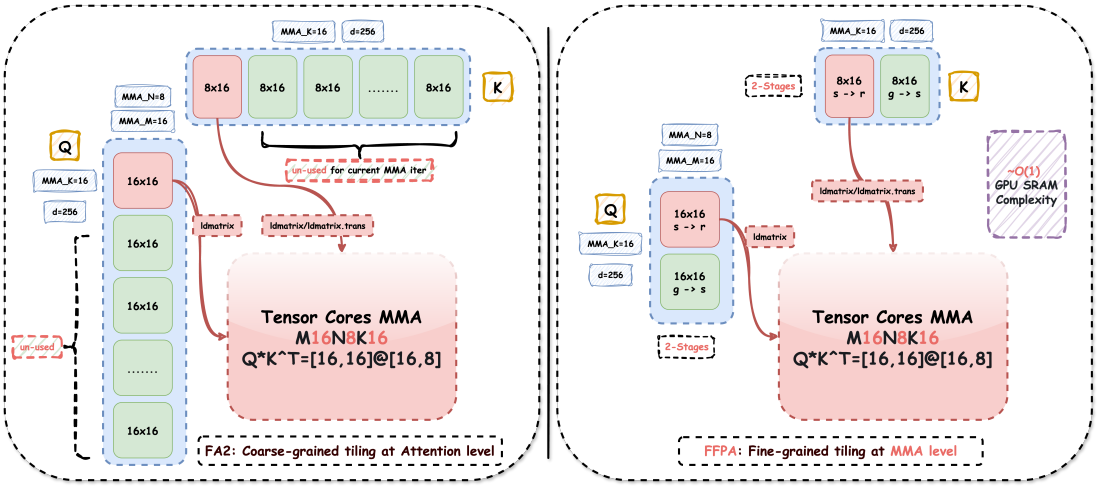


Figure 1: The head dimension D is chunked into D_{qk}-sized slices for $QK^\top$ accumulation and D_v-sized groups for PV computation. Only one chunk resides in SMEM at any time.

3.2 Generic Split-D Algorithm (CUDA & Triton, SM≥80)

The generic Split-D algorithm targets a broad range of GPU architectures (SM≥80: Ampere, Ada, Hopper, Blackwell) through CUDA C++ and Triton implementations. Both forward and backward passes apply full Split-D tiling.

3.2.1 Forward Pass

Algorithm 1 presents the Split-D forward pass for a single program that owns one B_r-sized query block. The outer loop iterates over KV blocks; inside each KV block, a **QK accumulation loop** sums partial dot products over $N_{\text{qk}} = \lceil D/D_{\text{qk}} \rceil$ chunks, followed by online softmax and a **PV accumulation loop** over $N_v = \lceil D/D_v \rceil$ V-groups. **The output accumulator remains $O(D)$ fp32.** Although the QK dot products and PV multiplications are computed over D_{chunk}-sized slices, the O accumulator O_i must be maintained across the full head dimension D to support online softmax rescaling (Step 18 in Algorithm 1). When the running max m_i updates, *all* N_v V-group accumulators must be rescaled synchronously by α_{rescale} to keep the partial sums consistent. This requires the compiler to keep the entire $O_i \in \mathbb{R}^{B_r \times D}$ in fp32 registers, consuming $O(D)$ register space—the same as standard FlashAttention—which is essential for numerical precision but the primary obstacle to supporting head dimensions beyond ∼1024.

Algorithm 1 Generic Split-D Forward Pass (single program)

Input: $Q, K, V \in \mathbb{R}^{N \times D}$, scale $\alpha = 1/\sqrt{D}$
Output: $O \in \mathbb{R}^{N \times D}$, LSE $\in \mathbb{R}^N$
Hyperparams: $B_r, B_c, D_{\text{qk}}, D_v$, $N_{\text{qk}} = \lceil D/D_{\text{qk}} \rceil$, $N_v = \lceil D/D_v \rceil$

$i \leftarrow$ program_id() ▷ owns Q-block i
offs_m $\leftarrow [iB_r, \ldots, (i+1)B_r - 1]$
$m_i \leftarrow (-\infty)^{B_r}$; $\ell_i \leftarrow \mathbf{0}^{B_r}$
for $v = 0$ **to** $N_v - 1$ **do**
    $O_i^{(v)} \leftarrow \mathbf{0} \in \mathbb{R}^{B_r \times D_v}$
for $j = 0$ **to** $\lceil N/B_c \rceil - 1$ **do**
    offs_kv $\leftarrow [jB_c, \ldots, (j+1)B_c - 1]$
    // **Phase 1: Split-D QK accumulation**
    $S \leftarrow \mathbf{0} \in \mathbb{R}^{B_r \times B_c}$
    for $c = 0$ **to** $N_{\text{qk}} - 1$ **do**
        offs_d $\leftarrow [cD_{\text{qk}}, \ldots, (c+1)D_{\text{qk}} - 1]$
        $q \leftarrow \text{load}(Q[\text{offs\_m}, \text{offs\_d}])$
        $k \leftarrow \text{load}(K[\text{offs\_kv}, \text{offs\_d}])$
        $S \leftarrow S + q\,k^\top$
    $S \leftarrow \alpha \cdot S$; apply mask/bias/causal
    // **Phase 2: Online softmax**
    $m_{\text{new}} \leftarrow \max(m_i, \text{rowmax}(S))$
    $\alpha_{\text{rescale}} \leftarrow \exp(m_i - m_{\text{new}})$
    $P \leftarrow \exp(S - m_{\text{new}})$; apply dropout
    $\ell_i \leftarrow \ell_i \cdot \alpha_{\text{rescale}} + \text{rowsum}(P)$
    // **Phase 3: Split-D PV accumulation**
    for $v = 0$ **to** $N_v - 1$ **do**
        offs_d $\leftarrow [vD_v, \ldots, (v+1)D_v - 1]$
        $v_{\text{tile}} \leftarrow \text{load}(V[\text{offs\_kv}, \text{offs\_d}])$
        $O_i^{(v)} \leftarrow \alpha_{\text{rescale}} \odot O_i^{(v)} + P\,v_{\text{tile}}$
    $m_i \leftarrow m_{\text{new}}$
// **Phase 4: Final normalization and writeback**
for $v = 0$ **to** $N_v - 1$ **do**
    $O_i^{(v)} \leftarrow O_i^{(v)} \oslash (\ell_i \oplus \varepsilon)$
    $\text{store}(O[\text{offs\_m}, \text{offs\_d}], O_i^{(v)})$
$\text{store}(\text{LSE}[\text{offs\_m}], m_i + \log \ell_i)$

