FLASHBOB-2: ALGORITHMIC AND HARDWARE OPTIMIZATIONS FOR ATTENTION JVPS AND DOUBLE BACKWARD

Anonymous authors
Paper under double-blind review

ABSTRACT

Jacobian–vector products (JVPs) and second-order derivatives of attention arise in training consistency models and learning through gradient updates. FlashAttention primarily optimizes attention evaluation and first-order reverse-mode differentiation, while specialized kernels such as FlashBoB and `jvp_flash_attention` support forward-mode or second-order operations. Building on FlashAttention-3/-4, we introduce FlashBoB-2 for attention JVPs and backward-over-backward (BoB) on Hopper and Blackwell GPUs, combining a reformulated BoB computation with schedules designed for each GPU. First, we optimize FlashBoB's computational graph, reducing the matrix-multiply count for exact BoB from eighteen to fifteen without storing $N \times N$ intermediates and with on-chip memory independent of sequence length N. Second, on Hopper, separate warpgroups build intermediate results and accumulate outputs, so each output product starts as soon as its inputs are ready. Third, on Blackwell, we reuse Tensor Memory once each value has been read for the last time and keep intermediate results in shared memory. In causal BF16 benchmarks with head dimension 128, FlashBoB-2 achieves BoB speedups of 2.7–$3.0\times$ on H100 and 3.6–$4.2\times$ on B200 over FlashBoB. For non-causal JVPs at 1K–128K against two baselines, speedups are 1.78–$5.44\times$ on H100 and 3.16–$7.34\times$ on B200.

1 INTRODUCTION

Attention is the core operation of the Transformer (Vaswani et al., 2017). Many training methods need derivatives of attention beyond the standard backward pass, and two operations cover most of them. The first is the attention Jacobian-vector product (JVP), which measures how attention's output changes when its inputs move in a given direction. Continuous-time consistency models and mean-flow models train on it; these models generate images, audio, and robot actions in one or a few steps (Lu & Song, 2025; Geng et al., 2025; Li et al., 2026; Chen et al., 2026b). The second is *backward-over-backward* (BoB), the derivative of attention's backward pass (double backward). Methods that differentiate a gradient or a gradient step need BoB, including meta-learning, test-time training, memory written by gradient descent, and second-order optimizers (Finn et al., 2017; Kemaev et al., 2025; Calian et al., 2025; Tandon et al., 2025; Kuratov et al., 2026; Liu et al., 2024a).

Standard kernels support neither operation (Lu & Song, 2025; Kuratov et al., 2026), so some methods write their own kernels (Lu & Song, 2025; Zheng et al., 2026b; Zhou et al., 2026). Others change their losses to avoid or approximate the JVP (Luo et al., 2026; Hu et al., 2026), or fall back on PyTorch's double backward, which runs out of memory at 8K tokens on an H100 (Appendix G.1.2).

Dao et al. (2022) introduced FlashAttention, an exact attention algorithm that is I/O-aware: it splits the attention matrix into blocks called tiles and computes each tile in fast on-chip memory. It never writes the full attention matrix to the GPU's off-chip high-bandwidth memory (HBM), and its backward pass recomputes the tiles it needs. Dao (2024) divided this work better among thread blocks and warps, the groups of 32 threads that run together. Shah et al. (2024) and Zadouri et al. (2026) rebuilt the kernels around the asynchronous Tensor Core instructions of NVIDIA's Hopper and Blackwell GPUs. Givans et al. (2026) carried this execution model to BoB with FlashBoB. Its first pass (Pass 1 in its paper), which we call the *row pass*, goes over blocks of queries and computes two statistics per query row; its second, the *column pass*, goes over blocks of keys and uses them to finish the remaining outputs. On an A6000 GPU, FlashBoB was 2.1–$6.3\times$ faster than FlashBack, a prior exact BoB kernel (Engstrom & Feldmann, 2025).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

However, FlashBoB achieves low utilization on modern GPUs. At $d = 128$ and 4K–256K tokens on an H100, it reaches 25–28% of peak matrix-multiplication (matmul) throughput, while FlashAttention-3's first-order backward reaches 54–62% (Figures 1 and 3).[1] Part of this gap comes from the implementation: FlashBoB is written in Triton, without the asynchronous hardware features that FlashAttention-3 and -4 are built around. The schedules of FlashAttention-3 and -4 do not carry over directly: they were designed for first-order backward, and FlashBoB runs 18 matmuls per tile against its five.

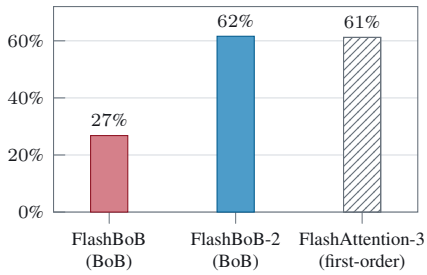

Figure 1: **On an H100 GPU, FlashBoB-2 runs BoB at slightly higher utilization than first-order backward.** Utilization (counted matmul FLOPs/s over BF16 peak) at 16K tokens ($d = 128$, causal), each kernel credited with its own matmul count; first-order backward: one run. Figure 3 has both GPUs and all lengths.

First-order backward makes one pass over the attention matrix and runs five matmuls per tile. FlashBoB's released kernels need two passes and eighteen matmuls per tile (Givans et al., 2026). Two passes are needed because two scalars per query depend on that query's entire row. The technical challenge is twofold: to remove redundant work from FlashBoB's computation, and to fit what remains into each GPU's on-chip memory without letting that memory grow with the sequence length N.

To this end, we propose FlashBoB-2, which reorganizes the computation of BoB and schedules it separately for Hopper and Blackwell GPUs to significantly speedup the computation of BoB on modern GPUs. Our main contributions are as follows:

- **Optimized BoB computational graph:** We remove redundant work from FlashBoB's computational graph while keeping BoB exact. FlashBoB-2 then needs fifteen matmuls per tile instead of eighteen, and on-chip memory stays independent of sequence length, as in FlashBoB (Section 4).
- **Separate warpgroup roles on Hopper:** On Hopper GPUs, matmul results accumulate in registers, and the column pass needs more registers than one warpgroup can hold. We split the work between two warpgroups: one builds the intermediate tiles, and the other runs the output matmuls. Each output matmul starts as soon as its input tile is ready (Section 4.3).
- **Tensor Memory reuse on Blackwell:** The Hopper schedule does not carry over, because Blackwell runs out of a different resource. Its matmuls accumulate in Tensor Memory, a separate on-chip memory, and the column pass's accumulators and recomputed tiles fill it, so we reuse each region as soon as its value has been read for the last time. The matmuls also read their right operand only from shared memory, so we keep the intermediate tiles there, where every matmul can read them (Section 4.4).
- **A fused attention JVP:** Our row pass computes the attention JVP with no extra work, so one pass serves two purposes: it is the first half of BoB and a complete JVP kernel. FlashBoB's first pass also computed the JVP, but it spent five more matmuls per tile on other outputs. The same kernel therefore serves any method that needs a JVP, such as consistency and mean-flow models, and a fused version also returns the attention output (Sections 4.5 and 5.3).
- **Experimental validation:** We benchmark FlashBoB-2 on H100 and B200 GPUs. At head dimension 128, FlashBoB-2 runs BoB 2.7–3.0× faster than FlashBoB on H100, where it matches the utilization of FlashAttention-3's first-order backward, counting each kernel's own matmuls. On B200, it is 3.6–4.2× faster than FlashBoB. Our attention JVP is 1.8–2.0× faster than the fastest existing JVP kernel we measured on H100 and 3.2–3.7× on B200 (Section 5).

2 Related Work

I/O-aware attention. Long-context attention has been pursued along two lines. Approximate methods change the operator: sparse patterns, low-rank projections, kernelized attention, and state-space models reduce or replace the quadratic interaction between queries and keys (Kitaev et al., 2020; Beltagy et al., 2020; Zaheer et al., 2020; Wang et al., 2020; Katharopoulos et al., 2020; Choromanski et al., 2021; Gu & Dao, 2024). FlashAttention keeps exact attention and changes the memory

[1] Counted matmul FLOPs per second over BF16 peak, as in FlashAttention-2 (Dao, 2024).

schedule instead (Dao et al., 2022). Building on the online softmax and memory-efficient attention (Milakov & Gimelshein, 2018; Rabe & Staats, 2021), it recomputes attention tiles from saved softmax statistics, so the full attention matrix is never written to HBM. Saha & Ye (2024) analyze its I/O complexity. FlashAttention-2 improved its work partitioning (Dao, 2024). FlashAttention-3 overlaps data movement and matmuls on Hopper through warp specialization and asynchronous matmul instructions (Shah et al., 2024). FlashAttention-4 redesigns these pipelines around Blackwell's Tensor Memory (Zadouri et al., 2026). FlashBoB-2 belongs to the exact line. It uses the hardware mechanisms of FlashAttention-3 and -4, but it needs new schedules because BoB's column pass does more work per tile than first-order backward in FlashAttention series.

Kernel programming tools and warp specialization. Several systems make FlashAttention-style kernels easier to write. Triton compiles block-level Python programs to GPU kernels (Tillet et al., 2019). CUTLASS and its Python interface CuTe DSL provide building blocks for CUDA kernels, and cuDNN provides tuned primitives (Thakkar et al., 2023; NVIDIA, 2025; Chetlur et al., 2014). ThunderKittens provides tile-level primitives, and FlexAttention compiles attention variants written in PyTorch (Spector et al., 2025; Dong et al., 2025). Warp specialization, which gives different warps different jobs, dates to CudaDMA and Singe (Bauer et al., 2011; 2014). Twill and Tawa generate warp-specialized schedules automatically for a given loop (Soi et al., 2026; Chen et al., 2026a). These tools schedule a computation that is fixed in advance. FlashBoB-2 first changes the computation of BoB, then designs a schedule for each GPU.

Implementations of attention BoB and JVPs. Automatic differentiation computes BoB by differentiating the backward program itself (Drucker & Le Cun, 1992; Pearlmutter, 1994; Griewank & Walther, 2008; Baydin et al., 2018). Differentiating standard attention twice in PyTorch or JAX gives a correctness reference (Paszke et al., 2019; Bradbury et al., 2018), but it stores intermediates the size of the attention matrix, and checkpointing recomputes them without removing them (Chen et al., 2016). FlashBack fuses exact BoB into one kernel that sums the key-side outputs with atomic adds (Engstrom & Feldmann, 2025). FlashBoB splits it into two streaming passes without atomics (Givans et al., 2026); neither stores an intermediate of that size. For the JVP, sCM proposes a FlashAttention-style kernel (Lu & Song, 2025), and `jvp_flash_attention`, rCM, Causal-rCM, and Terminal Velocity Matching write their own (Morehead, 2025; Zheng et al., 2026b;a; Zhou et al., 2026). FlashBoB-2 starts from FlashBoB's two passes. It removes their redundant work, adds CuTe DSL kernels for Hopper and Blackwell, and uses its row pass as a JVP kernel.

Applications. Second-order optimizers estimate curvature with Hessian-vector products (Pearlmutter, 1994; Yao et al., 2021; Liu et al., 2024a). Meta-learning, data curation, test-time training, and gradient-written memory differentiate through inner gradient steps (Finn et al., 2017; Kemaev et al., 2025; Calian et al., 2025; Tandon et al., 2025; Kuratov et al., 2026). Continuous-time consistency models and mean-flow models train on attention JVPs (Lu & Song, 2025; Geng et al., 2025; Zheng et al., 2026b). Appendix B lists these methods and what each needs from attention.

3 BACKGROUND

Attention. For one attention head (Vaswani et al., 2017) with queries, keys, and values $Q, K, V \in \mathbb{R}^{N\times d}$, sequence length N, and head dimension d,

$$S = \tau QK^{\mathsf{T}} + M, \qquad P = \mathrm{softmax}(S), \qquad O = PV, \tag{1}$$

where $\tau = 1/\sqrt{d}$, softmax is row-wise, and the mask $M \in \{0, -\infty\}^{N\times N}$ removes disallowed entries. Lowercase letters denote rows. Appendix A lists all symbols.

The FlashAttention execution model. A *thread block* is a group of GPU threads on one streaming multiprocessor (SM) that share its on-chip *shared memory*. A *tile* is the block of S or P for a query block of B_r rows and a key block of B_c rows (Dao et al., 2022; Dao, 2024). FlashAttention is exact: it computes softmax attention and its derivatives without approximation, up to floating-point rounding (FlashBoB-2 likewise; Appendix F). It stores no $N \times N$ matrix. Each thread block holds a fixed number of tiles, independent of N, recomputed on chip from the inputs and saved per-row statistics. The forward pass saves only O and the log-sum-exp $L_i = \log \sum_j \exp(S_{ij})$, which recovers any tile of P as $P_{ij} = \exp(S_{ij} - L_i)$.

First-order backward. With dX the loss gradient of X, the backward pass computes

$$dP = dO\, V^{\mathsf{T}}, \quad dS_{ij} = P_{ij}(dP_{ij} - D_i), \quad dV = P^{\mathsf{T}} dO, \quad dQ = \tau\, dS\, K, \quad dK = \tau\, dS^{\mathsf{T}} Q, \tag{2}$$

where $D_i = \sum_j P_{ij}\, dP_{ij}$ sums over row i. FlashAttention obtains D_i without forming dP (Dao et al., 2022, App. B.2, Eq. 4):

$$D_i = do_i^{\mathsf{T}} o_i. \tag{3}$$

The backward pass is then one pass: each thread block holds one key block, runs five matmuls per tile, and adds its partial dQ into HBM with atomic adds (Dao, 2024; Shah et al., 2024). FlashBoB-2 uses them only for $\widetilde{d}(Q)$; FlashBoB avoids them (Givans et al., 2026).

Attention JVPs and BoB. Write $O = f(Q, K, V)$ and let $U_Q, U_K, U_V \in \mathbb{R}^{N\times d}$ be given tangents (input directions). The attention JVP returns O and its derivative $\dot{O}$ along these tangents; a dot marks such a derivative. BoB instead differentiates the backward pass (2). There, U_Q, U_K, and U_V are the upstream gradients of dQ, dK, and dV from the rest of the network. Define $\langle A, B\rangle = \mathrm{tr}(A^\top B)$ as the Frobenius inner product. BoB returns the derivatives of the scalar $\Phi = \langle dQ, U_Q\rangle + \langle dK, U_K\rangle + \langle dV, U_V\rangle$ with respect to the inputs of the backward pass: $\widetilde{d}(Q) = \partial\Phi/\partial Q$, and likewise $\widetilde{d}(K)$, $\widetilde{d}(V)$, and $\widetilde{d}(dO)$. The two operations are closely linked. With J the Jacobian of f and $U = (U_Q, U_K, U_V)$, the backward pass returns $J^{\mathsf{T}} dO = (dQ, dK, dV)$, so $\Phi = \langle J^{\mathsf{T}} dO, U\rangle = \langle dO, JU\rangle = \langle dO, \dot{O}\rangle$. Hence $\widetilde{d}(dO) = \dot{O}$: BoB's fourth output is the attention JVP.

FlashBoB computes BoB under the FlashAttention execution model (Givans et al., 2026). BoB needs two more row statistics, $\alpha_i = \dot{L}_i$ and $E_i^\circ = \dot{D}_i$, the derivatives of the log-sum-exp and of D_i. Each sums over every key in the row, and every key-block output depends on both, so FlashBoB uses two passes. The *row pass* holds a query block, streams over its key blocks, and completes $\widetilde{d}(dO)$, $\widetilde{d}(Q)$, and the two statistics. The *column pass* holds a key block, streams over the query blocks, and completes $\widetilde{d}(K)$ and $\widetilde{d}(V)$. Only the $2N$ statistics cross between the passes. The released FlashBoB implementation runs ten matmuls per tile in the row pass and eight in the column pass.

Hopper. On an H100, a *warpgroup* of four warps issues WGMMA (warpgroup-level matrix multiply-accumulate), the asynchronous Tensor Core matmul instruction (NVIDIA Corporation, 2022; 2026). WGMMA accumulates its results in the warpgroup's registers, at most 255 per thread. FlashAttention-3 pairs it with the Tensor Memory Accelerator (TMA), which copies tiles asynchronously, in *warp specialization* (Bauer et al., 2011; Shah et al., 2024): a producer warpgroup loads tiles into shared memory while consumer warpgroups issue matmuls.

Blackwell. On a B200, a single thread issues the `tcgen05.mma` matmul, which accumulates in Tensor Memory (TMEM) instead of registers. TMEM holds 128 lanes (rows) by 512 columns of 32-bit values per SM, all allocatable by one thread block (NVIDIA Corporation, 2026). A matmul's left operand (A) comes from TMEM or shared memory, its right operand (B) only from shared memory.

4 FlashBoB-2: Algorithm and Implementation

4.1 Optimized Computational Graph for BoB

Where FlashBoB repeats work. FlashBoB's row pass runs ten matmuls per tile, but only five of them compute $\widetilde{d}(dO)$. The other five serve two purposes. Two recompute the tiles $dP = dO\, V^{\mathsf{T}}$ and $C = dO\, U_V^{\mathsf{T}}$, which the column pass also recomputes, to form $E_i^\circ = \dot{D}_i$. Three accumulate $\widetilde{d}(Q)$, whose terms also need dP and C. We remove both groups from the row pass.

Removing the repeated work. FlashBoB forms E° by differentiating the definition $D_i = \sum_j P_{ij}\, dP_{ij}$, which needs the tiles of dP and C. We instead differentiate FlashAttention's identity $D_i = do_i^{\mathsf{T}} o_i$ (Equation 3) with dO fixed. Since $\widetilde{d}(dO) = \dot{O}$, this gives

$$E_i^\circ = do_i^{\mathsf{T}}\, \widetilde{d}(dO)_i, \tag{4}$$

a row dot product with an output that the row pass already computes. Just as FlashAttention forms D without dP, the row pass now forms E° without dP or C. It still needs them for $\widetilde{d}(Q)$, but the column pass recomputes both for its own outputs, so we compute $\widetilde{d}(Q)$ there (Appendix C).