3.2.2 Backward Pass and Precision Strategy

The backward pass computes dQ, dK, and dV from dO and the saved LSE. FFPA uses two specialized kernels:

dKdV Kernel. Iterates over KV blocks as the outer loop. For each KV block j, it streams all query blocks through SRAM, recomputing $S_{i(j)}$, $P_{i(j)}$, and the softmax statistics via Split-D. It accumulates dV_j and dK_j to

global memory using a load-modify-store pattern.

dQ Kernel. Iterates over Q-blocks as the outer loop, recomputing $S_{i(j)}$ for all KV blocks and accumulating dQ_i. A pre-processing kernel computes $\Delta_i = \text{rowsum}(dO_i \odot O_i)$, enabling $dS_{i(j)} = P_{i(j)} \odot (dO_i V_j^\top - \Delta_i)$.

Algorithm 2 outlines the Split-D dKdV backward pass; the dQ kernel follows a symmetric structure with Q-blocks as the outer loop.

Fused vs. split launch. The dKdV and dQ kernels can be launched either fused (one combined kernel computing both dK/dV and dQ) or split (two separate kernels). In the fused approach, a single kernel must manage the register state for both gradient computations simultaneously. For dQ in particular, reconstructing the full $dS_{i(j)}$ gradient requires access to dO, V, and P—data that is already resident during the dKdV pass. A fused kernel can reuse this data in-place, avoiding redundant loads of Q, K, V, and O from HBM. However, the combined register pressure is substantially higher, and the dQ portion must either use atomic-add to accumulate into the global dQ tensor (when multiple K-tile programs contribute to the same Q-tile) or re-loop over KV tiles per Q-block, incurring a second full pass through Q, K, V, and O.

A split launch assigns dKdV and dQ to independent kernels, each owning a disjoint loop nest (dKdV: KV-block outer; dQ: Q-block outer). This reduces per-kernel register pressure, eliminates the atomic-add on dQ by dedicating each Q-tile to a single program, and enables independent autotuning and backend specialization (e.g., TMA or warp-specialized scheduling) for each kernel. We find that the split strategy yields $\sim$**1.1**$\times$ speedup on SM$<$90 GPUs and $\sim$**2**$\times$ speedup on SM$\geq$90 GPUs with the Hopper CuTe DSL TMA + warp-specialized backend, due to the lower register pressure and finer-grained scheduling control.

Algorithm 2 Generic Split-D Backward Pass — dKdV and dQ (split launch)

Input: $Q, dO, O \in \mathbb{R}^{N \times D}$, saved LSE $\in \mathbb{R}^N$
Output: dK, dV (dKdV kernel) or dQ (dQ kernel)
Hyperparams: $B_r, B_c, D_{\text{qk}}, D_v$

dKdV Kernel (owns K-block j):
Load K-block K_j, V-block V_j into SMEM (persisted across Q-blocks)
for $i = 0$ **to** $\lceil N/B_r \rceil - 1$ **do**
    // **Phase 1: Recompute S, P via Split-D QK**
    $S \leftarrow \mathbf{0}$; accumulate $q\,k^\top$ over D_{qk}-chunks
    Compute $P_{i(j)} = \exp(\alpha S - \text{LSE}_i)$
    // **Phase 2: dV via Split-D PV**
    for $v = 0$ **to** $N_v - 1$ **do**
        $dV_j^{(v)} \leftarrow dV_j^{(v)} + P_{i(j)}^\top dO_{\text{chunk}}$
    // **Phase 3: dS and dK**
    $dP_{i(j)} \leftarrow \sum_v dO_{\text{chunk}} V_{\text{chunk}}^\top$ (Split-D dP)
    $dS_{i(j)} \leftarrow P_{i(j)} \odot (dP_{i(j)} - \Delta_i)$
    for $c = 0$ **to** $N_{\text{qk}} - 1$ **do**
        $dK_j^{(c)} \leftarrow dK_j^{(c)} + dS_{i(j)}^\top q$ (Split-D dK)

dQ Kernel (owns Q-block i, symmetric to dKdV):
The outer loop iterates over K-blocks; inner loop recomputes
    $S_{i(j)}$ and $dS_{i(j)}$ as above, then accumulates
    $dQ_i \leftarrow dQ_i + dS_{i(j)} K_j$ via Split-D D_{qk}-chunks.
A pre-processing kernel computes $\Delta_i = \text{rowsum}(dO_i \odot O_i)$,
    available to both dKdV and dQ.

Precision challenge. When gradients are stored in bf16 (7-bit mantissa), the load-modify-store pattern in dKdV and dQ introduces a bf16 round-trip at every partial update. With up to $T_c = \lceil N/B_c \rceil$ contributing tiles per element ($T_c = 128$ for N=8192, B_c=64, our default benchmark configuration), the accumulated error can exceed 10^{-2} in causal backward, where contributions are unequally weighted near the diagonal.

FFPA solution: fp32 gradient buffer. FFPA provides the option to allocate DK, DV, and DQ tensors in fp32 (`grad_kv_storage_dtype = torch.float32`, `grad_q_storage_dtype = torch.float32`). This doubles the IO bandwidth for gradient reads and writes (fp32 is 2$\times$ bf16), but eliminates bf16 round-trip errors entirely. On compute-limited SM$\leq$89 GPUs (e.g., L20 at 119.5 dense FP16 TFLOPS), this IO-for-precision trade-off is strictly

preferable to recomputing S and dP across D-chunks, which would consume precious tensor core throughput. On SM≥90 GPUs with abundant compute, the recompute approach (§3.3) becomes viable.

3.3 Hopper-Specialized Algorithm (CuTe DSL, SM≥90, $D = 512$)

On Hopper GPUs, the WGMMA (Warp Group Matrix Multiply-Accumulate) instruction can process a full 512×64 tile in a single operation. This enables a qualitatively different algorithm design that is *not* a straightforward port of the generic Split-D kernel.

3.3.1 Forward: Full-D $QK^\top$

The CuTe DSL forward pass performs $QK^\top$ as a **single full-D WGMMA**—no D-chunk loop. The Q tile of shape $[B_r, 512]$ and K tile of $[512, B_c]$ are loaded once via TMA (Tensor Memory Accelerator) and multiplied in one instruction. The softmax and PV phases are then split along the *output column* dimension (Split-N) rather than the head dimension.