The resulting passes. The row pass runs five matmuls per tile: S, the two terms of the score derivative $F = \dot{S} = \tau(U_Q K^{\mathsf{T}} + Q U_K^{\mathsf{T}})$, $P\, U_V$, and $(P \odot F)V$. It completes $\widetilde{d}(dO)$ and the first row

statistic $\alpha = \mathrm{rowsum}(P \odot F)$, then forms E° with Equation (4). Given the saved O and L, $\widetilde{d}(dO)$ is exactly the attention JVP $\dot{O}$ (Section 4.5). The column pass runs ten matmuls per tile (Figure 2). It recomputes S, F, dP, and C with five. With $X_{ij} = dP_{ij} - D_i$, it then forms three intermediate tiles, dS and the derivatives $\widetilde{d}(dP) = \dot{P}$ and $\widetilde{d}(S) = \dot{dS}$:

$$dS = P \odot X, \qquad \widetilde{d}(dP) = P \odot (F - \alpha), \qquad \widetilde{d}(S) = P \odot \big((F - \alpha) \odot X + C - E^\circ\big). \tag{5}$$

Five more matmuls read these tiles and give the outputs

$$\widetilde{d}(V) = \widetilde{d}(dP)^\mathsf{T} dO, \qquad \widetilde{d}(K) = \tau\big(\widetilde{d}(S)^\mathsf{T} Q + dS^\mathsf{T} U_Q\big), \qquad \widetilde{d}(Q) = \tau\big(\widetilde{d}(S)\, K + dS\, U_K\big). \tag{6}$$

Each thread block holds a key block, finishes $\widetilde{d}(K)$ and $\widetilde{d}(V)$ for it, and adds a partial $\widetilde{d}(Q)$ that is summed over key blocks, like dQ in first-order backward.

Matmul count. FlashBoB-2 thus computes BoB exactly with $5 + 10 = 15$ matmuls per tile, against FlashBoB's $10 + 8 = 18$, and like FlashBoB it stores no $N \times N$ matrix. Among the direct two-pass splits of Appendix C.3, the $5 + 10$ split needs the fewest matmuls per tile, because every other one recomputes dP in both passes (Proposition C.1). Every variant with fewer matmuls that we found stores row-pass tiles for the column pass, and those tiles grow with N. Ten of the fifteen matmuls now run in the column pass, so its schedule decides performance.

4.2 Scheduling Requirements of the Column Pass

Per tile, the column pass runs ten matmuls, twice as many as first-order backward, and it differs in three ways that each GPU meets differently (Figure 2).

More live state. It holds four recomputed tiles (S, F, dP, C; first-order backward holds S and dP) next to the $\widetilde{d}(K)$, $\widetilde{d}(V)$, and partial $\widetilde{d}(Q)$ accumulators. Hopper keeps matmul results in registers, at most 255 per thread, and this state does not fit one warpgroup (Section 4.3). Blackwell keeps them in Tensor Memory, which it fills (Section 4.4).

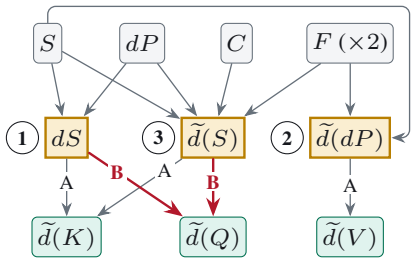

Figure 2: **Why the column pass needs a new schedule.** Per tile, it runs ten matmuls and forms three intermediate tiles (orange), signalled on Hopper in the numbered order; on Blackwell, dS and $\widetilde{d}(S)$ are also read as the right operand (B) of the $\widetilde{d}(Q)$ matmuls. Gray: recomputed tiles; green: outputs.

Tiles ready at different times. dS is ready after dP, $\widetilde{d}(dP)$ after F, and $\widetilde{d}(S)$ after dP, F, and C. FlashAttention-3 instead forms P and then dS in one short chain (Shah et al., 2024). This order matters on Hopper because the state is split across warpgroups: each tile passes through shared memory from the warpgroup that forms it to the one that runs the output matmuls. With asynchronous matmuls, each output matmul can start when its tile arrives, if tiles are signaled separately (Section 4.3).

Tiles read as right operands. On both GPUs, a matmul reads its right operand only from shared memory (NVIDIA Corporation, 2026). On Hopper this costs nothing, since the tiles already pass through shared memory. On Blackwell, matmul results land in Tensor Memory, and a tile stored there can supply only the left operand. FlashAttention-4 reads P and dS only as left operands, so P can stay there (Zadouri et al., 2026). Our column pass keeps the keys on the Tensor Memory lanes, because $\widetilde{d}(K)$ and $\widetilde{d}(V)$ have one row per key, so it computes $\widetilde{d}(Q)$ transposed. Its $\widetilde{d}(Q)$ matmuls then read dS and $\widetilde{d}(S)$ as right operands, so these tiles must be copied to shared memory, where one BF16 copy serves both operand roles (Section 4.4).

We inherit three mechanisms from FlashAttention-3 and -4. A producer issues the TMA loads (Section 3). A TMA reduce-add, an asynchronous bulk add into HBM, sums the partial $\widetilde{d}(Q)$ tiles. The sum runs in completion order, so $\widetilde{d}(Q)$ is not bitwise repeatable; Hopper has a slower deterministic mode (Appendix F). On Blackwell, a single MMA warp issues all matmuls.

Algorithm 1 Separate roles in the Hopper column pass (one key block). Warpgroups: P producer, C construction, A accumulation (the math symbol C is a recomputed tile). Each tile is signalled when stored, so output matmuls need not wait for the last tile.

1: **for** each query block i **do**
2: P: load Q_i, $U_{Q,i}$, dO_i; C reads the row statistics of block i
3: C: recompute S and $[dP \mid C]$; form P; issue the two F matmuls
4: C: store $dS = P \odot (dP - D)$; **signal** tile 1
5: A: **wait** tile 1; $\widetilde{d}(K) \mathrel{+}= \tau\, dS^{\mathsf{T}} U_{Q,i}$; $\widetilde{d}(Q)_i \leftarrow \tau\, dS\, U_K$
6: C: **wait** F; store $\widetilde{d}(dP)$, then $\widetilde{d}(S)$; **signal** tiles 2 and 3
7: A: **wait** tile 2; $\widetilde{d}(V) \mathrel{+}= \widetilde{d}(dP)^{\mathsf{T}} dO_i$
8: A: **wait** tile 3; $\widetilde{d}(K) \mathrel{+}= \tau\, \widetilde{d}(S)^{\mathsf{T}} Q_i$; $\widetilde{d}(Q)_i \mathrel{+}= \tau\, \widetilde{d}(S)\, K$
9: P: TMA reduce-add of $\widetilde{d}(Q)_i$ into the global FP32 $\widetilde{d}(Q)$
10: **end for**

4.3 Hopper: Splitting Tile Construction and Accumulation

Register limit. On H100, a warpgroup accumulates matmul results in registers, at most 255 per thread. At $d = 128$, recomputing four tiles (S, F, dP, C) and forming three intermediate tiles needs about 144 registers per thread. The $\widetilde{d}(K)$, $\widetilde{d}(V)$, and partial $\widetilde{d}(Q)$ accumulators need 192 more (both estimated from how matmul results are laid out across the warpgroup's threads). Giving each output its own warpgroup, which holds the tile state and that output, would fit ($144 + 64$ registers), but each would recompute the tiles it reads: by our count, 18 column-pass matmuls instead of 10. This count excludes designs that move outputs to shared memory or split them across launches.

Separate roles. We instead give each consumer warpgroup one role (Algorithm 1; Appendix D.2). The *construction* warpgroup runs the five recomputation matmuls and forms the three intermediate tiles from their results, which needs about 144 registers per thread. The *accumulation* warpgroup runs the five output matmuls and holds the $\widetilde{d}(K)$, $\widetilde{d}(V)$, and partial $\widetilde{d}(Q)$ accumulators, which need about 192. The producer loads the inputs. Each role then fits within the 255-register limit.

One barrier per tile. If one signal covered all three tiles, all five output matmuls would wait for the last tile, which three of them do not need. We give each tile its own shared-memory buffer and barrier, a flag one warpgroup sets and the other waits on. Output matmuls run while later tiles are formed (Algorithm 1, lines 5 and 7 need not wait for the last tile), and the dS buffer is freed early.

Row pass. The row-pass state fits in registers, with no room for a second key tile. As in FlashAttention-3's pingpong schedule (Shah et al., 2024), two consumer warpgroups take adjacent query blocks that share key tiles, so one issues matmuls while the other forms P and $P \odot F$.

4.4 Blackwell: Tensor Memory Reuse and Shared-Memory Operands

Tensor Memory capacity. Blackwell matmuls accumulate into TMEM (128 lanes by 512 columns of 32-bit values), whose capacity replaces the register limit. As in FlashAttention-4, we store the column-pass tiles transposed, with the 128 keys along the lanes, because $\widetilde{d}(K)$ and $\widetilde{d}(V)$ have one row per key. At $d = 128$, $\widetilde{d}(K)$ and $\widetilde{d}(V)$ take 128 FP32 columns each for the whole query loop. The current S^{T}, F^{T}, dP^{T}, and C^{T} take 64 each, filling all 512 columns and leaving none for the partial $\widetilde{d}(Q)$ (Figure 9). FlashAttention-4 already reuses TMEM in its backward pass, writing P over S and dS and dQ over dP (Zadouri et al., 2026). The column pass needs such reuse too. After C^{T} is last read, to form $\widetilde{d}(S)$, its 64 TMEM columns accumulate the partial $\widetilde{d}(Q)^{\mathsf{T}}$.

Left and right operands. A Blackwell matmul reads its left operand from TMEM or shared memory, its right operand only from shared memory. The $\widetilde{d}(K)$ and $\widetilde{d}(V)$ matmuls read the intermediate tiles as left operands, which could come from TMEM. The $\widetilde{d}(Q)$ matmuls compute $\widetilde{d}(Q)^{\mathsf{T}} = \tau\big(K^{\mathsf{T}} \widetilde{d}(S)^{\mathsf{T}} + U_K^{\mathsf{T}} dS^{\mathsf{T}}\big)$ with the head dimension along the lanes, so they read $\widetilde{d}(S)^{\mathsf{T}}$ and dS^{T} as right operands. Computing $\widetilde{d}(Q)$ untransposed would need tiles with query rows along the lanes, which a keys-on-lanes TMEM tile cannot supply. We write all three tiles to shared mem-

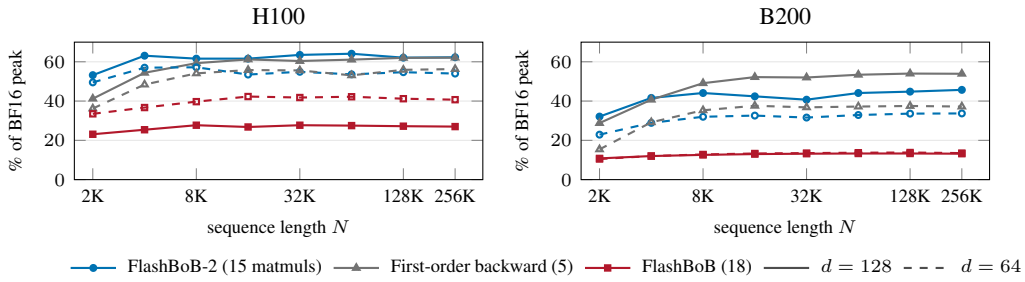

Figure 3: **On H100, FlashBoB-2 runs BoB at the utilization of first-order backward.** Counted matmul FLOPs per second over BF16 peak; causal, batch 4, 12 heads; medians of three or four runs. First-order backward (one run): FlashAttention-3 (H100), -4 (B200).

ory in BF16, so one copy serves as both the left and the right operand. After their last read, the dS and $\widetilde{d}(S)$ buffers receive the FP32 partial $\widetilde{d}(Q)$ from TMEM for the reduce-add.

Matmul issue. At $d = 128$, one MMA warp issues the ten matmuls in order, eight compute warps form the tiles, and one load warp issues loads. Section 5.2 compares this with three MMA warps.

Row pass. The row pass has no such conflict: its output matmuls $P\,U_V$ and $(P \odot F)V$ read P and $P \odot F$ as left operands, so both tiles stay in TMEM (Appendix D.3).

4.5 Attention JVPs

The row pass is a JVP kernel. Given the saved O and L, the row pass returns $\dot{O} = \widetilde{d}(dO)$ and $\dot{L} = \alpha$. Without its E° epilogue it never reads dO, so a JVP-only method can call it directly.

A fused kernel without a saved forward pass. Consistency models such as sCM compute attention and its JVP in one forward pass (Lu & Song, 2025). For them, a fused kernel returns O, $\dot{O}$, and L together. It keeps an online softmax, accumulates PV next to $\dot{O}$ as a sixth matmul per tile, and applies the $-\operatorname{Diag}(\alpha)O$ correction, on the row-pass schedules of Sections 4.3 and 4.4. On B200 it adapts FlashAttention-4's conditional rescaling, which rescales only when the row maximum rises by more than a fixed threshold; the H100 kernel rescales every tile (Appendix D.4).

5 Experiments

We evaluate FlashBoB-2, whose kernels are written in CuTe DSL (NVIDIA, 2025), on an H100 and a B200 GPU. The baseline is FlashBoB's released eighteen-matmul Triton kernel, autotuned (Givans et al., 2026). FlashAttention-3 and -4's backward passes serve as utilization references. Inputs are BF16 with FP32 accumulation. As in FlashAttention-2 (Dao, 2024), each matmul counts $2d$ FLOPs per unmasked attention entry, and each kernel is credited with its own matmul count (15 for FlashBoB-2, 18 for FlashBoB, 5 for first-order backward, 6 for the fused JVP). *Utilization* is these FLOPs per second over the dense BF16 peak, as in FlashAttention-2 and -3; crediting FlashBoB with its 18 matmuls raises its utilization. Every timed configuration first passes a correctness check: all four outputs against full FP64 for $N \le 512$, and at long context sampled rows of $\widetilde{d}(Q)$ and $\widetilde{d}(dO)$ against FP64 and full tensors against another implementation. In an H100 study at $d = 128$, sampled FP64 rows of every output stay within 0.0081 relative ℓ_2 error, except $\widetilde{d}(Q)$ at 4K (0.033): one sampled row has exact $\widetilde{d}(Q) = 0$, and without it the error is 0.0048. A Sophia-H (Liu et al., 2024a) training run ends within 0.005 of the evaluation loss of PyTorch's double backward (Appendix F). The BoB sweep reports medians of three or four runs (protocol: Appendix E).

5.1 BoB Speed and Utilization

On H100, FlashBoB-2 runs three times the matmuls of first-order backward at the same utilization. From 4K tokens, FlashBoB-2 sustains 62–64% of peak at $d = 128$ and 54–57% at $d = 64$. FlashAttention-3's backward sustains 54–62% and 48–56%, respectively (Figure 3). FlashBoB-2 runs more matmuls per exponential (15 and two per tile, against five and one), which makes high

Table 1: **Where the speedup over FlashBoB comes from.** Complete BoB at $d = 128$ unless noted. (a) The speedup is split into two steps: the fifteen-matmul computation, written in Triton like FlashBoB, then our CuTe DSL kernels. (b) Each row replaces one of our design choices with an alternative and reports how many times slower the alternative is. In rows 3, 4, and 6 the rest of the kernel is unchanged, and row 5 was measured on an earlier kernel; the alternatives in rows 1–2 need kernels of their own. Protocols: Appendix G.2.

(a) Step	H100 ($N = 2$K–32K)	B200 ($N = 2$K–256K)
FlashBoB (18 matmuls, Triton) → 15 matmuls in Triton	1.11–1.17×	1.04–1.07×
15 matmuls in Triton → FlashBoB-2 (CuTe DSL)	1.85–2.69×	3.4–4.0×

(b)	Our choice	Alternative	Setting	Slower by
1	$5 + 10$ split	$7 + 9$ split	H100, 2K–16K	1.24–1.30×
2	15 matmuls, tiles recomputed	12–14, tiles in HBM (Triton)	H100, d=64, 2K–32K	2.24–2.65×
3	One barrier per tile	One for all three tiles	H100, 4K, 16K	1.04–1.05×
4	Producer issues loads	A consumer warpgroup	H100, row pass, 4K, 16K	1.02–1.03×
5	Two warpgroup roles	One warpgroup for both roles	H100, $d = 64$	1.24×
6	One MMA warp	Three MMA warps	B200, 4K, 16K	1.03–1.05×

utilization easier. FlashBoB sustains 25–28% at $d = 128$. FlashBoB-2 is 2.7–3.0× faster than FlashBoB at $d = 128$ and 1.5–1.9× at $d = 64$; at $N = 16$K and $d = 128$ it takes 40.9 ms against FlashBoB's 112 ms. PyTorch's double backward, which runs in FP32 and stores $N \times N$ intermediates, is 30–52× slower and runs out of memory at 8K tokens on H100 and 16K on B200 (Appendix G.1.2). The utilization holds to 256K tokens (Appendix G). At 16K and $d = 128$, Tensor Cores are busy in 94% of row-pass and 75% of column-pass cycles (Appendix G.1.4).