WG0 (warps 0–3, Producer) issues TMA loads for Q, K, and V in a double-buffered fashion, overlapping data movement with computation.

WG1 (warps 4–7, Consumer-QK) performs the full-D WGMMA, computes online softmax, and produces the PV-front half (gO columns 0:256) plus LSE. It writes the softmax probability matrix P (bf16) and row-wise rescaling factors (fp32) to shared memory via dedicated pipelines for WG2.

WG2 (warps 8–11, Consumer-PV) consumes the P matrix (bf16) and row scale (fp32) from WG1 via shared-memory pipelines, and computes the PV-back half (gO columns 256:512).

3.3.2 Backward: Split-D with S,dP Recompute

The backward pass returns to Split-D ($D_{\text{chunk}} = 256$, 2 passes), but leverages Hopper's compute headroom to **recompute** S and dP across D-chunks, eliminating the bf16 round-trip that the generic algorithm must tolerate or pay IO to avoid.

The dKdV kernel uses a 2-warpgroup design with explicit named barriers.

WG0 (warps 0–3, Producer) issues TMA loads for K and V, persisting them across both D-passes to amortize HBM traffic.

WG1 (warps 4–7, Consumer-S-dV) recomputes $S_{i(j)} = Q_i K_j^\top$ via 2 WGMMA sub-steps per D-chunk (each sub-step processes half of D_{chunk}=256), applies mask and softmax, then writes P to shared memory in **both** bf16 (for WG1's own dV GEMM) and fp32 (for WG2's dS computation).

WG2 (warps 8–11, Consumer-dS-dK) recomputes $dP = dO\,V^\top$ via 2 WGMMA sub-steps, then reads P from the **fp32 channel** to compute $dS = P \odot (dP - \Delta) \times \alpha$ without any bf16 round-trip, where $\Delta_i = \text{rowsum}(dO_i \odot O_i)$ and $\alpha = 1/\sqrt{D}$ are obtained from the pre-processing kernel.

Named-barrier handshake. The WG1→WG2 P data flow is synchronized by a `PFull`/`PEmpty` named-barrier pair. WG1 arrives at `PFull` after writing both bf16 and fp32 P to SMEM; WG2 waits on `PFull` before reading the fp32 channel. After consuming P and computing dS, WG2 arrives at `PEmpty`, signaling WG1 to reuse the P buffer for the next tile. WG2 pre-arrives at `PEmpty` at kernel startup to prevent deadlock on the first iteration.

Why recompute is viable here. The additional GEMM operations to re-materialize S and dP consume approximately 40% more WGMMA throughput compared to a non-recompute design. On Hopper GPUs with ∼989 dense FP16 TFLOPS of tensor core throughput, this overhead is absorbed by the abundant compute headroom, while the fp32 precision channel guarantees that the softmax gradient—the most numerically sensitive operation in attention backward—suffers no bf16 degradation.

3.4 Backward Precision Strategy: A Hardware-Driven Trade-off

The backward precision challenge is specific to the backward pass. The forward pass has no such issue: Split-D forward uses online softmax with fp32 accumulation for m_i and ℓ_i, keeping output error within standard mixed-precision bounds ($\leq 6 \times 10^{-3}$ in BF16) regardless of storage dtype. Consequently, for inference-only deployments (forward pass only), FFPA works correctly on any GPU without any precision-protection mechanism.

For training (forward + backward), Table 3 summarizes the two backward precision strategies and the hardware characteristics that motivate each choice. On compute-limited SM≤89 GPUs, we pay an IO cost; on compute-rich SM≥90 GPUs, we pay a compute cost. The common goal is eliminating bf16 round-trip errors in gradient accumulation.

Table 3: Backward pass precision strategies. The forward pass has no precision issue and works correctly on any GPU.

	SM≤89 (L20)	**SM≥90 (H200)**
Strategy	fp32 buffer	recompute + fp32 channel
Extra cost	IO (2× bw)	Compute (extra GEMM)
Feasibility	IO ≪ recompute	Surplus compute absorbs
Precision	fp32 accum., no bf16 loss	fp32 P, no bf16 loss

3.5 Multi-Backend Dispatch

FFPA exposes a unified `ffpa_attn_func` API that selects the appropriate backend at runtime:

- $D \leq 256$: fallback to PyTorch SDPA (cuDNN FlashAttention).
- $D > 256$, SM≥90, $D \in [320, 512]$: CuTe DSL backend (Hopper-specialized).
- $D > 256$, SM≥80: Triton backend (default, with optional warp-specialized TMA mode on SM≥90).
- CUDA backend available as a forward-only reference implementation.

Triton kernels support persistent autotuning that pre-computes optimal block sizes $(B_r, B_c, D_{\text{qk}}, D_v)$ and pipeline stages per GPU model, stored as JSON configurations loaded at import time.

4 Experiments

We evaluate FFPA across many GPU architectures, organized by compute capability: Hopper (H200, H800, H20), Blackwell (RTX 5090), and Ada (L20).

4.1 Experimental Setup

Hardware. NVIDIA H200 SXM (Hopper, 989 dense FP16 TFLOPS), NVIDIA H800 PCIe (Hopper, 756 dense FP16 TFLOPS), NVIDIA GeForce RTX 5090 (Blackwell, 419 dense FP16 TFLOPS), NVIDIA H20 (Hopper, 148 dense FP16 TFLOPS), NVIDIA L20 (Ada Lovelace, 119.5 dense FP16 TFLOPS). All experiments use PyTorch 2.11 and CUDA 13.0.

Baseline. We compare against PyTorch's `F.scaled_dot_product_attention` (SDPA) with the "efficient" backend. For $D > 256$, cuDNN FlashAttention is not supported, so SDPA falls back to a memory-efficient attention in all our measurements.