On B200, the speedup is larger, but a gap to first-order backward remains. At $d = 128$, FlashBoB-2 is 3.6–4.2× faster than FlashBoB, which sustains 11–13% of peak. At $d = 64$ the speedup is 2.6–3.0×. From 4K tokens at $d = 128$, FlashBoB-2 sustains 41–46% of peak, against 41–54% for FlashAttention-4. Two measurements describe the gap. First, at 16K our column pass is not HBM-bound, and its Tensor Cores are busy in 52% of cycles. Its most common stall mostly waits for an S, F, or $\widetilde{d}(Q)^\mathsf{T}$ matmul before warps can form the next tile. Per-instruction counters show no bank conflict (Appendix G.1.4). Second, a 128×64 `tcgen05.mma` issues at 66% of peak with both operands in shared memory, against 99% with the left operand in TMEM (Appendix D.3). With TMEM full, several column-pass matmuls of this shape read both operands from shared memory.

5.2 Ablation: Sources of the Speedup

The CuTe DSL kernels contribute more than the matmul count (Table 1a). In Triton, the fifteen-matmul computation is 1.11–1.17× faster than FlashBoB at $d = 128$ (1.00–1.07× at $d = 64$), below the $18/15 = 1.2$ that the matmul count predicts. Running the same fifteen matmuls in our CuTe DSL kernels gives a further 1.85–2.69× on H100. This step measures the kernels as a whole, not single scheduling choices, and we have no equally tuned eighteen-matmul CuTe DSL kernel. On B200, the computation gives 1.04–1.07× and the CuTe DSL kernels 3.4–4.0×. At $d = 128$ on H100, 80–83% of the CuTe DSL kernels' gain is in the column pass (Appendix G.2.1).

Each schedule choice beats its alternative (Table 1b, rows 3–6). On H100, one barrier for all three tiles instead of one per tile costs 4.3–5.0%, and issuing next-tile loads from a consumer warpgroup instead of the producer slows the row pass by 1.7–2.7%. Doing construction and accumulation in one warpgroup, which fits in registers only at $d = 64$ and only with serialized matmuls, costs 24% there (on an earlier kernel). On B200, three MMA warps instead of one cost 3.0–4.8%.

No alternative split is faster in our implementations (Table 1b, rows 1–2). A CuTe DSL kernel with a $7 + 9$ split (seven row-pass and nine column-pass matmuls) runs its column pass 5% faster, but its row pass is 1.9–2.2× slower: the extra row-pass work leaves registers for only one query block per thread block. This comparison therefore measures the split together with the schedule it allows. The twelve- to fourteen-matmul variants take 2.24–2.65× our time (H100, $d = 64$). They lose even to the fifteen-matmul Triton kernel, which takes 1.14–1.65× our time.

Table 2: **The fused JVP is faster than both baselines at every length.** O and $\dot{O}$; batch 2, 24 heads, $d = 128$. Baselines: rCM (Zheng et al., 2026b), `jvp_flash_attention` (Morehead, 2025) (`jvp_flash`). Peak counts six matmuls; all lengths in Table 22.

	H100				B200			
N	ms	rCM	`jvp_flash`	Peak	ms	rCM	`jvp_flash`	Peak
1K	0.13	1.84×	4.55×	62%	0.10	3.16×	5.45×	35%
4K	1.67	1.98×	5.03×	75%	1.11	3.54×	6.60×	49%
16K	25.9	1.97×	5.44×	77%	15.5	3.60×	7.20×	57%
128K	1,810	1.82×	4.95×	71%	1,152	3.31×	7.25×	49%

Table 3: **BoB's share of the step bounds the step speedup.** B200, median of four runs. s: BoB's (sCM: the JVPs') share of the baseline's timed region; k: its speedup inside the step, timed differently from Section 5.1. Baselines: FlashBoB (GradMem, test-time training (TTT)), rCM (sCM).

Workload	Context	Share s	k	Amdahl	Measured
GradMem, 1B parameters	16K	51.0%	4.34×	1.65×	1.62×
sCM, forward-mode phase	128K	94.3%	3.40×	2.99×	2.97×
TTT, full attention	128K	26.6%	2.91×	1.21×	1.20×
TTT, released 8K window	32K	5.9%	2.74×	1.04×	1.01×

5.3 Attention JVPs

On the attention shape of our sCM workload, the fused kernel beats both Triton baselines at every length (Table 2). On H100, it is 1.78–2.02× faster than rCM's kernel, which runs the same six matmuls, and 4.55–5.44× faster than `jvp_flash_attention`. On B200, the speedups are 3.16–3.67× and 5.45–7.34×. From 4K up, it reaches 70–77% of BF16 peak on H100 and 49–57% on B200. With a causal mask, it is 4.0–5.9× and 5.6–7.6× faster than `jvp_flash` (Appendix G.3).

The fifteen-matmul rewrite alone makes the JVP 1.9–3.3× faster. FlashBoB's row pass computes it with ten matmuls per tile. In Triton with the same autotuner, dropping its five extra matmuls makes the JVP 2.9–3.3× faster on H100 and 1.9–2.2× on B200. Our CuTe DSL row pass is a further 1.9–2.1× and 3.2–3.9× faster, or 5.6–6.9× and 7.0–8.0× in total ($N = 1$K–32K; Table 23).

5.4 Training Steps

Step speedup follows BoB's share of the step. If BoB takes a share s of a baseline step and runs k times faster while the rest of the step is unchanged, Amdahl's law (Amdahl, 1967) predicts a step speedup of $1/((1 - s) + s/k)$. For a 1B-parameter GradMem model (Kuratov et al., 2026) at 16K tokens, BoB takes half of each step, and the step runs 1.62× faster (Table 3), near the predicted 1.65×. In sCM's forward-mode phase at 128K, attention JVPs take 94% of the time, and the phase runs 2.97× faster than with rCM's kernel. In a full-attention stress test of test-time training (TTT) (Tandon et al., 2025) at 128K, BoB takes 27% of the step, and the step runs 1.20× faster. With TTT's released 8K window, BoB takes 6%, and the step runs 1.01× faster (Appendix G.4).

6 Conclusion

In this paper, we introduce FlashBoB-2, which removes redundant work from FlashBoB's computational graph, from eighteen to fifteen matmuls per tile, and develops architecture-specific schedules for Hopper and Blackwell around each GPU's scarcest on-chip resource. At $d = 128$, FlashBoB-2 runs BoB 2.7–3.0× faster than FlashBoB on H100, at first-order backward's utilization, and 3.6–4.2× on B200. Its row pass is also an attention JVP kernel: fused with the forward pass, it is 1.8–2.0× faster than the fastest existing JVP kernel we measured on H100 and 3.2–3.7× on B200.

Limitations and future work. FlashBoB-2 supports BF16 softmax attention with head dimensions 64 and 128 and full, causal, or sliding-window masks. FP8 and FP4 kernels, linear, sparse, and custom-mask attention, and state-space models are exciting future work. In BF16, a query that sees only one key, such as the first query under a causal mask, has an exact $\widetilde{d}(Q)$ of zero, but FlashBoB-2 returns a rounding error there that FlashBoB avoids (Appendix F).

AI USE STATEMENT

AI tools assisted with manuscript editing, algebraic checks, and organization of experimental evidence. The authors are responsible for verifying all claims, results, citations, and source code.

ETHICS STATEMENT

This work contributes GPU kernels for existing training methods. It involves no human subjects, collects or releases no data, and introduces no new model capability; its effect is to make existing second-order and forward-mode training computations faster.

REPRODUCIBILITY STATEMENT

Section 4 specifies the algorithm and its schedules, and Appendix C derives its outputs; Appendix C.3 proves Proposition C.1. Appendix D describes the kernels and their resource use. Appendix E gives the hardware, software, shapes, tuning policy, timing boundaries, and FLOP convention, and Appendix F gives the numerical checks. The supplementary material contains the kernels, the benchmark and validation scripts, and the measurement records behind every reported number.

REFERENCES

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, pp. 483–485, Atlantic City, New Jersey, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

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)*, pp. 1–11. ACM, 2011. doi: 10.1145/2063384.2063400. URL https://doi.org/10.1145/2063384.2063400.

Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging warp specialization for high performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 119–130. ACM, 2014. doi: 10.1145/2555243.2555258. URL https://doi.org/10.1145/2555243.2555258.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. URL https://jmlr.org/papers/v18/17-468.html.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020. URL https://arxiv.org/abs/2004.05150.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs. Software, 2018. URL http://github.com/jax-ml/jax.

Dan Andrei Calian, Greg Farquhar, Iurii Kemaev, Luisa Zintgraf, Matteo Hessel, Jeremy Shar, Junhyuk Oh, András György, Tom Schaul, Jeff Dean, Hado van Hasselt, and David Silver. DataRater: Meta-learned dataset curation. In *Advances in Neural Information Processing Systems*, volume 38, pp. 20060–20092, 2025. doi: 10.52202/085713-0677. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/1cfb3b066461f1c542be5242a622ce4b-Abstract-Conference.html.

Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. Tawa: Automatic warp specialization for modern GPUs with asynchronous references. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 255–267, 2026a. doi: 10.1109/CGO68049.2026.11394849. URL https://doi.org/10.1109/CGO68049.2026.11394849.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. URL https://arxiv.org/abs/1604.06174.

Yang Chen, Xiaoguang Ma, and Bin Zhao. Mean-flow based one-step vision-language-action. *arXiv preprint arXiv:2603.01469*, 2026b. URL https://arxiv.org/abs/2603.01469.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. URL https://arxiv.org/abs/1410.0759.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations (ICLR)*, pp. 35549–35562, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/98ed250b203d1ac6b24bbcf263e3d4a7-Abstract-Conference.html.

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*, volume 35, pp. 16344–16359, 2022. doi: 10.52202/068431-1189. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex-Attention: A programming model for generating fused attention variants. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/61a9278dfef5f871b5e472389f8d6fa1-Abstract-Conference.html.

Harris Drucker and Yann Le Cun. Improving generalization performance using double backpropagation. *IEEE Transactions on Neural Networks*, 3(6):991–997, 1992. doi: 10.1109/72.165600. URL https://ieeexplore.ieee.org/document/165600.

Logan Engstrom and Axel Feldmann. Flashback: A FlashAttention backwards-over-backwards. GitHub repository, 2025. URL https://github.com/lengstrom/flashback.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135. PMLR, 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

FlashAttention contributors. [CuTe] speed up SM100 hdim-64 bwd: dedicated P/dS TMEM slots + per-warp signaling. GitHub pull request #2804, Dao-AILab/flash-attention, 2026. URL https://github.com/Dao-AILab/flash-attention/pull/2804. Merged September 9, 2026, merge commit d86dabd.

Zhengyang Geng, Mingyang Deng, Xingjian Bai, Zico Kolter, and Kaiming He. Mean flows for one-step generative modeling. In *Advances in Neural Information Processing Systems*, volume 38, pp. 75460–75482, 2025. doi: 10.52202/085713-2534. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/6d13e085b79d454da5910e4ca82a3d9d-Abstract-Conference.html.

Anthony Givans, Michael Crawshaw, and Mingrui Liu. FlashBoB: I/O-efficient exact backward-over-backward for softmax attention. *arXiv preprint arXiv:2609.24089*, 2026. URL https://arxiv.org/abs/2609.24089.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics (SIAM), second edition, 2008. ISBN 978-0-89871-659-7. doi: 10.1137/1.9780898717761. URL https://epubs.siam.org/doi/book/10.1137/1.9780898717761.

Roger Grosse, Juhan Bae, Cem Anil, Nelson Elhage, Alex Tamkin, Amirhossein Tajdini, Benoit Steiner, Dustin Li, Esin Durmus, Ethan Perez, Evan Hubinger, Kamilė Lukošiūtė, Karina Nguyen, Nicholas Joseph, Sam McCandlish, Jared Kaplan, and Samuel R. Bowman. Studying large language model generalization with influence functions. *arXiv preprint arXiv:2308.03296*, 2023. URL https://arxiv.org/abs/2308.03296.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *Conference on Language Modeling*, 2024.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5767–5777, 2017. URL https://papers.nips.cc/paper_files/paper/2017/hash/892c3b1c6dccd52936e27cbd0ff683d6-Abstract.html.

Zheyuan Hu, Chieh-Hsin Lai, Ge Wu, Yuki Mitsufuji, and Stefano Ermon. MeanFlow transformers with representation autoencoders. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 25709–25718, June 2026. URL https://openaccess.thecvf.com/content/CVPR2026/html/Hu_MeanFlow_Transformers_with_Representation_Autoencoders_CVPR_2026_paper.html.

M. F. Hutchinson. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics – Simulation and Computation*, 18(3):1059–1076, 1989. doi: 10.1080/03610918908812806. URL https://doi.org/10.1080/03610918908812806.

Chaokang Jiang, Desen Zhou, Jiuming Liu, and Kevin Li Sun. VectorWorld: Efficient streaming world model via diffusion flow on vector graphs. *arXiv preprint arXiv:2603.17652*, 2026. URL https://arxiv.org/abs/2603.17652.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5156–5165. PMLR, 2020. URL https://proceedings.mlr.press/v119/katharopoulos20a.html.

Iurii Kemaev, Dan A. Calian, Luisa M. Zintgraf, Gregory Farquhar, and Hado van Hasselt. Scalable meta-learning via mixed-mode differentiation. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 29687–29705. PMLR, 2025. URL https://proceedings.mlr.press/v267/kemaev25a.html.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1885–1894. PMLR, 2017. URL https://proceedings.mlr.press/v70/koh17a.html.

Yuri Kuratov, Matvey Kairov, Aydar Bulatov, Ivan Rodkin, and Mikhail Burtsev. GradMem: Learning to write context into memory with test-time gradient descent. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://openreview.net/forum?id=hoLdhnkP0P.

Xiquan Li, Junxi Liu, Yuzhe Liang, Zhikang Niu, Wenxi Chen, and Xie Chen. MeanAudio: Fast and faithful text-to-audio generation with mean flows. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14378–14393, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.654. URL https://aclanthology.org/2026.acl-long.654/.

Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. In *The Twelfth International Conference on Learning Representations (ICLR)*, pp. 1621–1650, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/06960915ba8674c7a898ec0b472b80ff-Abstract-Conference.html.

Tian Yu Liu, Aditya Golatkar, and Stefano Soatto. Tangent transformers for composition, privacy and removal. In *The Twelfth International Conference on Learning Representations (ICLR)*, pp. 28508–28525, 2024b. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/7adf155d85da5d2ef23c307f999b1f98-Abstract-Conference.html.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In Silvia Chiappa and Roberto Calandra (eds.), *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pp. 1540–1552. PMLR, 2020. URL https://proceedings.mlr.press/v108/lorraine20a.html.

Cheng Lu and Yang Song. Simplifying, stabilizing and scaling continuous-time consistency models. In *The Thirteenth International Conference on Learning Representations (ICLR)*, pp. 50611–50649, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/7e9c2053258b1bdd32ff2654802cd594-Abstract-Conference.html.

Tianze Luo, Haotian Yuan, and Zhuang Liu. SoFlow: Solution flow models for one-step generative modeling. In *The Fourteenth International Conference on Learning Representations (ICLR)*, pp. 101623–101641, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/a654298de44ec95dec28aa37e8d0af51-Abstract-Conference.html.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2113–2122, Lille, France, 2015. PMLR. URL https://proceedings.mlr.press/v37/maclaurin15.html.

Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for GANs do actually converge? In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 3481–3490. PMLR, 2018. URL https://proceedings.mlr.press/v80/mescheder18a.html.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018. URL https://arxiv.org/abs/1805.02867.

Alex Morehead. JVP flash attention. Software, September 2025. URL https://github.com/amorehead/jvp_flash_attention.

NVIDIA. CuTe DSL. CUTLASS 4 documentation, 2025. URL https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html.

NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Architecture*. NVIDIA Corporation, 2022. URL https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c. Whitepaper.

NVIDIA Corporation. *Parallel Thread Execution ISA Version 9.4*. NVIDIA Corporation, 9.4 edition, 2026. URL https://docs.nvidia.com/cuda/parallel-thread-execution/.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994. doi: 10.1162/neco.1994.6.1.147. URL https://direct.mit.edu/neco/article/6/1/147/5766/Fast-Exact-Multiplication-by-the-Hessian.

Yansong Peng, Kai Zhu, Yu Liu, Pingyu Wu, Hebei Li, Xiaoyan Sun, and Feng Wu. FACM: Flow-anchored consistency models. In *The Fourteenth International Conference on Learning Representations (ICLR)*, pp. 7520–7543, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0d0dac08f4199f0c348dd2feace0305a-Abstract-Conference.html.

Markus N. Rabe and Charles Staats. Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021. URL https://arxiv.org/abs/2112.05682.

Danyal Rehman, Tara Akhound-Sadegh, Artem Gazizov, Yoshua Bengio, and Alexander Tong. FALCON: Few-step accurate likelihoods for continuous flows. In *The Fourteenth International Conference on Learning Representations (ICLR)*, pp. 70449–70473, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/72bcad2c3fe7625fc233fabe84a6d7ef-Abstract-Conference.html.

Barna Saha and Christopher Ye. I/O complexity of attention, or how optimal is FlashAttention? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 43024–43042. PMLR, 2024. URL https://proceedings.mlr.press/v235/saha24a.html.

Andrea Schioppa, Polina Zablotskaia, David Vilar, and Artem Sokolov. Scaling up influence functions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 8179–8186, 2022. doi: 10.1609/aaai.v36i8.20791. URL https://ojs.aaai.org/index.php/AAAI/article/view/20791.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems*, volume 37, pp. 68658–68685, 2024. doi: 10.52202/079017-2193. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/7ede97c3e082c6df10a8d6103a2eebd2-Abstract-Conference.html.

Rupanshu Soi, Rohan Yadav, Fredrik Kjolstad, Alex Aiken, Maryam Mehri Dehnavi, Michael Garland, and Michael Bauer. Optimal software pipelining and warp specialization for tensor core GPUs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 1875–1890, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/soi.

Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. Consistency models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 32211–32252. PMLR, 2023. URL https://proceedings.mlr.press/v202/song23a.html.