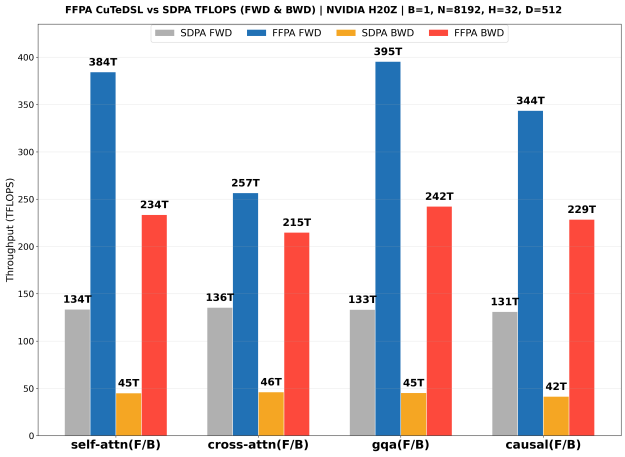
Metrics. Wall-clock latency (milliseconds, median of 10 iterations after 2 warmup), effective TFLOPS computed from the theoretical attention FLOP count ($4N^2D$ forward, $10N^2D$ backward for self-attention), and speedup relative to SDPA. Numerical error is verified via $\max|O_{\text{FFPA}} - O_{\text{SDPA}}|$.

Default configuration. $B = 1$, $H = 32$, $N = 8192$, $D = 512$, BF16 compute with FP32 accumulation. FFPA Triton backend uses "max" autotune mode.

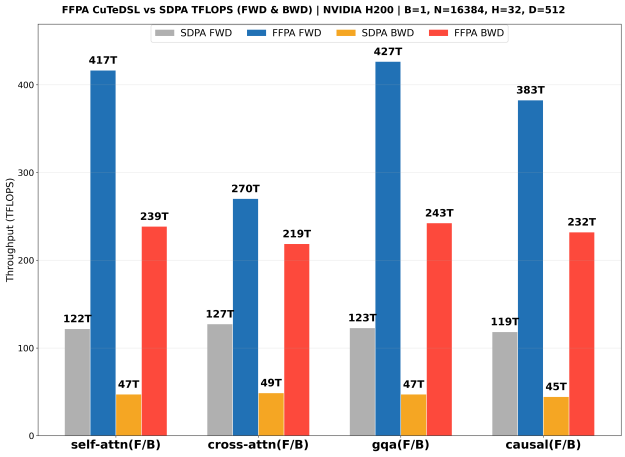
4.2 Hopper Results (H200, H800, H20)

Table 4 reports self-attention results on three Hopper GPUs.

H200 (CuTe DSL). The H200 achieves the highest absolute throughput in our evaluation. At N=8192, forward reaches **372 TFLOPS** (**2.75**×) with a latency of only 11.8 ms; backward reaches **227 TFLOPS** (**4.94**×) at 48.4 ms. Scaling to N=16384, forward throughput rises to **416 TFLOPS** (**3.39**×, 42.3 ms) and backward to **235 TFLOPS** (**4.99**×, 187.0 ms). The forward TFLOPS represents 42% of the H200's 989 TFLOPS peak—a strong result for an attention kernel whose arithmetic intensity is limited by the $O(N^2D)/O(ND)$ FLOP-to-IO ratio. The backward speedup is consistently higher than forward (4.9–5.0× vs. 2.8–3.4×) because SDPA's backward is less optimized at large D and suffers from additional kernel launch and synchronization overhead. GQA (H_{kv}=8) further pushes forward to **426 TFLOPS** at N=16384 (Table 7), exploiting reduced KV memory traffic to better utilize tensor cores.



(a) N=8192



(b) N=16384

Figure 2: H200 CuTe DSL TFLOPS comparison vs. SDPA (D=512).

H800 (Triton). The H800 PCIe variant uses the generic Triton backend. At N=8192, it delivers **203 TFLOPS** forward (**2.64**×) and **87 TFLOPS** backward (**2.53**×). The H800's PCIe bandwidth (50 GB/s vs. H200's NVLink 900 GB/s) is not a bottleneck for single-GPU attention, but the absence of CuTe DSL tailoring means the Triton path achieves 55% of H200's forward TFLOPS and 38% of its backward TFLOPS—consistent with the ratio of their peak dense FP16 throughput (756 / 989).

H20 (Triton). The H20 is the most compute-limited Hopper GPU at 148 TFLOPS, making the IO-for-precision trade-off particularly favorable. At N=8192, forward achieves **112 TFLOPS** (**1.77**×, 39.1 ms) and backward **92 TFLOPS** (**3.26**×, 119.3 ms). The forward speedup is lower than on H200 because SDPA itself achieves high utilization at $D = 512$ when compute is scarce (64 / 148 = 43% vs. FFPA's 112 / 148 = 76%). The backward speedup of **3.26**× demonstrates that Split-D remains highly effective even on compute-limited hardware: the reduced SMEM pressure avoids SDPA's repeated HBM spills, and the IO cost of the fp32 gradient buffer is small relative to the H20's low compute ceiling.

Table 4: Hopper GPU self-attention results (B=1, H=32, D=512, BF16). TFLOPS as FFPA / SDPA; counts attention GEMM FLOPs only. H800 latency derived from reported TFLOPS.

GPU (Backend)	N	Forward			Backward		
		Latency (ms)	TFLOPS	Speedup	Latency (ms)	TFLOPS	Speedup
H200 CuTeDSL	8192	11.8	372 / 135	**2.75**	48.4	227 / 46	**4.94**
	16384	42.3	416 / 123	**3.39**	187.0	235 / 47	**4.99**
H800 Triton	8192	21.7	203 / 77	**2.64**	126.3	87 / 34	**2.53**
H20 Triton	8192	39.1	112 / 64	**1.77**	119.3	92 / 28	**3.26**

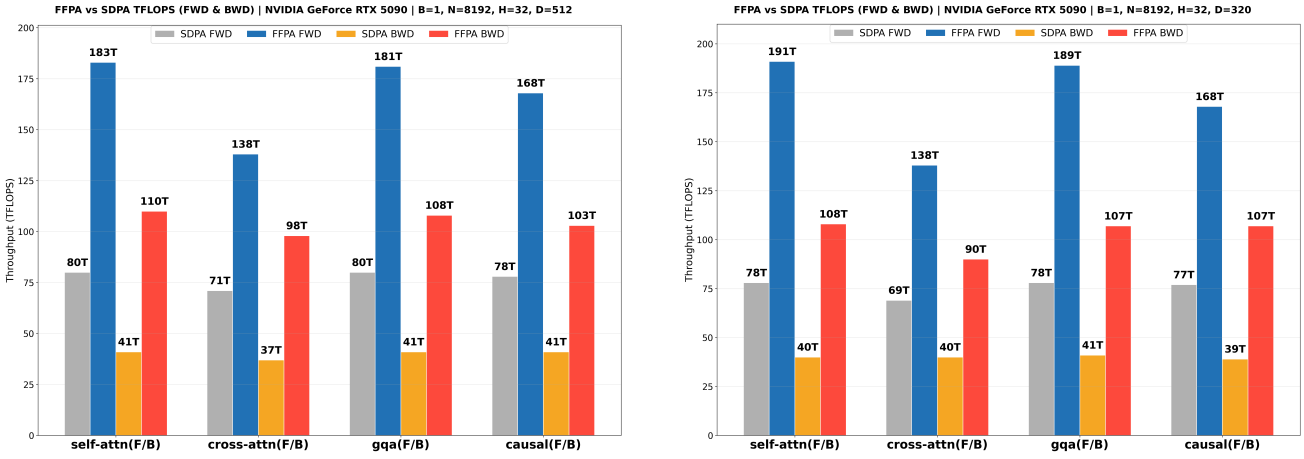
4.3 Blackwell Results (RTX 5090)

Figure 3 shows FFPA Triton results on the RTX 5090 (Blackwell, SM120, 419 dense FP16 TFLOPS).

At D=512, FFPA achieves **2.08**× forward and **2.49**× backward speedup over SDPA. The RTX 5090's Blackwell architecture brings a larger register file per SM (256 KB vs. Hopper's 256 KB on a different SM partition) and support for new WGMMA sub-variants, but the Split-D kernel—running in Triton's generic SM≥80 path—does not yet exploit Blackwell-specific features. The speedup is therefore lower than on H200 (2.75× forward) because the 5090's SDPA baseline achieves higher utilization on its 419 TFLOPS peak (SDPA forward at 156 TFLOPS = 37% peak).

At D=320, the speedup narrows (**1.72**× forward, **2.20**× backward) because SDPA handles moderate D more efficiently—the SMEM bottleneck is less severe at D=320, reducing Split-D's advantage. At D=1024, however, SDPA is entirely infeasible (kernel compilation fails or spills catastrophically), while FFPA continues to run—the

Split-D chunk size $D_{\text{qk}} = 64$ is independent of D, so register pressure per chunk is constant. This makes FFPA the only viable attention backend for $D \geq 768$ on Blackwell GPUs.



(a) D=512 (b) D=320

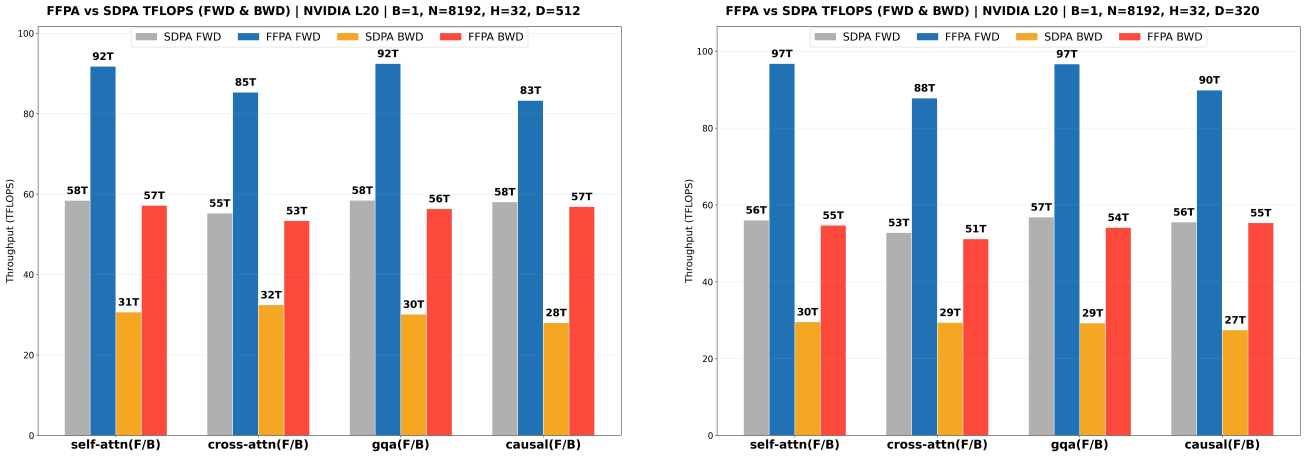
Figure 3: RTX 5090 Triton TFLOPS comparison vs. SDPA.

4.4 Ada Results (L20)

Figure 4 shows FFPA Triton results on the L20 (Ada Lovelace, SM89, 119.5 dense FP16 TFLOPS). The L20 is the least powerful GPU in our evaluation, with only 48 GB of GDDR6 memory and no TMA or warp-specialization support (SM89 lacks the DSMEMP and WGMMA-barrier features of SM90+).

At D=512, forward reaches **1.65**× (45.4 ms FFPA vs. 74.8 ms SDPA). The backward pass supports two precision modes: the bf16 gradient buffer default achieves **1.83**× (192.1 ms vs. 353.2 ms), while the fp32 buffer delivers **1.49**× (250.0 ms vs. 373.6 ms). The bf16-to-fp32 speedup gap of ∼23% reflects the IO cost of reading and writing fp32 gradients on a GPU with 288 GB/s memory bandwidth—modest enough that the fp32 buffer is a viable option when maximum precision is required.

At D=320, the speedup drops to **1.49**× forward and **1.61**× backward (bf16 buffer), as SDPA's standard triton kernel is already well-tuned for moderate D on Ada. The key value of FFPA on L20 is not peak speedup but *capability*: at $D > 512$, SDPA either fails to compile or produces silent correctness errors due to register spilling on Ada's smaller register file, while FFPA runs reliably up to $D = 1024$.



(a) D=512 (b) D=320

Figure 4: L20 Triton TFLOPS comparison vs. SDPA.

Table 5 details the latency breakdown.

Table 5: L20 Triton, self-attention latency in ms (B=1, H=32, N=8192, BF16). Bwd bf16 buffer: FFPA 192.1 ms vs. SDPA 353.2 ms; fp32 buffer: FFPA 250.0 ms vs. SDPA 373.6 ms.

D	Forward			Backward			
	FFPA (ms)	SDPA (ms)	Speedup	FFPA (ms)	SDPA (ms)	Speedup (bf16)	Speedup (fp32)
512	45.4	74.8	**1.65**	192.1	353.2	**1.83**	**1.49**

4.5 Attention Variants

FFPA supports the full range of attention variants: causal masking, GQA/MQA, cross-attention, decode attention ($N_q = 1$), additive bias, and dropout. Table 6 reports speedups across three GPUs (H200 CuTe DSL, H20 Triton, L20 Triton) at D=512, N=8192.