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Daniel Y. Fu, and Christopher Ré. ThunderKittens: Simple, fast, and *Adorable* kernels. In *The Thirteenth International Conference on Learning Representations (ICLR)*, pp. 2097–2130, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/05dc08730e32441edff52b0fa6caab5f-Abstract-Conference.html.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. Learning to (learn at test time): RNNs with expressive hidden states. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 57503–57522. PMLR, 2025. URL https://proceedings.mlr.press/v267/sun25h.html.

Arnuv Tandon, Karan Dalal, Xinhao Li, Daniel Koceja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, Carlos Guestrin, Jed McCaleb, Yejin Choi, and Yu Sun. End-to-end test-time training for long context. *arXiv preprint arXiv:2512.23675*, 2025. URL https://arxiv.org/abs/2512.23675.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS. Software, January 2023. URL https://github.com/NVIDIA/cutlass/tree/v3.0.0.

Philippe Tillet, H. T. 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 (MAPL)*, pp. 10–19. ACM, 2019. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5998–6008, 2017. URL https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W. Mahoney. ADAHESSIAN: An adaptive second order optimizer for machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 10665–10673, 2021. doi: 10.1609/aaai.v35i12.17275. URL https://ojs.aaai.org/index.php/AAAI/article/view/17275.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 912–926, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/ae8b0b5838ba510daff1198474e7b984-Abstract-Conference.html.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*, volume 33, pp. 17283–17297, 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html.

Kaiwen Zheng, Guande He, Min Zhao, Jintao Zhang, Huayu Chen, Jianfei Chen, Chen-Hsuan Lin, Ming-Yu Liu, Jun Zhu, and Qianli Ma. Causal-rCM: A unified teacher-forcing and self-forcing open recipe for autoregressive diffusion distillation in streaming video generation and interactive world models. *arXiv preprint arXiv:2606.25473*, 2026a. URL https://arxiv.org/abs/2606.25473.

Kaiwen Zheng, Yuji Wang, Qianli Ma, Huayu Chen, Jintao Zhang, Yogesh Balaji, Jianfei Chen, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. Large scale diffusion distillation via score-regularized continuous-time consistency. In *The Fourteenth International Conference on Learning Representations (ICLR)*, pp. 2582–2603, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0534abc9e6db91683d82186ef0d68202-Abstract-Conference.html.

Linqi Zhou, Mathias Parger, Ayaan Haque, and Jiaming Song. Terminal velocity matching. In *The Fourteenth International Conference on Learning Representations (ICLR)*, pp. 87506–87530, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/8c9b07a260b1e581a5378a3329d92359-Abstract-Conference.html.

A NOTATION

Table 4: **Symbols used in the paper.** Every $N \times N$ quantity is formed one $B_r \times B_c$ tile at a time and is never stored.

Symbol	Meaning
$Q, K, V \in \mathbb{R}^{N\times d}$; O	queries, keys, values, and output of one attention head; $\tau = 1/\sqrt{d}$
B_r, B_c	query and key block sizes; a *tile* is the $B_r \times B_c$ part of an $N \times N$ quantity for one query block and one key block
S, P, L_i	scores, probabilities, and log-sum-exp, with $P_{ij} = \exp(S_{ij} - L_i)$
dO; dQ, dK, dV	output gradient and first-order input gradients
dP, D_i, X, dS	$dP = dO\,V^\mathsf{T}$, $D_i = do_i^\mathsf{T} o_i$, $X = dP - D$, and $dS = P \odot X$
U_Q, U_K, U_V	tangents of (Q, K, V) for the JVP; equally, the upstream gradients of (dQ, dK, dV) for BoB, as in FlashBoB
$\dot{Y}$	derivative of Y along (U_Q, U_K, U_V) with dO held fixed
$\widetilde{d}(Y) = \partial\Phi/\partial Y$	BoB gradients; the outputs are $\widetilde{d}(Q), \widetilde{d}(K), \widetilde{d}(V)$, and $\widetilde{d}(dO) = \dot{O}$
F, C	$F = \dot{S} = \tau(U_Q K^\mathsf{T} + Q U_K^\mathsf{T})$ and $C = \dot{dP} = dO\,U_V^\mathsf{T}$
α_i, E_i°	the two row statistics the row pass produces (the other two are L_i and D_i): $\alpha_i = \dot{L}_i = \sum_j P_{ij} F_{ij}$ and $E_i^\circ = \dot{D}_i = do_i^\mathsf{T} \widetilde{d}(dO)_i$
E_i	FlashBoB's row statistic in place of E_i°, $E_i = E_i^\circ - \alpha_i D_i$ (Appendix C.2)
h	$h = F - \alpha$
$dS, \widetilde{d}(dP), \widetilde{d}(S)$	the three intermediate tiles of the column pass; $\widetilde{d}(dP) = P \odot h = \dot{P}$ and $\widetilde{d}(S) = P \odot (h \odot X + C - E^\circ) = \dot{dS}$
$[A \mid B]$	horizontal concatenation, e.g. $[dP \mid C] = dO\,[V^\mathsf{T} \mid U_V^\mathsf{T}]$

B METHODS THAT USE ATTENTION JVPS OR BOB

Many training methods need derivatives of attention beyond the standard backward pass. Second derivatives, which in attention models require BoB, arise in meta-learning and hyperparameter optimization (Finn et al., 2017; Maclaurin et al., 2015; Lorraine et al., 2020; Kemaev et al., 2025), data curation (Calian et al., 2025), test-time training (Sun et al., 2025; Tandon et al., 2025), gradient-written memory (Kuratov et al., 2026), gradient penalties (Gulrajani et al., 2017; Mescheder et al., 2018), influence functions (Koh & Liang, 2017; Schioppa et al., 2022), and optimizers that estimate the Hessian diagonal with Hutchinson's estimator (Hutchinson, 1989; Yao et al., 2021; Liu et al., 2024a); Grosse et al. (2023) instead approximate the Hessian to scale influence functions to language models. Attention JVPs train continuous-time consistency models (Song et al., 2023; Lu & Song, 2025; Zheng et al., 2026b; Peng et al., 2026), mean-flow and flow-map models for images, audio, molecules, and driving scenes (Geng et al., 2025; Li et al., 2026; Rehman et al., 2026; Jiang et al., 2026), and linearized transformers (Liu et al., 2024b). Table 5 lists the methods whose papers state how they obtain these derivatives through attention.

C DERIVATION AND PASS SPLIT

This appendix derives the four BoB outputs in forward mode and checks them against FlashBoB's reverse-mode derivation. It then proves the matmul count of direct two-pass splits, shows that the column pass needs both α and E°, and explains why the second pass holds key blocks. The last two subsections cover variants that use fewer matmuls and other exact constructions.

Table 5: **Several recent methods build their own attention kernels for JVPs or BoB, and others change their loss to avoid the JVP.** How each method obtains these derivatives, as its paper states.

Method	Derivative	What the paper states
sCM (Lu & Song, 2025)	JVP	FlashAttention does not compute the JVP; proposes a FlashAttention-style JVP
rCM (Zheng et al., 2026b)	JVP	builds a parallelism-compatible FlashAttention-2 JVP kernel
Causal-rCM (Zheng et al., 2026a)	JVP	builds a custom-mask FlashAttention-2 JVP kernel
FACM (Peng et al., 2026)	JVP	introduces a Chain-JVP to work around incompatibilities with fully sharded data parallel (FSDP) training
MeanFlow (Geng et al., 2025)	JVP	computes its training target with a framework JVP
SoFlow (Luo et al., 2026)	avoids JVP	designs its loss so that it does not require a JVP
MeanFlow-RAE (Hu et al., 2026)	avoids JVP	replaces the JVP with a finite-difference approximation
TVM (Zhou et al., 2026)	JVP and its backward	builds a fused attention kernel that supports backward passes through the JVP
GradMem (Kuratov et al., 2026)	BoB	needs backward-over-backward through attention, which FlashAttention and PyTorch's `scaled_dot_product_attention` (SDPA) do not support
TTT-E2E (Tandon et al., 2025)	BoB	differentiates through a gradient step (gradients of gradients)
Sophia-H (Liu et al., 2024a)	HVP	estimates the Hessian diagonal from Hessian–vector products

C.1 Forward-mode derivation

The forward-mode derivation is shorter than the reverse-mode one because the BoB outputs are directional derivatives of the first-order outputs.

Write $x = (Q, K, V)$ and $U = (U_Q, U_K, U_V)$. The first-order outputs are $(dQ, dK, dV) = J_f(x)^\mathsf{T} dO = \sum_k dO_k \nabla f_k(x)$, summed over the entries k of O. Then

$$\Phi(x) = \sum_k dO_k \, \langle \nabla f_k(x), U \rangle, \qquad \nabla_x \Phi = \sum_k dO_k \, H_{f_k}(x) \, U, \tag{7}$$

where H_{f_k} is the Hessian of the scalar f_k. Each H_{f_k} is symmetric, so this is also the derivative of $J_f(x)^\mathsf{T} dO$ along U with dO held fixed:

$$(\widetilde{d}(Q), \widetilde{d}(K), \widetilde{d}(V)) = \frac{d}{d\epsilon}\Big|_{\epsilon=0} (dQ, dK, dV)\big(x + \epsilon U,\, dO\big). \tag{8}$$

Together with $\widetilde{d}(dO) = \dot{O}$ (Section 3), every BoB output is the derivative of a first-order quantity along U with dO fixed.

Forward pass. Differentiating the forward pass gives the row-pass accumulation. With $S = \tau Q K^\mathsf{T}$ (the mask is constant), $P_{ij} = \exp(S_{ij} - L_i)$, and $O = PV$,

$$\dot{S} = \tau(U_Q K^\mathsf{T} + Q U_K^\mathsf{T}) = F, \tag{9}$$

$$\dot{L}_i = \textstyle\sum_j P_{ij} \dot{S}_{ij} = \alpha_i, \tag{10}$$

$$\dot{P}_{ij} = P_{ij}(\dot{S}_{ij} - \dot{L}_i) = P_{ij} h_{ij}, \tag{11}$$

$$\dot{O} = \dot{P}\, V + P\, U_V = (P \odot h)V + P U_V. \tag{12}$$

Expanding $P \odot h = P \odot F - \mathrm{Diag}(\alpha) P$ and using $\mathrm{Diag}(\alpha) PV = \mathrm{Diag}(\alpha) O$ turns the last line into

$$\widetilde{d}(dO) = P\, U_V + (P \odot F)\, V - \mathrm{Diag}(\alpha)\, O, \qquad \alpha = \mathrm{rowsum}(P \odot F), \tag{13}$$

which takes five matmuls per tile: S, the two terms of F, $P\, U_V$, and $(P \odot F)V$.

Backward pass. Differentiating the first-order backward pass gives E° and the column-pass tiles. With dO fixed, $dP = dO\,V^\mathsf{T}$, $D_i = do_i^\mathsf{T} o_i$, $X = dP - D$, and $dS = P \odot X$,

$$\dot{dP} = dO\,U_V^\mathsf{T} = C, \tag{14}$$

$$\dot{D}_i = do_i^\mathsf{T} \dot{o}_i, \tag{15}$$

$$\dot{dS}_{ij} = \dot{P}_{ij} X_{ij} + P_{ij} \dot{X}_{ij} = P_{ij}\big(h_{ij} X_{ij} + C_{ij} - \dot{D}_i\big). \tag{16}$$

Equation (15) with $\dot{o}_i = \widetilde{d}(dO)_i$ is Equation (4), so $E_i^\circ = \dot{D}_i$.

A row pass that does not use Equation (4) must evaluate E° tile by tile. Expanding Equation (15) with $\dot{o}_i = \sum_j P_{ij} h_{ij} v_j + \sum_j P_{ij} u_{V,j}$ gives this tile-wise form:

$$E_i^\circ = \textstyle\sum_j P_{ij} h_{ij}\, dP_{ij} + \sum_j P_{ij} C_{ij} = \sum_j P_{ij}\big(C_{ij} + F_{ij} X_{ij}\big). \tag{17}$$

The second equality first uses $\sum_j P_{ij} h_{ij} = 0$ to replace dP by X, and then uses $\sum_j P_{ij} X_{ij} = 0$ to replace h by F. The tile-wise form needs the tiles of dP and C.

The remaining outputs follow directly. $\widetilde{d}(dP) = P \odot h = \dot{P}$, and Equation (16) gives $\widetilde{d}(S) = \dot{dS}$, which is Equation (5). Differentiating $dQ = \tau\, dS\, K$, $dK = \tau\, dS^\mathsf{T} Q$, and $dV = P^\mathsf{T} dO$ along U gives Equation (6).

C.2 Agreement with FlashBoB's reverse-mode derivation

FlashBoB's reverse-mode derivation gives the same column-pass tiles. FlashBoB derives $\widetilde{d}(P) = C + F \odot X - \mathrm{Diag}(\alpha)\, dP$, $E_i = \sum_j P_{ij} \widetilde{d}(P)_{ij}$, and $\widetilde{d}(S) = P \odot (\widetilde{d}(P) - E)$. Substituting $F = h + \alpha$ and $dP = X + D$ gives

$$\widetilde{d}(P)_{ij} = C_{ij} + h_{ij} X_{ij} - \alpha_i D_i, \tag{18}$$

so $E_i = E_i^\circ - \alpha_i D_i$ and $\widetilde{d}(P)_{ij} - E_i = C_{ij} + h_{ij} X_{ij} - E_i^\circ$. Multiplying by P_{ij} recovers Equation (5). The $\alpha_i D_i$ terms cancel. The column pass therefore needs D only inside X, and α and E° both enter as scalar subtractions in one elementwise step.

C.3 Direct two-pass splits

The fifteen-matmul count is the minimum for splits that evaluate the BoB equations as written. This subsection defines that class and proves the count.

Setting. We count matmuls as tile products: a $B_r \times d$ by $d \times B_c$ product or its transpose. A fused instruction counts each product it performs. A *direct split* evaluates Equations (4), (13), (5), and (6) as written, under three rules:

- the row pass completes $\widetilde{d}(dO)$, α, and E° from the saved O, L, and D;
- only per-row values and $N \times d$ outputs cross between the passes, so no tile does;
- each output matmul of Equation (6) runs whole in one pass.

Proposition C.1 (Direct two-pass splits)**.** *Among direct splits, the fewest matmuls per tile are* $5+10$*, with all five output matmuls in the column pass. Moving one or both* dS *matmuls into the row pass gives* $7 + 9$ *or* $8 + 8$*.*

Proof of Proposition C.1. No tile crosses between the passes, so every tile that a pass reads is formed in that pass.

Row pass. Completing $\widetilde{d}(dO)$ requires $P\,U_V$ and $(P \odot F)V$. Forming P requires S, and forming $P \odot F$ requires both terms of F. Every direct split therefore runs these five matmuls in the row pass. The statistics α and E° need only row reductions and a row dot product, which add no matmul.

Column pass. The $\widetilde{d}(V)$ matmul reads $\widetilde{d}(dP)$, which needs α. The two $\widetilde{d}(S)$ matmuls read $\widetilde{d}(S)$, which needs α and E°. Both statistics are complete only after the row pass, so these three matmuls

run in the column pass. That pass therefore forms $\widetilde{d}(dP)$ and $\widetilde{d}(S)$, and with them P, F, X, and C. This takes the five matmuls S, the two terms of F, dP, and C. With its three output matmuls, the column pass runs at least eight.

The two dS matmuls. The tile $dS = P \odot (dP - D)$ needs P and dP, and D is known before either pass. Let $m \in \{0, 1, 2\}$ of the two dS matmuls run in the row pass. If $m > 0$, the row pass also forms dP, one more matmul. The column pass still forms dP for $\widetilde{d}(S)$ and runs the remaining $2-m$ dS matmuls. The two passes therefore run

$$\underbrace{5 + m + \mathbf{1}_{m>0}}_{\text{row pass}} \qquad \text{and} \qquad \underbrace{10 - m}_{\text{column pass}}$$

matmuls per tile, which gives $5 + 10$, $7 + 9$, and $8 + 8$ for $m = 0, 1, 2$. Forming dP in the row pass without moving a dS matmul gives the dominated $6 + 10$. □

FP64 check. An FP64 reference that implements the four splits with tiles matches double-backward automatic differentiation on small causal and non-causal problems to FP64 rounding error.

Scope. The proposition fixes the equations. Evaluations that obtain α and E° through other identities, or that reuse values across tiles, lie outside it. Appendix C.7 gives two such constructions. A minimum count is also not a minimum latency (Section 5.2), because the split decides which schedule fits. The $7 + 9$ row pass lacks the registers to hold two query blocks per thread block, so two warpgroups cannot share one key stream. The complete BoB with the $7 + 9$ split is 24–30% slower (Section 5.2). Both α and E° cross between the passes in every direct split, and Appendix C.4 shows that a column pass that reads only its own blocks and per-row scalars needs at least two scalars per row.

C.4 Necessity of two row statistics

The two row statistics that the row pass produces are sufficient for the column pass and, in a local sense, necessary. Fix an unmasked entry with $p = P_{ij} > 0$ and fixed local values F_{ij}, X_{ij}, C_{ij}. The two tiles that depend on α_i and E_i° are $\widetilde{d}(dP)_{ij} = p(F_{ij} - \alpha_i)$ and $\widetilde{d}(S)_{ij} = p\big((F_{ij} - \alpha_i)X_{ij} + C_{ij} - E_i^\circ\big)$. Their Jacobian with respect to the statistics is nonsingular:

$$\frac{\partial(\widetilde{d}(dP)_{ij}, \widetilde{d}(S)_{ij})}{\partial(\alpha_i, E_i^\circ)} = \begin{pmatrix} -p & 0 \\ -pX_{ij} & -p \end{pmatrix}, \qquad \det = p^2 > 0. \tag{19}$$

The lemma below shows that inputs outside the local blocks can move the two statistics independently, so one scalar per row cannot carry them.

Lemma C.2 (Two per-row scalars are necessary)**.** *Fix an unmasked entry (i, j) and an unmasked key j' in another key block with $P_{ij} > 0$, $P_{ij'} > 0$, $q_i \neq 0$, and $do_i \neq 0$. Suppose a column pass computes $(\widetilde{d}(dP)_{ij}, \widetilde{d}(S)_{ij})$ from the inputs of its own key block and query block and from k scalars per row received from the row pass, and that both the scalars and the column pass's computation depend differentiably on their inputs. Then $k \geq 2$, and (α_i, E_i°) attains $k = 2$.*

Proof. Perturb only the tangents of key j': $u_{K,j'} \mathrel{+}= t\, q_i/(\tau\|q_i\|^2)$ and $u_{V,j'} \mathrel{+}= s\, do_i/\|do_i\|^2$. This adds t to $F_{ij'}$ and s to $C_{ij'}$ and leaves P, X, and every input of the blocks holding j and i unchanged. From $\alpha_i = \sum_\ell P_{i\ell} F_{i\ell}$ and Equation (17),

$$\frac{\partial(\alpha_i, E_i^\circ)}{\partial(t, s)} = P_{ij'} \begin{pmatrix} 1 & 0 \\ X_{ij'} & 1 \end{pmatrix},$$

of rank two. Composed with the Jacobian above, also of rank two, the derivative of the two tiles in (t, s) has rank two. A map that factors through k scalars has rank at most k. □

The lemma counts scalars. A column pass that reads the saved row of $\widetilde{d}(dO)$ could form E° itself, and that vector then counts as part of what it receives. The lemma does not apply to a row with a single unmasked key.

Table 6: **Every fewer-matmul variant we found stores tiles whose size grows with N.** Matmuls per tile in the row pass (R) and column pass (C); F counts as two matmuls. *Stored* is what one query block of B_r rows stores between the passes besides the row statistics. FlashBoB-2 passes only two numbers per row and no $N \times N$ data. *Fits to* is the largest N at which the stored tiles alone fit in the 232,448-byte shared memory of an H100 thread block at $B_r = 64$ in BF16, a necessary condition.

Variant	Matmuls	Total	Stored	Fits to
Store P and F	R: $S, F, PU_V, (P \odot F)V$. C: $dP, C, \widetilde{d}(V), \widetilde{d}(K)(2), \widetilde{d}(Q)(2)$	12	$2B_rN$	$N \leq 908$
Store F	as above, plus S in C	13	B_rN	$N \leq 1816$
Store P	as above, plus F in C	14	B_rN	$N \leq 1816$
FlashBoB-2	R: $S, F, PU_V, (P \odot F)V$. C: $S, F, dP, C, \widetilde{d}(V), \widetilde{d}(K)(2), \widetilde{d}(Q)(2)$	**15**	–	any
Move one dS matmul	as FlashBoB-2, plus dP and one dS matmul in R	$7 + 9 = 16$	–	any
Deferred F	R: $S, F, dP, C, PU_V, H_0V, JK, J^\mathsf{T}Q, H_0^\mathsf{T}dO$. C: S, dP and five corrections	17	–	any
FlashBoB (Triton)	R: $S, F, dP, C, PU_V, (P \odot F)V, (P \odot \widetilde{d}(P)^\circ)K, dS\,U_K, PK$. C: $S, F, dP, C, \widetilde{d}(V), \widetilde{d}(K)(2)$	$10 + 8 = 18$	–	any
FlashBoB (paper)	as FlashBoB (Triton), plus $(P \odot dP)K$ in R	19	–	any

A small example illustrates the lemma. Take one scalar query and two keys with $Q = 1$, $K = (0, 0)$, $V = (0, 2)$, $dO = 1$, $U_Q = 0$, $U_K = (0, u)$, and $U_V = (0, v)$. The two row-pass statistics are $\alpha = u/2$ and $E^\circ = (u + v)/2$. The tiles of the *first* key, whose own inputs do not change, are $\widetilde{d}(dP)_1 = -u/4$ and $\widetilde{d}(S)_1 = -v/4$. A distant key thus changes the two local tiles independently.

C.5 Choice of key blocks for the second pass

The second pass holds key blocks because that leaves one output to be summed across thread blocks. After the row pass, each tile contributes to $\widetilde{d}(Q)$, which sums over key blocks, and to $\widetilde{d}(K)$ and $\widetilde{d}(V)$, which sum over query blocks. A second pass whose thread blocks hold key blocks finishes $\widetilde{d}(K)$ and $\widetilde{d}(V)$ locally and leaves only $\widetilde{d}(Q)$ to be summed across thread blocks. A second pass that held query blocks would finish $\widetilde{d}(Q)$ locally and leave two outputs to be summed. First-order backward makes the same choice for dQ (Dao, 2024).

C.6 Variants with fewer matmuls

Every variant we found that uses fewer matmuls per tile than the direct $5 + 10$ split of Appendix C.3 stores row-pass tiles for the column pass instead of recomputing them. Table 6 lists every variant the paper compares, with the tiles each stores between the passes. This subsection gives the matmul accounting, shows that the stored tiles grow with the unmasked key span, and reports the measured cost.

Matmul accounting. The rows with more than fifteen matmuls use no stored tiles, and two of them need notes. The 17-matmul variant avoids recomputing F in the column pass. With $H_0 = P \odot F$ and $J = P \odot C + F \odot dS$, its row pass accumulates JK, $J^\mathsf{T}Q$, and $H_0^\mathsf{T}dO$ beside $\widetilde{d}(dO)$. Its column pass corrects them with the row statistics through five further matmuls. In the FlashBoB rows, $\widetilde{d}(P)^\circ = C + F \odot X$. The released FlashBoB kernel reads the saved first-order dQ in place of one row-pass matmul, so it runs one matmul fewer than the equations in its paper.

Stored tiles on chip. Stored tiles span whole rows, so their on-chip size grows with N. A variant that stores k matrices of size $N \times N$ holds kB_rN values per query block. For an on-chip budget of M values it fits only up to $N \leq M/(kB_r)$. The last column of Table 6 applies this bound to

an H100 thread block. All three bounds lie below every length in our BoB sweeps. At $N = 16$K, P and F for one block of 64 query rows take 4 MiB, eighteen times an H100 thread block's shared memory.

Stored tiles in HBM. Writing the stored values to HBM instead creates a stored $N \times N$ tensor. The row pass writes kN^2 values per batch and head and the column pass reads them back, or half that under causal packing. At batch 4 and 12 heads in BF16, one dense matrix takes 24 GiB at $N = 16$K. One causally packed matrix takes 12 GiB at 16K, 192 GiB at 64K, and 3 TiB at 256K.

For stored $N \times N$ tensors that must pass through HBM, the transfers alone give a lower bound on the latency. At $N = 16{,}384$, $d = 64$, batch 4, and 12 heads, storing P and F in FP32 writes $51{,}542{,}753{,}280$ bytes and reads them back. At the H100's 3.35 TB/s this traffic takes at least 30.13 ms, even after crediting 1 GiB of the writes and 1 GiB of the reads as never reaching HBM. That is more than the 23.0 ms FlashBoB-2 takes for the whole BoB. The corresponding BF16 bounds are 7.05, 7.05, and 14.74 ms for storing P, F, or both. These bounds do not decide the runtime (Appendix C.6).

Stored tiles grow with the unmasked key span. A fixed number of stored tiles saves a fixed number of matmuls per query block, so its saving per tile shrinks as the query block visits more key tiles. The following result makes this tradeoff exact within the family of stored-tile variants.

Proposition C.3 (Recomputation–storage tradeoff)**.** *Consider a query block that visits T key tiles and variants of the $5 + 10$ computation that avoid recomputation only by storing $p \le T$ row-pass tiles of P and $f \le T$ tiles of F for the column pass, with at most R stored tiles in total. The fewest matmuls for the block are*

$$W_{\min}(T, R) = \begin{cases} 15T - 2R, & 0 \le R \le T, \\ 14T - R, & T \le R \le 2T, \\ 12T, & R \ge 2T. \end{cases}$$

Averaging at most $15 - \delta$ matmuls per tile, for $0 < \delta \le 2$, therefore requires $R \ge \delta T/2$; averaging at most fourteen requires $p + f \ge \lceil T/2 \rceil$.

Proof. A stored F tile saves its two matmuls and a stored P tile saves the S matmul, so $W = 15T - p - 2f$. Under $p + f \le R$ this is smallest when F tiles are stored first: $f = \min(R, T)$, then $p = \min(R - f, T)$. □

Storing R tiles saves at most $2R$ matmuls per query block. By Proposition C.3, saving a fixed number of matmuls per tile therefore needs stored tiles in proportion to the number of unmasked key tiles, so the stored tiles grow with the unmasked key span. Under a sliding window of width W, T is at most $\lceil W/B_c \rceil + 1$ and does not grow with N. Fifteen is thus the fewest matmuls per tile we found without stored tiles that grow with the unmasked key span.

Three remarks limit the scope of the proposition. Storing dP or C tiles does not help, because the $5 + 10$ row pass does not form them: storing one adds a row-pass matmul to save a column-pass matmul. Storing about half of the F tiles is the least storage that averages fourteen. This follows from the accounting; we did not implement such a variant. The stored tiles of Table 6 are BF16, so these variants also form $F - \alpha$ from a rounded F.

Measured cost. The implemented fewer-matmul variants take 2.24–2.65× the latency of FlashBoB-2 from 2K to 32K. They are also slower than the fifteen-matmul Triton kernel. We implemented the 12-, 13-, and 14-matmul variants in Triton with their stored $N \times N$ tensors in HBM. Each received an independent search over 48 row-pass and 60 column-pass configurations (before pruning). We compared them on H100 with the FlashBoB-2 kernel (causal, BF16, batch 4, 12 heads, $d = 64$). The search ran at 4K, and new runs then confirmed the fixed configurations with paired timing. Every variant passes the FP64 check, and its outputs stay within 4.2×10^{-3} relative ℓ_2 of FlashBoB-2 at every confirmed length.

The 12-matmul variant, which stores both P and F, is 13–19% slower than the faster of the other two from 2K to 16K, and it exceeds the GPU's memory at 32K. Up to 16K the stored tensors are

dense BF16. The 12-matmul variant stores two, 48 GiB at 16K with a peak allocation of 51 GiB, and the 13- and 14-matmul variants store one, 24 GiB. At 32K the 13- and 14-matmul variants store one causally packed BF16 matrix, 48 GiB, and would need about 192 GiB at 64K. The 12-matmul variant's two packed matrices, 96 GiB, were rejected before allocation.

Packed storage does not remove the gap. Dense storage doubles the traffic of the stored tensors below 32K, but at 32K, where the 13- and 14-matmul variants use packed storage, they are still $2.54\times$ and $2.24\times$ slower than FlashBoB-2. Neither the matmul count nor the size of the stored tensors orders the variants: the 13-matmul variant is faster than the 14-matmul variant from 4K to 16K and slower at 32K.

Storing the tiles on chip instead runs into register spills. A microbenchmark keeps half of the F values of one 64-row query block live across a row reduction. The kernel stays in registers up to a represented $N = 1024$ (64 KiB of state). It reaches the 255-register limit with 58 spills at $N = 2048$. At $N = 4096$ it spills 2,698 values and takes $11.1\times$ as long as at 2048 for twice the state.

C.7 Other exact constructions

Proposition C.1 covers direct splits of Equations (4), (13), (5), and (6). Two exact constructions outside that class show why its scope matters. We do not use either construction.

α and E° from feature summaries. If the first-order output $dQ = \tau\, dS\, K$ is available, exchanging the sums over keys and features in Equation (17) gives

$$\alpha_i = \tau\big[u_{Q,i}^{\mathsf{T}}(PK)_i + q_i^{\mathsf{T}}(PU_K)_i\big], \qquad E_i^\circ = u_{Q,i}^{\mathsf{T}} dQ_i + \tau\, q_i^{\mathsf{T}}(dS\, U_K)_i + do_i^{\mathsf{T}}(PU_V)_i. \tag{20}$$

A row pass then forms S and dP and accumulates PK, PU_K, $dS\,U_K$, and PU_V, six matmuls, and obtains both statistics without forming F. The column pass recomputes S, F, dP, and C and runs five output matmuls, with $(P \odot F)V$ in place of the $dS\,U_K$ term of $\widetilde{d}(Q)$ that the row pass already holds. The total is $6 + 10 = 16$ matmuls when dQ is saved, and 17 when it must be recomputed.

Recovering one entry per row. For exact attention, $\mathrm{rowsum}(dS) = \mathrm{rowsum}(\widetilde{d}(dP)) = \mathrm{rowsum}(\widetilde{d}(S)) = 0$. The first follows from $D_i = \sum_j P_{ij}\, dP_{ij}$ and $\sum_j P_{ij} = 1$, the second from the definition of α, and the third from Equation (17). A pass that holds query rows can therefore skip one unmasked entry per row and recover its three tile values as the negated sums of the rest. With J unmasked entries and n_q nonempty query rows, this costs $15J - 5n_q$ full-width dot products. The saving is of lower order than the $15J$ total, so statements of fifteen per tile hold to leading order in the number of unmasked entries. On Tensor Core tiles, skipping one entry per row need not remove any issued instruction.

FP64 check. Every variant in Table 6, the sixteen-matmul construction, and the recovery construction match FP64 double-backward automatic differentiation on dense and causal masks to FP64 rounding error.

D Kernel Implementation

This section gives the kernel-level detail behind Sections 4.3–4.5. It first lists both passes with their issue and wait points, then covers the Hopper kernels, the Blackwell kernels, the JVP kernels, the attention variants, and the resource budget.

D.1 Both passes, with issue and wait points

Figure 4 shows how the two passes cover the causal block grid, Figure 5 shows the column pass in full, and Algorithms 2 and 3 list each pass for one block. Two verbs mark the asynchronous matmuls. *Issue* starts a matmul, and *wait* blocks until it completes. Everything between the two overlaps the matmul.

In the column pass, each line names the Hopper warpgroup that executes it: P (producer), C (construction), or A (accumulation). On Blackwell, eight compute warps form the tiles, and at $d = 128$ one MMA warp issues all ten matmuls.

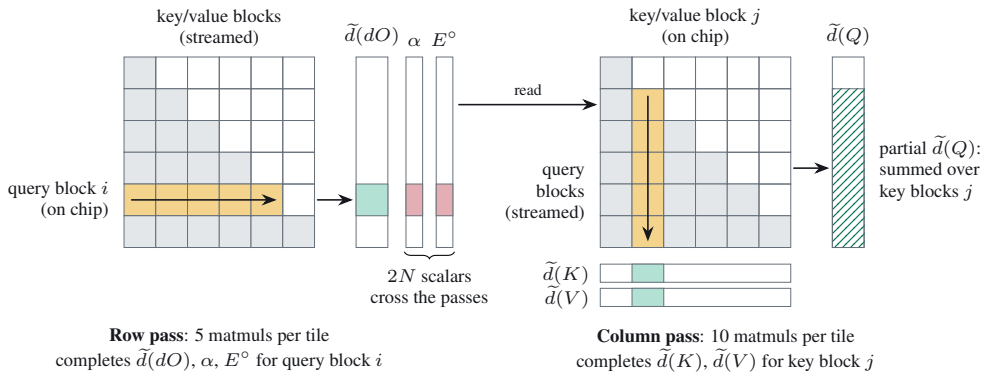

Figure 4: **Only $2N$ scalars cross between the passes.** The row pass holds one query block and sweeps across key blocks. The column pass holds one key block and sweeps down query blocks. It finishes $\widetilde{d}(K)$ and $\widetilde{d}(V)$ for its key block and adds a partial $\widetilde{d}(Q)$, which is summed over key blocks.

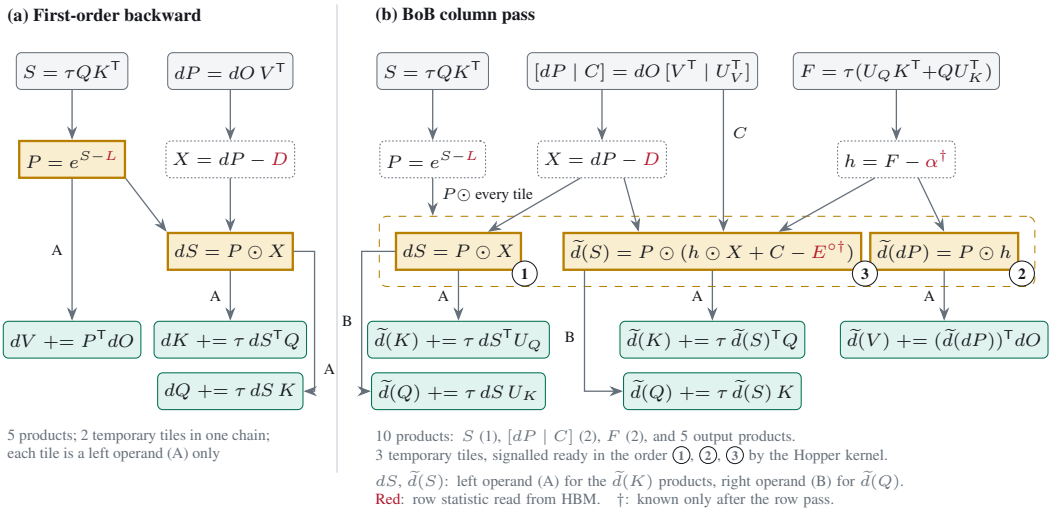

Figure 5: **Each column-pass tile reads row statistics, and two of them exist only after the row pass.** (a) First-order backward. (b) A detailed version of Figure 2, with the elementwise steps and the row statistics each tile reads. Red symbols are row statistics read from HBM, and † marks the two that the row pass produces. The order ①–③ is the order in which the Hopper kernel signals the tiles. A and B are the left and right operand positions in the Blackwell instruction; the B labels refer to the transposed product $\widetilde{d}(Q)^\mathsf{T} = \tau\big(K^\mathsf{T}\widetilde{d}(S)^\mathsf{T} + U_K^\mathsf{T} dS^\mathsf{T}\big)$ that Blackwell computes.