H200 CuTe DSL. Self-attention achieves **2.75**× / **4.94**× (fwd/bwd). GQA (H_{kv}=8) yields the highest forward speedup at **2.98**× and backward at **5.18**×, because reducing KV heads from 32 to 8 cuts SMEM pressure in half, allowing larger tile sizes. Causal masking has a moderate impact (**2.64**× / **5.15**×): the additional mask application step costs <5% throughput but is fused into the online softmax, avoiding a separate kernel launch. Non-aligned sequence lengths achieve **2.59**× / **6.05**×—the highest backward speedup among all variants—because SDPA falls back to a generic attention path for non-multiples of 64, which is particularly slow in the backward pass. Cross-attention (**1.81**× / **4.15**×) has lower forward speedup because N_q=1024 limits the query tile parallelism. Decode attention is not yet supported on the CuTe DSL backend.

H20 Triton. The H20 shows the widest speedup range at D=512: decode backward reaches **6.05**×—the highest single-variant speedup in our evaluation—because SDPA launches a separate kernel per decode token (N_q=1, 8192 tokens) and the launch overhead dominates on a 148 TFLOPS GPU. Decode forward (**3.31**×) benefits from FFPA's specialized GEMV path that fuses the per-token dot product into a batched kernel. Self-attention (**1.77**× / **3.26**×), GQA (**1.74**× / **3.26**×), causal (**1.81**× / **3.15**×), and dropout (**1.59**× / **3.03**×) all show consistent 2–3× backward gains. Attention mask (**1.62**× / **2.80**×) and non-aligned (**1.76**× / **3.40**×) confirm that Split-D handles irregular patterns without performance cliffs.

Table 6: Attention variant **speedups** (B=1, H=32, N=8192, D=512, BF16).

Variant	H200 CuTeDSL		H20 Triton		L20 Triton		
	Fwd	Bwd	Fwd	Bwd	Fwd	Bwd (bf16)	Bwd (fp32)
Self-attn	2.75	4.94	1.77	3.26	1.65	1.83	1.49
Cross (N_q=1024)	1.81	4.15	1.57	3.34	1.59	1.63	1.21
Decode (N_q=1)	—	—	3.31	6.05	1.03	2.20	2.20†
GQA (H_{kv}=8)	2.98	5.18	1.74	3.26	1.65	1.79	1.47
Causal	2.64	5.15	1.81	3.15	1.49	2.00	1.55
Attn mask	—	—	1.62	2.80	1.56	1.93	1.44
Dropout (p=0.1)	—	—	1.59	3.03	1.55	1.74	1.34
Non-aligned	2.59	6.00	1.76	3.40	1.60	1.91	1.54

† Decode-attn uses fp32 GEMV; fp32 buffer has no effect.

L20 Triton. On the L20, speedups are more modest but universal: self-attention (**1.65**× / **1.83**× bf16), decode (**1.03**× / **2.20**×), GQA (**1.65**× / **1.79**×), causal (**1.49**× / **2.00**×), attention mask (**1.56**× / **1.93**×), dropout (**1.55**× / **1.74**×), and non-aligned (**1.60**× / **1.91**×). The decode forward speedup of only **1.03**× reflects the L20's GEMV throughput being already well-utilized by SDPA's cuBLAS path. When using the fp32 gradient buffer, backward speedups drop to 1.2–1.5× (–see Table 6), confirming the ∼23% overhead observed in self-attention.

Scaling to 16K and D=320. Table 7 extends the evaluation to N=16384 and D=320 on H200 CuTe DSL. At D=512, N=16384, forward throughput scales to **426 TFLOPS** (GQA, **3.46**×)—the highest TFLOPS in our evaluation—and backward reaches **5.53**× (non-aligned). At D=320, N=16384, forward achieves **2.61**× (GQA) and backward **4.03**× (non-aligned), confirming that Split-D benefits hold across both sequence lengths and head dimensions. The backward speedup is consistently larger than forward (3.4–5.5× vs. 1.5–3.5×) because

SDPA's backward path at large D requires materializing intermediate attention matrices that Split-D avoids via recomputation.

Table 7: H200 CuTeDSL variant results (B=1, H=32, BF16). TFLOPS as FFPA / SDPA.

Variant	Forward			Backward		
	TFLOPS	SDPA	Speedup	TFLOPS	SDPA	Speedup
			D=512, N=8192			
Self-attn	372	135	**2.75**	227	46	**4.94**
Cross (N_q=1024)	247	137	**1.81**	193	47	**4.15**
GQA (H_{kv}=8)	403	135	**2.98**	238	46	**5.18**
Causal	347	131	**2.64**	217	42	**5.15**
Non-aligned	359	139	**2.59**	233	39	**6.05**
			D=512, N=16384			
Self-attn	416	123	**3.39**	235	47	**4.99**
Cross (N_q=1024)	264	127	**2.08**	198	48	**4.10**
GQA (H_{kv}=8)	426	123	**3.46**	240	47	**5.10**
Causal	381	119	**3.21**	226	44	**5.12**
Non-aligned	403	131	**3.09**	240	43	**5.53**
			D=320, N=8192			
Self-attn	305	138	**2.21**	164	43	**3.82**
Cross (N_q=1024)	216	141	**1.54**	154	44	**3.51**
GQA (H_{kv}=8)	332	140	**2.37**	162	43	**3.77**
Causal	267	135	**1.98**	151	40	**3.77**
Non-aligned	297	143	**2.08**	168	37	**4.57**
			D=320, N=16384			
Self-attn	322	128	**2.52**	161	45	**3.60**
Cross (N_q=1024)	221	133	**1.66**	154	46	**3.35**
GQA (H_{kv}=8)	335	128	**2.61**	172	45	**3.84**
Causal	297	129	**2.30**	151	43	**3.55**
Non-aligned	314	129	**2.44**	170	42	**4.03**

4.6 End-to-End Training

We validate FFPA in a realistic training setting on Gemma4-31B [5], an open-weights multimodal language model whose attention layers are split into two groups: 50 sliding-window layers with $D = 256$ (which remain on SDPA) and 10 full-attention layers with $D = 512$ (accelerated by FFPA). The experiment runs on a single node of 8×H200 GPUs with FSDP2 (data-parallel degree 8, global batch size 8, gradient accumulation 4) and activation checkpointing at sequence length 8K.

At the **same memory footprint** as the SDPA baseline (∼65 GiB per GPU), FFPA CuTe DSL delivers **1.4–1.5**× higher end-to-end training throughput. The training loss curve overlaps the SDPA baseline within normal bf16 noise, confirming that the Split-D forward and the recompute-based backward preserve full numerical fidelity in a production training loop.

Table 8: Training FLOPs breakdown for Gemma4-31B at N=8192 (forward + backward). $h = 5376$, $f = 21504$, $a = 32$. Sliding layers use D=256, kv_h=16; global layers use D=512, kv_h=4.

Component	Sliding (50 layers)	Global (10 layers)	Total	%
Attention	385 T	154 T	539 T	27%
QKV+O Proj	326 T	98 T	424 T	21%
MLP	852 T	170 T	1022 T	52%
Total	**1563 T**	**422 T**	**1985 T**	**100%**

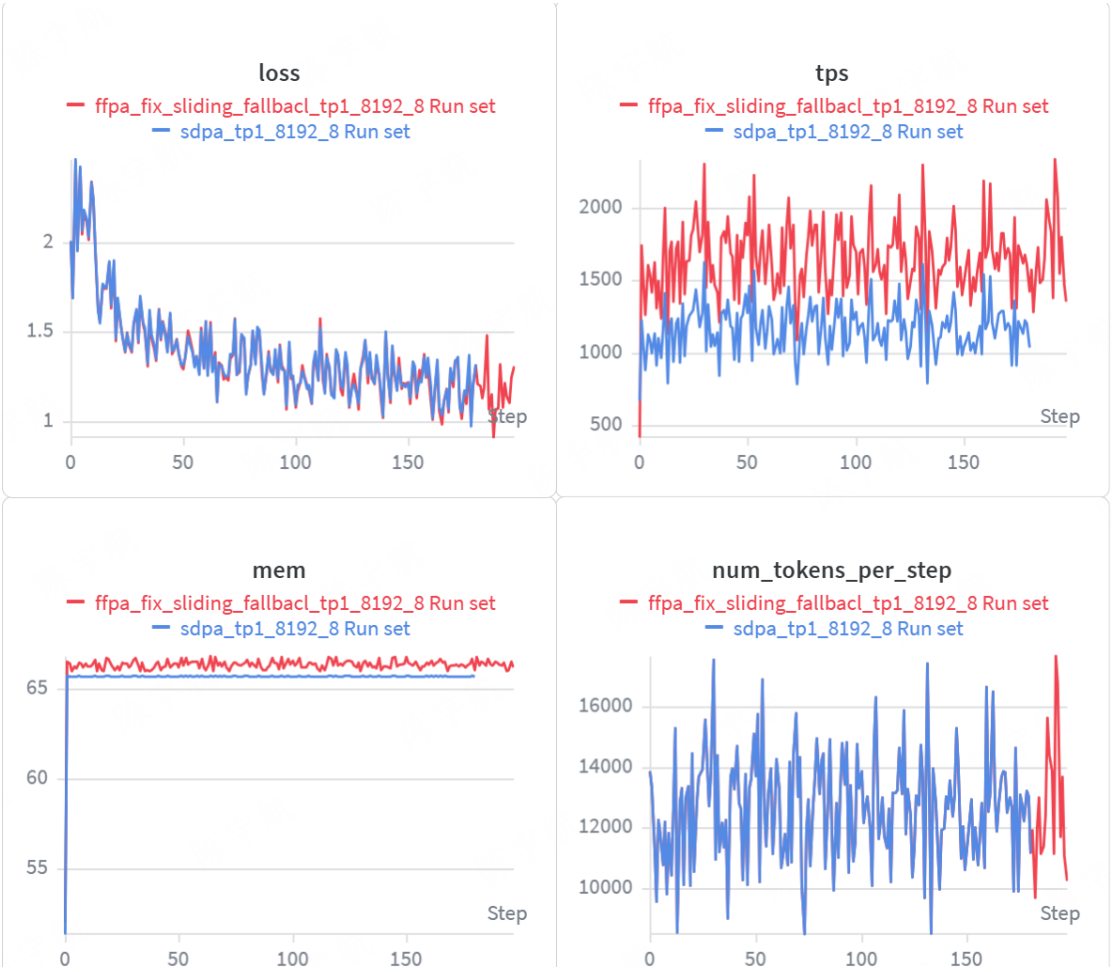


Figure 5: End-to-end training throughput on Gemma4-31B (8×H200, FSDP2, 8K sequence length). FFPA accelerates the 10 D=512 attention layers, achieving 1.4–1.5× overall throughput improvement at the same memory footprint as SDPA.

Table 8 shows the per-component training FLOPs breakdown. The 10 global attention layers (D=512) account for only 17% of the 60 layers but contribute 29% of total attention FLOPs (154 T / 539 T) and 21% of all training FLOPs. However, FLOPs alone understate the bottleneck: dense matrix multiplications (QKV projections and MLP) typically achieve 70–80% of peak TFLOPS on H200, while SDPA attention at D=512 reaches only ∼46 TFLOPS backward (<5% of peak) versus ∼300+ TFLOPS at D=256. Correcting for this utilization gap, attention accounts for ∼55% of total wall time despite representing only 27% of FLOPs. Within attention, the 10 global layers dominate: with >6× lower utilization than the sliding-window layers, they consume **60–70% of attention wall time**—roughly 33–38% of total training time. FFPA accelerates these layers' attention by ∼5×, reducing their contribution to ∼7% of total time and yielding the observed **1.4–1.5**× end-to-end throughput gain. This arithmetic confirms that accelerating only 10 out of 60 layers can deliver a 1.5× overall speedup when those layers are the dominant bottleneck. With FFPA, the previously critical D=512 layers become performance-neutral, shifting the optimization focus to other components of the training pipeline.

4.7 Ablation Study

We analyze the contribution of key design choices using the Triton backend on L20 (N=8192, D=512).

Split-D chunk size. Increasing D_{qk} from 64 to 128 reduces the number of QK accumulation steps by half but increases register pressure per chunk. Autotuning selects $D_{\text{qk}} = 64$ for $D = 512$ on L20, indicating that the reduced register pressure outweighs the benefit of fewer loop iterations. On H800 with larger register files (256 KB per SM), autotuning occasionally selects $D_{\text{qk}} = 128$.