D.2 Hopper

This subsection gives the register, shared-memory, and barrier layout of the Hopper kernels behind Section 4.3; Figure 6 contrasts the column-pass roles with FlashAttention-3's. It covers both passes at $d = 128$, the $\widetilde{d}(Q)$ reduction, long sequences, $d = 64$, and the wait at the end of each iteration.

Row pass at $d = 128$. The row pass runs two consumer warpgroups, each on its own query block. Each thread block has a producer warpgroup that reserves 24 registers per thread and two consumer warpgroups that reserve 240. Each consumer warpgroup holds a 64-row query block, with 64-by-64 FP32 accumulators for S and F and a 64-by-128 FP32 accumulator for $\widetilde{d}(dO)$. The K, U_K and V, U_V streams each have two stages. After Q and U_Q are copied into registers, their shared-memory region is reused, first for dO and O and then for storing $\widetilde{d}(dO)$. Each consumer warpgroup runs

Algorithm 2 Row pass for one query block of B_r rows. Given the saved (O, L), the attention JVP is complete after line 11.

Require: resident Q, U_Q; saved L, O; streamed $K_j, U_{K,j}, V_j, U_{V,j}$ for each key block j with unmasked entries

1: $\widetilde{d}(dO) \leftarrow 0$, $\alpha \leftarrow 0$
2: **for** each key block j **do**
3: wait for the TMA stage $(K_j, U_{K,j})$
4: **issue** $S = \tau Q K_j^\mathsf{T}$, $F_1 = \tau U_Q K_j^\mathsf{T}$, $F_2 = \tau Q U_{K,j}^\mathsf{T}$ ▷ Blackwell: $[S \mid F_2]$ in one instruction
5: **wait** S; $P \leftarrow \exp(S - L)$ ▷ overlaps F_1, F_2
6: wait for the TMA stage $(V_j, U_{V,j})$; **issue** $\widetilde{d}(dO) \mathrel{+}= P\, U_{V,j}$
7: **wait** F_1, F_2; $G \leftarrow P \odot (F_1 + F_2)$; $\alpha \mathrel{+}= \mathrm{rowsum}(G)$ ▷ overlaps $P U_{V,j}$
8: **issue** $\widetilde{d}(dO) \mathrel{+}= G\, V_j$
9: **wait** for all matmuls ▷ Hopper waits here (Appendix D.2); Blackwell runs one block ahead
10: **end for**
11: $\widetilde{d}(dO) \leftarrow \widetilde{d}(dO) - \mathrm{Diag}(\alpha)\, O$; write $\widetilde{d}(dO)$ and α ▷ $\dot{O}$ and $\dot{L}$
12: **BoB only:** load dO; $E^\circ \leftarrow \mathrm{rowdot}(dO, \widetilde{d}(dO))$; write E° ▷ Equation (4)

Algorithm 3 Column pass for one key block of B_c keys. Hopper warpgroups: P producer, C construction, A accumulation.

Require: resident K, U_K, V, U_V; streamed $Q_i, U_{Q,i}, dO_i$ and row statistics $L_i, D_i, \alpha_i, E_i^\circ$ for each query block i with unmasked entries

1: A: $\widetilde{d}(K) \leftarrow 0$, $\widetilde{d}(V) \leftarrow 0$
2: **for** each query block i **do**
3: P: TMA load of $Q_i, U_{Q,i}, dO_i$; C reads the four row statistics from global memory
4: C: **issue** $S = \tau Q_i K^\mathsf{T}$ and $[dP \mid C] = dO_i [V^\mathsf{T} \mid U_V^\mathsf{T}]$; **wait** S; $P \leftarrow \exp(S - L_i)$; **issue** F_1, F_2
5: C: **wait** dP; $X \leftarrow dP - D_i$, $dS \leftarrow P \odot X$; store dS (BF16); **signal** dS
6: A: **wait** dS; **issue** $\widetilde{d}(K) \mathrel{+}= \tau\, dS^\mathsf{T} U_{Q,i}$, $\widetilde{d}(Q)_i = \tau\, dS\, U_K$ ▷ before F is done
7: C: **wait** F_1, F_2; $h \leftarrow F_1 + F_2 - \alpha_i$, $\widetilde{d}(dP) \leftarrow P \odot h$, $\widetilde{d}(S) \leftarrow P \odot (h \odot X + C - E_i^\circ)$
8: C: store $\widetilde{d}(dP)$; **signal** $\widetilde{d}(dP)$; store $\widetilde{d}(S)$; **signal** $\widetilde{d}(S)$
9: A: **wait** $\widetilde{d}(dP)$; **issue** $\widetilde{d}(V) \mathrel{+}= \widetilde{d}(dP)^\mathsf{T} dO_i$ ▷ overlaps the $\widetilde{d}(S)$ store
10: A: **wait** $\widetilde{d}(S)$; **issue** $\widetilde{d}(K) \mathrel{+}= \tau\, \widetilde{d}(S)^\mathsf{T} Q_i$, $\widetilde{d}(Q)_i \mathrel{+}= \tau\, \widetilde{d}(S)\, K$
11: A: **wait** for the two $\widetilde{d}(Q)_i$ matmuls; write $\widetilde{d}(Q)_i$ (FP32) to a slot of a circular buffer; **signal**
12: P: **wait** for that slot; TMA reduce-add into the global FP32 $\widetilde{d}(Q)$
13: **end for**
14: A: write $\widetilde{d}(K)$, $\widetilde{d}(V)$

its own epilogue on a private barrier, which keeps the two independent. Figure 7 shows the two overlaps inside one key block.

Column pass at $d = 128$. The column pass splits its register state across two warpgroups because one warpgroup cannot hold it. The thread block keeps K, U_K, V, U_V in shared memory as eight 64-by-64 panels and streams Q, U_Q, dO through two stages. The construction warpgroup holds a 64-by-128 FP32 accumulator for $[dP \mid C]$ and a 64-by-64 accumulator for S, which it reuses for F. The accumulation warpgroup holds $\widetilde{d}(K)$ and $\widetilde{d}(V)$ as 64-by-128 FP32 accumulators and the partial $\widetilde{d}(Q)$ as two 64-by-64 accumulators. One warpgroup holding both roles would need about 144 registers per thread for the construction state and 192 for the accumulators. We derive these two numbers from the fragment shapes; they do not come from a compiler report.

The register counts in this paper are of three kinds: estimates of live state, register allocations set with `setmaxnreg`, and compiled counts. `setmaxnreg` is the PTX instruction that moves registers between warpgroups at run time. The final kernel allocates 24 registers to the producer, 232 to construction, and 248 to accumulation. The total of 504 is the most a 384-thread block can

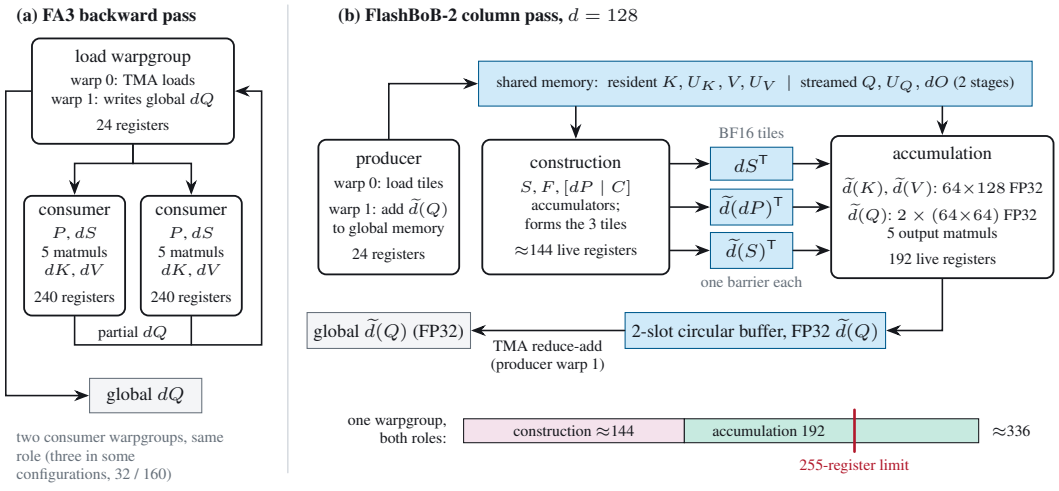

Figure 6: **The Hopper column pass splits its consumer warpgroups because one warpgroup cannot hold both states.** (a) FlashAttention-3's backward pass gives its consumer warpgroups the same work. (b) In the FlashBoB-2 column pass at $d = 128$, construction (≈144 registers per thread) and accumulation (192) together exceed the 255-register limit. Separate warpgroups hold them and exchange three tiles through shared memory, with one barrier per tile.

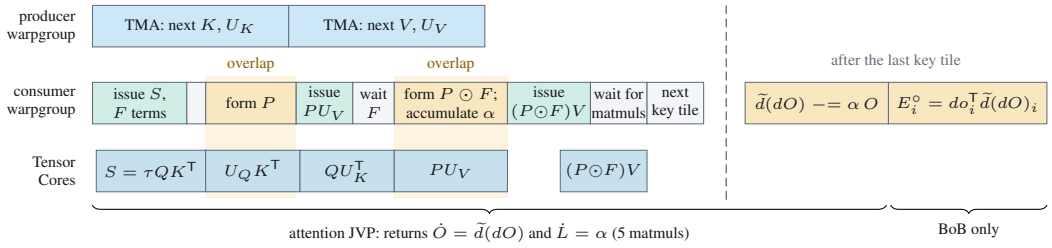

Figure 7: **The Hopper row pass forms tiles while matmuls run.** Forming P overlaps the Tensor Core computation of F, and forming $P \odot F$ overlaps PU_V. Everything under the first brace is the attention JVP given the saved (O, L). The E° epilogue is the only BoB-specific step. Widths are schematic.

hold. The compiled binary launches with 168 registers per thread, 64,512 for 384 threads, which is the same total that `setmaxnreg` then redistributes.

The three tiles pass through separate shared-memory buffers, each with its own barrier (Figure 8). Removing a dependency that the equations do not impose cannot lengthen the best possible schedule, because every schedule that is legal with the dependency remains legal without it. Appendix G.2.3 measures the effect in practice.

$\widetilde{d}(Q)$ reduction. Each partial $\widetilde{d}(Q)$ reaches global memory through a shared-memory buffer and a TMA reduce-add, the fastest of the three paths we measured. The accumulation warpgroup writes each FP32 partial $\widetilde{d}(Q)$ to a two-slot circular buffer in shared memory. A producer warp then adds it to the global FP32 $\widetilde{d}(Q)$ with a TMA reduce-add (Figure 10). This path needs no workspace beyond the FP32 destination. The order of the additions depends on scheduling, so $\widetilde{d}(Q)$ is not bitwise repeatable (Appendix F).

The two alternatives were slower. Adding directly from registers into global memory ran at 0.75–0.89× the speed; we attribute this to L2 atomics. A padded circular buffer emptied row by row ran at 0.52–0.63×; we attribute this to emptying the buffer.

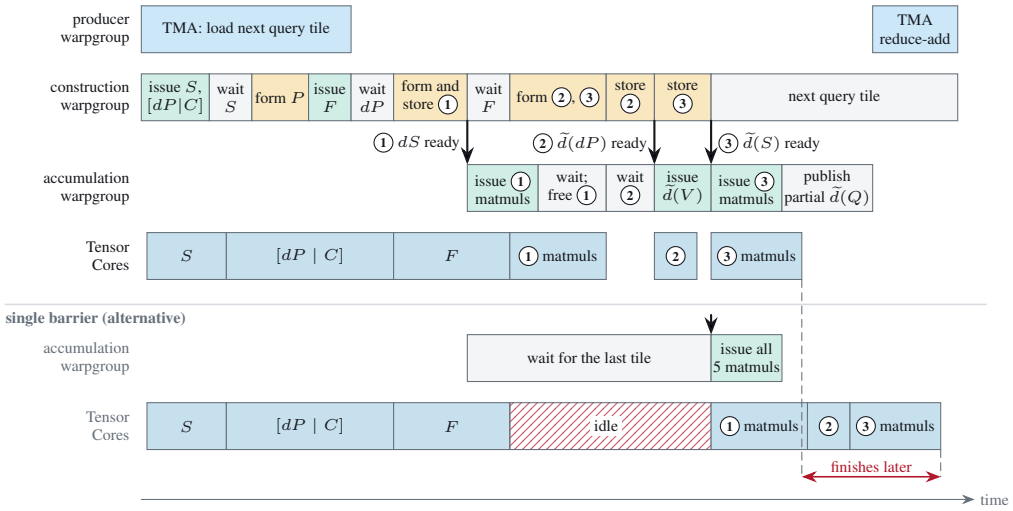

Figure 8: **On Hopper, one barrier per tile lets output matmuls start early.** Top: the column pass for one query block at $d = 128$. The accumulation warpgroup issues the dS matmuls and $\widetilde{d}(V)$ before $\widetilde{d}(S)$ is stored. Bottom: with a single barrier, all five output matmuls wait for the last tile, and the Tensor Cores idle in the hatched interval. Widths are schematic.

Long sequences. At long N, one column-pass launch loses L2 reuse, so both head dimensions split the column pass into several launches. In one long launch, the persistent column-pass thread blocks stop progressing together and stop sharing streamed query tiles in L2. At $N = 256$K and $d = 128$, a single launch has an L2 hit rate of 55%, against 89% for shorter launches. It reads 47.3 KiB from HBM per streamed tile, against 9.4 KiB.

Each launch covers a bounded number of tiles, so the thread blocks start together at every launch boundary. We measured the two schedules paired within one run at 256K. At $d = 128$, a single launch is $1.11\times$ slower than shorter launches at the median and up to $1.36\times$ slower, and its time varies between runs. At $d = 64$, it is $1.36\times$ slower.

At $d = 64$, the code that splits the launches also avoids a trailing launch of at most three tiles. At 2K, such a launch had turned one planned launch into two. Avoiding it cut that measurement from 0.471 to 0.401 ms.

Head dimension 64. At $d = 64$, both passes run more warpgroups that form tiles. Matmul work per tile halves while most elementwise work does not, so the balance shifts toward forming tiles. The $\widetilde{d}(dO)$ accumulator shrinks from 64 to 32 registers per thread. This lets the row pass run one producer and three consumer warpgroups, each reserving 160 registers, with three-stage key and value streams. The column pass runs two construction warpgroups and one accumulation warpgroup.

The column pass exchanges its three tiles through asynchronous `mbarrier` pipelines. Named barriers, the numbered hardware barriers of a thread block, do not suffice here, because two construction warpgroups exceed the budget of 16 named barriers per thread block.

The $d = 64$ row pass reads both Q and U_Q from shared memory. Keeping both in registers does not fit the 160-register allocation, and the compiler miscompiled that variant: it placed $P \odot F$ over the U_Q registers and dropped the $U_Q K^\mathsf{T}$ term. Keeping only Q in registers is correct but makes the row pass 2.1% slower at $N = 4{,}096$ and 5.9% slower at $N = 16{,}384$. The complete BoB, timed in the same runs, is 1.8% and 1.0% slower. That kernel is not limited by shared memory: at $N = 4$K it reaches 81.75% Tensor Core activity at 66.9% L1 utilization.

End-of-iteration wait. Each iteration of the Hopper kernels waits for all of its WGMMA work before the next iteration starts. In the Hopper column kernel, the next tile's score WGMMA, which targeted its own accumulator, was once left in flight as the next iteration started. `ptxas` then made every WGMMA in the kernel synchronous and replaced the three intended waits with waits for all

outstanding work. Issuing the matmul with no consumer wait, with an unguarded wait, or with a guarded wait all gave the same result. The pipelined variant is correct and does not spill, but it is 21.7% slower at $N = 4$K and $d = 64$ in the column pass it was meant to accelerate.

D.3 BLACKWELL

This subsection gives the details behind the Blackwell schedule of Section 4.4. It describes the warp roles and TMEM use of the row and column passes, a microbenchmark of how fast shared memory supplies matmul operands, and the reduction that sums the partial $\widetilde{d}(Q)$.

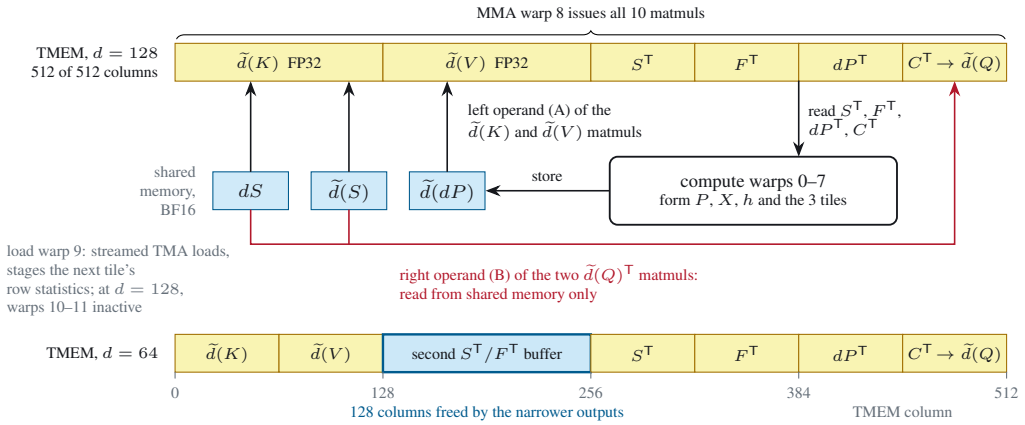

Figure 9: **At $d = 128$, the Blackwell column pass fills all 512 TMEM columns, so the partial $\widetilde{d}(Q)$ reuses the columns of C^T.** The output accumulators and the current S^T, F^T, dP^T, and C^T tiles occupy the 512 columns. The partial $\widetilde{d}(Q)$ takes the C^T columns after their last read. The three tiles are written to shared memory, because the $\widetilde{d}(Q)^\mathsf{T}$ matmuls read dS and $\widetilde{d}(S)$ as right operands. At $d = 64$, the freed columns hold a second $S^\mathsf{T}/F^\mathsf{T}$ buffer.

Row pass. The row pass keeps its tiles P and $P \odot F$ in TMEM, because `tcgen05.mma` reads them only as left operands. Its three score matmuls take two instructions. One wide instruction computes $[S \mid F_2] = \tau Q[K^\mathsf{T} \mid U_K^\mathsf{T}]$. A second instruction reads U_Q from TMEM, where it is copied once per query block, and adds $F_1 = \tau U_Q K^\mathsf{T}$ in place. The thread block has ten warps, named as in FlashAttention-4: warps 0–7 are compute warps that form tiles, warp 8 is the MMA warp that issues the matmuls, and warp 9 is the load warp that issues TMA loads. TMEM holds the output accumulators, the score state, the packed P and $P \odot F$ tiles, and the resident U_Q. The score pipeline runs one key block ahead of the output pipeline. Under a causal mask, query blocks are visited longest first.

Column pass at $d = 128$. The thread block has twelve warps. Warps 0–7 are the compute warps, warp 8 is the MMA warp and issues all ten matmuls, warp 9 is the load warp and issues the streamed TMA loads and stages the next block's row statistics, and warps 10–11 are idle. $\widetilde{d}(K)$ and $\widetilde{d}(V)$ stay in FP32 TMEM across the query loop. The partial $\widetilde{d}(Q)$ reuses the TMEM columns of $C^\mathsf{T} = U_V\, dO^\mathsf{T}$ after their last read (Figure 9). The three tiles are written to BF16 shared-memory buffers, which the five output matmuls read.

Column pass at $d = 64$. The narrower outputs free 128 TMEM columns, and the $d = 64$ kernel uses them to double-buffer S^T and F^T. Each slot is released once its tiles are formed. FlashAttention-4 spends such space on dedicated TMEM slots for P and dS (FlashAttention contributors, 2026). The $d = 64$ kernel also uses four input stages, issues the $\widetilde{d}(Q)$ matmuls first, and splits the matmuls across three MMA warps, 8, 10, and 11.

Operand supply. A matmul of shape 128×64 with both operands in shared memory issues below its peak rate. A microbenchmark on B200 times long chains of `tcgen05.mma` with $M = 128$.

With both operands in shared memory (the SS form in FlashAttention-4's terms), the issue rate is 66% of peak at $N = 64$ and 99–99.7% at $N = 128$ and 256. With the left operand in TMEM (the TS form), $N = 64$ runs at 99%. These measurements fit a shared-memory operand path that sustains the full rate only when $1/M + 1/N \leq 1/64$. The microbenchmark does not depend on our kernels. The profile of the final kernel (Appendix G.1.4) cannot see these operand reads, so we do not attribute a share of the column pass's time to this limit.

Summing $\widetilde{d}(Q)$. Each column-pass thread block adds its partial $\widetilde{d}(Q)$ into one FP32 global buffer with a TMA reduce-add, in the order the thread blocks finish (Figure 10a). The buffer has shape $(G + 1, d, N)$, where G is the number of batch and query-head groups, and it is zeroed on the same stream before the column pass. Each thread block holds a 128-key block and streams 64-row query blocks. Warps 4–7 copy the scaled FP32 $\widetilde{d}(Q)^{\mathsf{T}}$ accumulator from TMEM to shared memory. At $d = 128$ the copy goes into the tile buffers, which are no longer read. At $d = 64$ it goes into a 16 KiB staging buffer. One thread then issues the TMA reduce-add. A final pass transposes the sum and rounds it to BF16 once, in place, using the spare group to avoid overwriting unread values. Because $\widetilde{d}(Q)$ is indexed by query head, grouped-query attention does not change this reduction. It changes the $\widetilde{d}(K)$ and $\widetilde{d}(V)$ sums instead. Each column-pass thread block sums, in FP32, the contributions of the query heads it packs. A second launch then sums the remaining heads that share a key-value head, in FP32 and ascending head order, and rounds once to BF16.

Memory of the $\widetilde{d}(Q)$ sum. Holding every group at once would take $4(G + 1)dN$ bytes, which is 6.125 GiB at $G = 48$, $d = 128$, and $N = 256$K. From $N = 64$K, the released code therefore processes one group per launch and frees each group's sum before the next. The B200 sweep and Table 7 use this code.

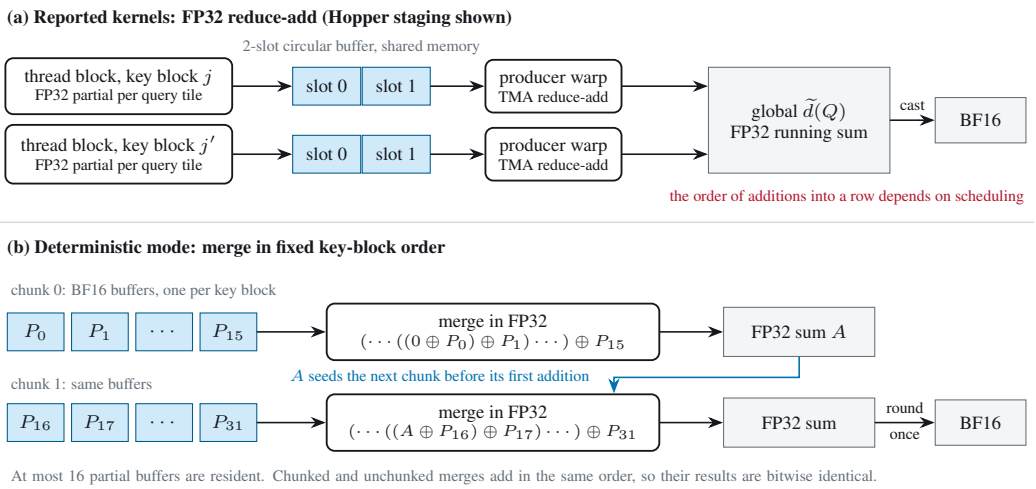

Figure 10: **The reported kernels sum the partial $\widetilde{d}(Q)$ in completion order; the Hopper deterministic mode sums it in a fixed order.** (a) The reported kernels on both GPUs add each partial into an FP32 global $\widetilde{d}(Q)$ in the order thread blocks finish, so the result is not bitwise repeatable. (b) The deterministic mode (Hopper; Appendix F) sums one BF16 per-key-block partial $\widetilde{d}(Q)$ buffer per key block in ascending order, in chunks of at most sixteen. Each chunk starts from the previous FP32 sum, so chunked and unchunked merges agree bitwise.

D.4 Attention JVP kernels

This subsection gives the kernel details behind the attention JVP of Section 4.5 and the timings of Section 5.3. It covers the two JVP interfaces, the fused kernel's algorithm, and its Hopper and Blackwell implementations.

Table 7: **At 256K tokens, the peak memory of FlashBoB-2 is within 0.3% of FlashBoB's on B200 and $1.15\times$ FlashBoB's on H100.** Peak allocated memory of BoB at $N = 256$K, $d = 128$, batch 4, 12 heads, causal, as reported by PyTorch's allocator.

GPU	$\widetilde{d}(Q)$ reduction	FlashBoB-2 (GiB)	FlashBoB (GiB)	Ratio
H100	reduce-add	45.22	39.27	$1.15\times$
B200	reduce-add	39.35	39.24	$1.00\times$

Two interfaces. The choice of kernel depends on whether the JVP will itself be differentiated. If it will, an ordinary attention forward pass saves O and L, and the BoB row pass (Algorithm 2) then returns $\dot{O} = \widetilde{d}(dO)$ and $\dot{L} = \alpha$. This costs the two forward matmuls plus the five row-pass matmuls in two launches, and it keeps O and L for the backward pass. If no backward pass through the JVP is needed, the fused kernel of Algorithm 4 computes O, $\dot{O}$, and L in one launch with six matmuls.

Fused kernel. The fused kernel maintains an online softmax with running maximum m, normalizer l, running row sum r of $\tilde{p} \odot F$, and rescaling factor c. It forms $\alpha = r/l$ internally and does not write it. With $P = \tilde{p}/l$, the epilogue gives $\hat{T}/l = \sum_j P_{ij}(F_{ij}v_j + u_{V,j})$, so $\dot{O} = \hat{T}/l - \alpha O$, which is Equation (13).

Algorithm 4 Fused attention JVP for one query block: O and $\dot{O}$ from the inputs in six matmuls, with no saved state.

Require: resident Q, U_Q; streamed $K_j, U_{K,j}, V_j, U_{V,j}$ for each key block j with unmasked entries

$m \leftarrow -\infty$; $\quad l, r \leftarrow 0$; $\quad \hat{O}, \hat{T} \leftarrow 0$
for each key block j **do**
  issue $S = \tau Q K_j^\mathsf{T}$, $\quad F_1 = \tau U_Q K_j^\mathsf{T}$, $\quad F_2 = \tau Q U_{K,j}^\mathsf{T}$ ▷ 3 matmuls
  $m' \leftarrow \max(m, \operatorname{rowmax} S)$; $\quad c \leftarrow e^{m-m'}$; $\quad \tilde{p} \leftarrow e^{S-m'}$; $\quad m \leftarrow m'$
  $G \leftarrow \tilde{p} \odot (F_1 + F_2)$; $\quad l \leftarrow c\,l + \operatorname{rowsum}(\tilde{p})$; $\quad r \leftarrow c\,r + \operatorname{rowsum}(G)$
  issue $\hat{O} \leftarrow c\,\hat{O} + \tilde{p} V_j$, $\quad \hat{T} \leftarrow c\,\hat{T} + \tilde{p} U_{V,j} + G V_j$ ▷ 3 matmuls
end for
$O \leftarrow \hat{O}/l$; $\quad \dot{O} \leftarrow \big(\hat{T} - (r/l)\,\hat{O}\big)/l$; $\quad L \leftarrow m + \log l$; $\quad$ write $O, \dot{O}, L$

Timed baselines. Section 5.3 times the fused kernel against the kernels of rCM and `jvp_flash_attention`. We do not time Causal-rCM, whose kernel is rCM's with custom masks. We also do not time Terminal Velocity Matching, whose forward kernel adapts an open-source Triton JVP kernel and whose own contribution is the backward pass through the JVP.

Hopper. The fused kernel supports BF16 dense, causal, and causal sliding-window attention; multi-head, grouped-query, and multi-query attention; and head dimensions 64 and 128, plus 32, 80, and 96 by exact padding. Two full-width FP32 output accumulators make the register budget tight. The kernel therefore reads Q and U_Q as shared-memory operands and does not keep them in registers. It uses 168 registers per thread and does not spill. For long dense sequences at $d = 128$, clusters of thread blocks that work on different query blocks of the same head share one multicast stream of K, U_K, V, U_V. The kernel then does not depend on independent thread blocks staying in step to share these loads in L2.

Blackwell. The fused kernel uses the row pass's ten-warp layout: two groups of four compute warps form tiles, one MMA warp issues the matmuls, and one load warp issues TMA loads. TMEM holds the S and F accumulators, both output accumulators, the packed tiles, and the resident U_Q. At $d = 128$ this uses all 512 columns.

Conditional rescaling on Blackwell. Rescaling for the online softmax is costly on Blackwell, because each rescale loads both output accumulators from TMEM, scales them, and stores them back. The kernel therefore keeps its current normalization until the row maximum exceeds it by more than a threshold in base-2 exponent. The threshold is 16. FlashAttention-4's conditional

rescaling (Zadouri et al., 2026) uses the same rule with a threshold of 8. When the threshold is crossed, the kernel applies an exact power-of-two rescale to all accumulated state, which preserves the relative weights. Masked scores are replaced before the row maximum is taken, so a fully masked tile cannot change the normalization.

D.5 Attention variants

This subsection states how the kernels handle grouped-query attention, sliding windows, and unequal query and key lengths, and which combinations our correctness tests cover.

Grouped-query attention. Let H_q and H_{kv} be the numbers of query and key-value heads and $g = H_q/H_{kv}$. Query head a reads key-value head $\lfloor a/g \rfloor$. The equations are unchanged after substituting the K, V, U_K, U_V of that head. The contributions of the query heads that share a key-value head are summed into its $\widetilde{d}(K)$ and $\widetilde{d}(V)$.

Sliding windows and unequal lengths. For a causal sliding window of width W, query i sees keys $\max(0, i - W + 1) \le j \le i$. Both passes skip tiles outside this range and use the same mask for L and the row statistics. Unequal query and key lengths change only offsets and stream bounds.

Tested combinations. We claim only the combinations our correctness tests cover. On Blackwell, these include causal sliding-window grouped-query attention at $d = 64$ and $d = 128$. The Hopper interface accepts grouped-query attention together with sliding windows or unequal lengths, but we have not tested those combinations. BoB supports head dimensions 64 and 128 only; head dimensions 32, 80, and 96 are available only in the fused JVP, through exact padding.

D.6 Resource budget

Table 8 collects the per-thread-block resources behind the schedules of Sections 4.3 and 4.4.

Shared memory. The Blackwell kernels use the full 232,448-byte dynamic shared-memory allocation, which is the 233,472-byte per-SM capacity less the 1,024 bytes reserved per block. For the $d = 64$ Blackwell column pass, Nsight Compute reports the full 232,448 bytes, including its 16 KiB $\widetilde{d}(Q)$ staging tile. The Hopper kernels leave between 7 and 33 KiB unused.

Registers. None of the reported kernels spills. The Hopper kernels divide their registers among warpgroups with `setmaxnreg`. For the final Blackwell $d = 128$ column pass, the compiled binary uses 168 registers per thread, a zero-byte stack frame, and no local-memory loads or stores.

E Experimental Protocol

This appendix gives the full protocol behind the measurements of Section 5: how FLOPs are counted, how kernels are timed, what a timed call contains, how clocks are recorded, which software runs, and which shapes are measured.

FLOP count. Each kernel is credited with its own number of matmuls per tile, and each matmul counts $2d$ FLOPs per unmasked entry of the attention matrix:

$$\text{FLOPs} = 2\, n\, d\, B\, H\, |\mathcal{V}|, \qquad |\mathcal{V}| = \begin{cases} N^2 & \text{dense,} \\ N^2/2 & \text{causal,} \end{cases} \tag{21}$$

where n is the number of matmuls per tile. Windowed and rectangular shapes use the exact number of unmasked entries. We use $n = 15$ for FlashBoB-2 and the fifteen-matmul Triton kernel, and $n = 18$ for FlashBoB. The FlashAttention-3 and FlashAttention-4 backward passes use $n = 5$, their own convention. The fused JVP uses $n = 6$: two forward matmuls, two for F, and two for $(P \odot F)V + PU_V$. rCM counts the same six. Softmax, the row statistics, other elementwise work, and the $\widetilde{d}(Q)$ reduction are not counted. Crediting FlashBoB with fifteen matmuls instead of eighteen would scale its curves by $5/6$.

Table 8: **No reported kernel spills, and every Blackwell kernel uses the full dynamic shared-memory allocation.** Resources per thread block. Hopper register entries are the per-thread register allocations of each warpgroup, set with `setmaxnreg`. Blackwell entries are compiled register counts, except the $d = 64$ column pass, which lists its per-role allocations (mean 168). Shared memory is the dynamic allocation against a 232,448-byte limit.

Kernel	Warps (roles)	Registers	Shared memory	TMEM columns	Stages
Hopper row, $d = 128$	12 (producer, 2 consumer)	24 / 240	194 KiB	–	2
Hopper row, $d = 64$	16 (producer, 3 consumer)	24 / 160	195 KiB	–	3
Hopper column, $d = 128$	12 (producer, construction, accumulation)	24 / 232 / 248	218 KiB	–	2
Hopper column, $d = 64$	16 (producer, 2 construction, accumulation)	24 / 176 / 176 / 136	220 KiB	–	4
Blackwell row, $d = 128$	10 (8 compute, 1 MMA, 1 load)	132	227 KiB	512 (384 live)	2
Blackwell column, $d = 128$	12 (8 compute, 1 MMA, 1 load, 2 idle)	168	227 KiB	512	1
Blackwell column, $d = 64$	12 (8 compute, 3 MMA, 1 load)	232 / 40	227 KiB	512	4

We report these FLOPs per second as a fraction of dense BF16 peak without sparsity: 989.5 TFLOP/s on H100 SXM and 2250 TFLOP/s on B200. These specified peaks correspond to SM clocks near 1830 and 1850 MHz (Shah et al., 2024; Zadouri et al., 2026). The boards boost to 1980 and 1965 MHz. The fraction is not a hardware-counter measurement of Tensor Core activity. Speedups are ratios of measured times and do not depend on the FLOP count.

Timing. Every BoB kernel at every length is timed with one method: `triton.testing.do_bench`, the call that the FlashAttention-3 benchmark script applies to forward passes, with a 3 ms warm-up and a 10 ms repetition window. It clears L2 before each call and does not synchronize between timed calls. We use this one method throughout because an earlier version of our benchmark script switched to synchronized CUDA events above 10 ms per call, and on host-bound calls the two methods disagreed by up to $2.4\times$.