Warp specialization and TMA. On SM≥90 GPUs, enabling TMA with warp specialization yields **44%–69%** higher throughput over the generic Triton baseline (Table 9 shows H200 Triton without TMA/WS; compare

with CuTe DSL results in Table 4). The gain comes from two sources: (1) TMA offloads address computation and data movement from the compute warps, and (2) warp specialization enables concurrent data movement and GEMM execution through hardware warp schedulers. On SM≤89 GPUs without TMA support, these features are unavailable, making the fp32 buffer the primary precision-protection mechanism.

Table 9: H200 Triton baseline without TMA or warp specialization (B=1, H=32, N=8192, D=512, BF16). TFLOPS as FFPA / SDPA.

Variant	Forward			Backward		
	TFLOPS	SDPA	Speedup	TFLOPS	SDPA	Speedup
Self-attn	259	136	**1.91**	134	46	**2.93**
Cross (N_q=1024)	232	138	**1.69**	106	47	**2.29**
Decode (N_q=1)	2.1	0.70	**2.96**	1.7	0.38	**4.48**
GQA (H_{kv}=8)	261	136	**1.93**	133	46	**2.92**
Causal	284	132	**2.16**	109	42	**2.59**
Attn mask	207	130	**1.59**	94	45	**2.12**
Dropout (p=0.1)	188	127	**1.48**	105	45	**2.32**
Non-aligned	245	141	**1.73**	127	39	**3.29**

4.8 Numerical Accuracy

Table 10 summarizes the numerical accuracy of FFPA relative to SDPA across compute precisions and hardware configurations. The forward pass is well-behaved: FP16 achieves $\max|O_{\text{FFPA}} - O_{\text{SDPA}}| \leq 5 \times 10^{-4}$, while BF16 reaches $\leq 6 \times 10^{-3}$, both within expected bounds for mixed-precision attention (FP16 10-bit mantissa vs. BF16 7-bit).

The backward pass is more sensitive. With the default bf16 gradient buffer on SM≤89 GPUs, the load-modify-store pattern introduces bf16 round-trip errors, and the worst-case discrepancy reaches $\max|dV| \approx 8 \times 10^{-2}$ in BF16 causal backward. Enabling the **fp32 gradient buffer** on these GPUs reduces the error to $\leq 2 \times 10^{-2}$ at the cost of 2× gradient IO bandwidth. On SM≥90 GPUs with the CuTe DSL backend, the fp32 precision channel in the recompute-based backward achieves the same $\leq 2 \times 10^{-2}$ bound without the IO penalty, by re-materializing the softmax and its gradient in fp32 within the kernel.

Table 10: Numerical accuracy vs. SDPA (B=1, H=32, N=8192, D=512). Lower is better.

	FP16 Fwd	BF16 Fwd	BF16 Bwd (causal, $\max\|dV\|$)
SM≤89, bf16 buffer	$\leq 5 \times 10^{-4}$	$\leq 6 \times 10^{-3}$	$\approx 8 \times 10^{-2}$
SM≤89, fp32 buffer	—	—	$\leq 2 \times 10^{-2}$
SM≥90, CuTe DSL	$\leq 5 \times 10^{-4}$	$\leq 6 \times 10^{-3}$	$\leq 2 \times 10^{-2}$

5 Conclusion

FFPA solves the shared memory bottleneck and significantly reduces Q/K/V register pressure for large-D attention. We presented FFPA, an attention algorithm that extends FlashAttention to $D > 256$ via Split-D, reducing SRAM complexity from $O(B_c \times D)$ to $O(1)$ and reducing Q/K/V register pressure to $O(D_{\text{chunk}})$. The backward pass, with its global load-modify-store accumulation on SM<90 GPUs and D_{chunk}-sized fp32 register accumulators on SM≥90 GPUs, achieves $O(D_{\text{chunk}})$ register complexity. FFPA provides a hardware-driven precision strategy: fp32 gradient buffers for compute-limited SM≤89 GPUs, and recompute-based precision for compute-rich SM≥90 GPUs with the CuTe DSL backend, which further exploits full-D WGMMA in the forward pass. Across many GPU architectures, FFPA achieves **1.5–6.1**× micro-benchmark speedup and reaches 426 TFLOPS on H200. On Gemma4-31B with 8×H200 GPUs, accelerating only the 10 D=512 layers delivers **1.4–1.5**× end-to-end training throughput improvement (§4).

The forward O accumulator remains the key bottleneck. In the forward pass, the output accumulator must be maintained in fp32 across the full head dimension to support online softmax rescaling of all V-groups

synchronously (§3.1). This $O(D)$ register pressure is the same as standard FlashAttention and is *not* reduced by Split-D. For $D = 512$ with $B_r = 128$, this requires 65,536 fp32 accumulator elements (256 KB), which already exceeds the register budget of current GPUs and is the primary obstacle to supporting $D = 1024$ or beyond.

The backward recompute strategy faces its own scaling limit. For $D > 1024$, the cost of re-materializing S and dP across D-chunks becomes comparable to the forward pass itself, and the fp32 gradient buffer's IO overhead grows with D. This tension between recompute cost and IO bandwidth is the key challenge for extending Split-D backward to head dimensions beyond 1024.

Future work. Addressing the forward O accumulator register bottleneck is the most impactful direction for further scaling Split-D. Promising approaches include warp-specialized accumulator partitioning (splitting the O accumulator across warp groups, as partially explored in the SM90 CuTe DSL forward kernel), FP8 accumulation for the output tensor, and hierarchical tiling that combines Split-D with sequence-parallel attention for distributed inference. Resolving the backward recompute vs. IO trade-off above $D = 1024$, alongside FP8 computation for both forward and backward on Hopper and Blackwell GPUs, are the key medium-term goals.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[2] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[3] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," *International Conference on Learning Representations (ICLR)*, 2024.

[4] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "FlashAttention-3: Fast and accurate attention with asynchrony and low-precision," *arXiv:2407.08608*, 2024.

[5] Google DeepMind, "Gemma 4: Byte for byte, the most capable open models," `https://huggingface.co/google/gemma-4-31B`, 2026.

[6] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[7] NVIDIA Corporation, "CUTLASS: CUDA templates for linear algebra subroutines," `https://github.com/NVIDIA/cutlass`, 2024.

[8] NVIDIA Corporation, "CuTe: CUDA Templates," in CUTLASS 3.x, `https://github.com/NVIDIA/cutlass`, 2024.