Each run has four blocks, with the kernel order reversed on alternate blocks. This gives 7–31 samples per kernel at a few milliseconds per call, and one sample per block once a call exceeds the repetition window. Each BoB configuration is repeated in three or four independent runs, and each FlashAttention reference in one. Fractions of peak use the per-run mean time, the statistic FlashAttention-3 reports, and take the median over runs. Times and speedups are the median over runs of the per-run medians.

JVP measurements use interleaved CUDA-event timing instead, because we expected that alternating whole blocks would not balance cache state across kernels this different. The protocol has 8 warm-up and 20 timed iterations, with one event pair per launch. The kernels are cycled within each iteration, and two to four input sets are rotated per iteration. Every JVP kernel is called under `torch.no_grad` through its explicit primal-and-tangent interface, which takes the primal inputs Q, K, V and their tangents U_Q, U_K, U_V. `jvp_flash_attention` runs the fastest of four configurations swept at 2K, 16K, and 128K. Only kernels that pass the correctness check are timed. The check compares against FP64 on sampled rows at every timed length, against PyTorch's math backend (the unfused SDPA math path) at 2K, and across kernels, all at relative $\ell_2 \leq 10^{-2}$.

What is timed. A timed BoB call is the whole kernel chain on the saved forward state: the kernel that computes D, the row pass, the column pass with its reduce-add, any grouped-head reductions, and the final cast of $\widetilde{d}(Q)$ to BF16. The GradMem and TTT steps on B200 include the reduce-add and its conversion. The first-order dQ, which only FlashBoB reads, is computed once outside the timed region. Compilation (including CuTe DSL just-in-time compilation) and Triton autotuning are

excluded, and every kernel is built and checked before timing. FlashBoB is autotuned. No CUDA graphs are used.

Clocks. Every measurement records the SM clock, power, temperature, and throttle reasons during its own timing window, because the cloud instances we used do not allow locking clocks. The B200's median SM clock is 1965 MHz in every configuration. From 16K upward, its 1000 W power cap is flagged in 11–20% of samples. On H100, the median SM clock per configuration falls from 1980 MHz at $N = 2$K to 1688 MHz at $N = 256$K ($d = 128$) as the board reaches its 700 W cap. From 16K upward, the power-cap reason is flagged in at least 87% of samples.

Software. All measurements run in single-use containers on Modal, built from `nvidia/cuda:12.8.1-devel-ubuntu22.04` with Python 3.12, PyTorch 2.8.0+cu128, CUDA 12.8, Triton 3.4.0, cuda-python 13.0.3, and `flash-attn` 2.8.3. The last is used only for the rectangular forward and first-order backward. Our kernels are written in CuTe DSL, the Python interface of CUTLASS, from the `nvidia-cutlass-dsl` package: version 4.7.0 on Hopper and 4.2.1 on Blackwell, to which our Blackwell kernels are pinned. The driver is 580.95.05 on both GPUs.

The FlashAttention baselines run in their own images. FlashAttention-3 comes from the community wheel `3.0.0+20260915.cu128torch280cxx11abitrue.0dc2cb` of upstream commit `0dc2cb`, since there is no official Hopper wheel. It runs on the same PyTorch and CUDA as our kernels. FlashAttention-4 comes from `flash-attn-4` 4.0.0b31 on PyTorch 2.13.0+cu130 and Triton 3.7.1. It is kept separate because it requires `nvidia-cutlass-dsl` 4.6.2 or later. The JVP baselines are rCM at `NVlabs/rcm` commit `ed3cb14` and `jvp_flash_attention` at `amorehead/jvp_flash_attention` commit `f7100a84`.

Shapes. The main BoB sweep is causal and BF16, with batch 4, 12 heads with $H_{kv} = H$, $d \in \{64, 128\}$, and N at the powers of two from 2K to 256K. Two further families use the same protocol. The first is the FlashAttention token budget $BN = 16{,}384$ with $H = 2048/d$. Above 16K the batch cannot shrink, so these rows run at $B = 1$ and hold BH fixed. The second is a grouped-query family at $B = 2$, $H_q = 32$, and $H_q/H_{kv} \in \{1, 2, 4, 8, 32\}$. The TTT attention calls use $B = 1$, $H = 32$, and 1,024 queries against 1,024 or 9,216 keys with window 0, 512, or 8,192.

The JVP sweep is dense, with $B = 2$, $H = 24$, and $d = 128$, the attention shape of our sCM diffusion transformer. N runs over the powers of two from 1K to 128K. The 256K configuration did not finish within the 3,600 s per-configuration timeout, so no kernel reports it.

F Numerical Accuracy and Sophia-H Training

This appendix supports the accuracy statements of Section 5. It reports the error of each output against FP64, analyzes the one large error, gives the run-to-run variation, compares RMSE with PyTorch attention in BF16, and compares a Sophia-H training run against PyTorch's math-backend double backward.

Correctness checks. Every timed configuration first passes a correctness check. Small cases ($N \leq 512$) compare all four outputs with a complete FP64 reference. The long-context sweep compares sampled rows of $\widetilde{d}(Q)$ and $\widetilde{d}(dO)$ with FP64, and compares all four outputs between implementations on full tensors. The focused study in Table 9 also compares sampled rows of $\widetilde{d}(K)$ and $\widetilde{d}(V)$ with a blocked FP64 reference on H100 at $d = 128$. It covers one head, selected query and key rows, and all 128 features. It compares no full tensors with FP64 and runs only on H100.

Largest error. The largest error is on $\widetilde{d}(Q)$ for the 4K input of the long-context sweep: 0.03285 before the BF16 conversion and 0.03292 after it. The conversion itself changes $\widetilde{d}(Q)$ by only 0.00164 relative to the FP32 sum. Most of this error therefore arises before conversion, in the BF16 matmuls, intermediate rounding, and accumulation, which this study does not separate. Every other returned output has error at most 0.00808. The 64K random case has a smaller error than the 4K sweep and concentrated inputs. It differs from them in inputs and sampled rows, so these points do not establish a trend with sequence length.

Table 9: **Every returned output stays at least $1.5\times$ below the sampled-check threshold.** Relative ℓ_2 error of sampled output rows against a blocked FP64 reference: H100, causal, BF16, $d = 128$, $N = 4$K except the last row. $\widetilde{d}(Q)_{32}$ is the FP32 sum before conversion and $\widetilde{d}(Q)_{16}$ the returned BF16 result. The sampled check uses a threshold of 0.05 per returned output, looser than the 0.005 of the full-tensor check at $N \leq 512$. The text analyzes the largest error.

Input	$\widetilde{d}(Q)_{32}$	$\widetilde{d}(Q)_{16}$	$\widetilde{d}(K)$	$\widetilde{d}(V)$	$\widetilde{d}(dO)$
Sweep input, 4K	3.29×10^{-2}	3.29×10^{-2}	4.76×10^{-3}	3.16×10^{-3}	5.75×10^{-3}
Near-uniform	2.09×10^{-3}	2.60×10^{-3}	2.45×10^{-3}	2.97×10^{-3}	1.10×10^{-3}
Concentrated	6.51×10^{-3}	6.72×10^{-3}	8.08×10^{-3}	2.90×10^{-3}	6.12×10^{-3}
Cancellation	1.90×10^{-3}	2.57×10^{-3}	2.45×10^{-3}	2.60×10^{-3}	1.16×10^{-3}
Random, $N = 64$K	2.45×10^{-3}	3.11×10^{-3}	3.65×10^{-3}	2.73×10^{-3}	1.52×10^{-3}

Table 10: **FlashBoB-2's error is lower than that of PyTorch's BF16 double backward in every cell.** Root-mean-square error (RMSE) of each output against an FP64 reference computed from the same BF16 inputs, in units of 10^{-3}; the lowest value in each column of a group is in bold. Causal, $N = 2$K, batch 2, 4 heads, one seed; $N = 1$K gives the same ordering between FlashBoB-2 and PyTorch. Outlier inputs follow FlashAttention-3 (Shah et al., 2024): $\mathcal{N}(0,1) + \mathcal{N}(0,100) \cdot \mathrm{Bernoulli}(0.001)$ for every input.

		$\mathcal{N}(0,1)$ inputs				Inputs with outliers			
Setting	Implementation	$\widetilde{d}(Q)$	$\widetilde{d}(K)$	$\widetilde{d}(V)$	$\widetilde{d}(dO)$	$\widetilde{d}(Q)$	$\widetilde{d}(K)$	$\widetilde{d}(V)$	$\widetilde{d}(dO)$
H100, $d = 64$	FlashBoB-2	0.518	0.512	0.334	**0.371**	6.91	2.53	1.01	**1.92**
	FlashBoB	**0.413**	**0.375**	**0.274**	**0.371**	**4.88**	**1.54**	**0.819**	**1.92**
	PyTorch (BF16)	0.943	0.944	0.582	0.723	8.71	3.32	2.50	2.86
H100, $d = 128$	FlashBoB-2	0.492	0.495	0.328	**0.378**	1.80	1.27	0.650	**0.998**
	FlashBoB	**0.415**	**0.383**	**0.270**	**0.378**	**1.61**	**0.868**	**0.549**	**0.998**
	PyTorch (BF16)	1.07	1.08	0.705	0.875	4.36	2.85	1.99	2.41
B200, $d = 64$	FlashBoB-2	**0.401**	0.397	0.276	0.409	5.54	2.23	1.05	1.90
	FlashBoB	0.420	**0.390**	**0.268**	**0.365**	**4.32**	**1.43**	**0.751**	**1.73**
	PyTorch (BF16)	0.926	0.929	0.583	0.723	7.81	3.16	2.28	2.71
B200, $d = 128$	FlashBoB-2	**0.387**	0.385	0.273	0.413	**1.70**	1.11	0.708	1.13
	FlashBoB	0.402	**0.370**	**0.265**	**0.368**	1.79	**0.835**	**0.561**	**1.07**
	PyTorch (BF16)	1.05	1.06	0.686	0.852	4.63	2.86	2.02	2.42

Query row 0. FlashBoB's $\widetilde{d}(Q)$ error on the same rows is 0.0063, and the difference comes from one sampled row, query row 0. For row 0, the kernel returns an error vector parallel to k_0 with norm 0.1051, which matches the 0.1050 predicted by the mechanism below. The relative error is pooled over the five sampled rows, so this row's absolute error is divided by the norms of the other four. Without row 0, the error is 0.0048 against 0.0062 for FlashBoB at 4K, 0.0044 against 0.0034 at 2K, and 0.0030 against 0.0034 at 8K. Over 58 rows spread across the sequence, the median per-row error is 0.28–0.30% for FlashBoB-2 and 0.23% for FlashBoB, with no growth in the number of key tiles.

The error comes from rounding in E°. Under the causal mask, query row 0 attends to a single key, so $X = h = 0$, $E^\circ_0 = C_{00}$, and its exact $\widetilde{d}(Q)$ is zero. Our row pass forms E° through the $(P \odot F)V$ matmul with $P \odot F$ rounded to BF16, while the column pass keeps C in FP32. The kernel therefore returns the rounding error $-\tau\, \delta E^\circ_0\, k_0$. This rounding error is the cost of Equation (4). FlashBoB avoids it by summing E° in FP32 over tiles of dP and C, which are two of the row-pass matmuls per tile that FlashBoB-2 removes.

General rule. Lemma F.1 gives the effect of any error in α or E° on every output.

Lemma F.1 (Sensitivity to α and E°)**.** *If the column pass uses $\alpha + a$ and $E^\circ + e$ with every other quantity unchanged, then, exactly, since the tiles are affine in the statistics,*

$$\Delta\widetilde{d}(dP) = -\operatorname{Diag}(a)P, \qquad \Delta\widetilde{d}(S) = -\operatorname{Diag}(a)\,dS - \operatorname{Diag}(e)P,$$
$$\Delta\widetilde{d}(V) = -P^{\mathsf{T}}\operatorname{Diag}(a)\,dO, \qquad \Delta\widetilde{d}(Q) = -\operatorname{Diag}(a)\,dQ - \tau\,\operatorname{Diag}(e)\,PK,$$
$$\Delta\widetilde{d}(K) = -\tau\big(dS^{\mathsf{T}}\operatorname{Diag}(a)Q + P^{\mathsf{T}}\operatorname{Diag}(e)Q\big), \qquad \operatorname{rowsum}(\widetilde{d}(S) + \Delta\widetilde{d}(S)) = -e,$$

so $\|\Delta\widetilde{d}(Q)_i\| \le |a_i|\,\|dQ_i\| + \tau|e_i|\max_{j\ \textit{unmasked}}\|k_j\|$.

Proof. Substitute into Equations (5) and (6); the row-sum identity uses $\operatorname{rowsum}(dS) = \operatorname{rowsum}(\widetilde{d}(S)) = 0$ (Appendix C.7). □

An error in E° therefore appears exactly as a violated row sum of $\widetilde{d}(S)$, a check a kernel could compute. In our row pass, $E_i^\circ = do_i^{\mathsf{T}}(\hat{P}U_V + \hat{G}V)_i - \alpha_i D_i$, where $\hat{P}$ and $\hat{G}$ are the BF16 operands for P and $G = P \odot F$. The column pass uses FP32 P, F, and C. The value consistent with the column pass is $\sum_j P_{ij}(C_{ij} + F_{ij}X_{ij})$. Relative to it, and up to FP32 accumulation, the error is

$$e_i = \sum_j (\hat{P}_{ij} - P_{ij})\,C_{ij} + \sum_j (\hat{G}_{ij} - G_{ij})\,dP_{ij}.$$

The second sum does not vanish when attention concentrates on one key, while the exact $\widetilde{d}(Q)_i$ does.

Limits. Rows whose attention is nearly one-hot may therefore carry the same kind of relative error as row 0. The concentrated input of Table 9 ($\widetilde{d}(Q)$ error 0.0067) does not show it at the sampled rows, but we have not measured rows constructed to be nearly one-hot.

Inputs and reference coverage. All cases use seeded BF16 normal tensors. Near-uniform attention scales Q, K by 0.1, and concentrated attention scales them by 3. Cancellation inputs use $U_Q = Q$ and $U_K = -K + 2^{-7}Z$ for a normal residual Z, and negate adjacent pairs of rows of dO and U_V. The FP64 reference covers batch and head $(0, 0)$ with query and key rows $\{0, 1, 63, 64, 127, 128, N/2 - 1, N/2, N - 2, N - 1\}$. The 4K sweep input uses rows $\{0, 1, 127, 2048, 4095\}$. Every selected row includes all 128 features. For the 4K sweep input, the BF16 $\widetilde{d}(Q)$ error has absolute ℓ_2 norm 0.1062 against a reference norm of 3.227, with maximum absolute error 0.02319. In the cancellation case, the $\widetilde{d}(V)$ reference norm is only 0.07579 and the absolute error is 1.973×10^{-4}.

Repeatability. Run-to-run variation is four orders of magnitude smaller than the 4K $\widetilde{d}(Q)$ error, so it does not explain that error. Our kernels sum the partial $\widetilde{d}(Q)$ in FP32 in completion order, so bitwise repeatability is not guaranteed. Over ten H100 launches per focused case, sampled $\widetilde{d}(K), \widetilde{d}(V), \widetilde{d}(dO)$ were bitwise unchanged. The largest sampled BF16 $\widetilde{d}(Q)$ variation was 1.20×10^{-6} relative ℓ_2, at 64K. FP32 variation reached 4.49×10^{-8}.

The released code offers a deterministic mode on Hopper ($d = 128$, causal, multi-head and grouped-query). Each key block writes its partial $\widetilde{d}(Q)$ to a per-key-block partial $\widetilde{d}(Q)$ buffer, and the partial buffers are summed in ascending key-block order, at most m resident at a time. The deterministic mode is bitwise identical across repeated launches and across $m \in \{4, 8, 16, 64\}$. Against the default mode, it costs $1.17\times$ the time and $3.39\times$ the memory at $N = 4096$ unchunked, and $1.11\times$ and $6.14\times$ at $N = 8192$. With $m = 16$, it costs $1.88\times$ and $1.61\times$ at 4096, and $3.53\times$ and $1.65\times$ at 8192.

Error against a same-precision baseline. FlashBoB-2 has 1.2–$3.7\times$ lower root-mean-square error (RMSE) than PyTorch's double backward run in BF16, and FlashBoB's RMSE is 0.61–$1.07\times$ that of FlashBoB-2 (Table 10). The checks above compare against FP64 but not against another implementation at the same precision. We therefore repeat the accuracy test of FlashAttention-3 (Shah et al., 2024, Sec. 4.3), which compares a kernel with the standard attention implementation in the same low precision.

Inputs. As in FlashAttention-3, each entry is drawn from $\mathcal{N}(0,1) + \mathcal{N}(0,100) \cdot \mathrm{Bernoulli}(0.001)$: normally distributed with standard deviation 1, plus, for 0.1% of entries, an independent term with standard deviation 10. FlashAttention-3 uses this distribution for Q, K, and V to simulate the outlier features and activations of large language models. We apply it to all seven BoB inputs (Q, K, V, dO, U_Q, U_K, U_V), and we also test $\mathcal{N}(0,1)$ without outliers. All inputs are rounded to BF16, and every implementation receives the same rounded tensors.

Reference and baseline. The reference evaluates the same rounded inputs in FP64 with PyTorch's double backward, so only the arithmetic differs between implementations. The standard implementation is attention written as PyTorch matmuls and a softmax and differentiated twice by autograd with BF16 tensors. It corresponds to FlashAttention-3's standard implementation, which that paper runs in FP16; we use BF16, the precision our kernels support. Table 10 reports the RMSE at $N = 2\text{K}$; the ranges in this paragraph cover both GPUs, $d \in \{64, 128\}$, and $N \in \{1\text{K}, 2\text{K}\}$. For scale, PyTorch's double backward in FP32 has RMSE between 5.4×10^{-8} and 6.5×10^{-7} on the same inputs.

Results. FlashBoB-2's RMSE is lower than PyTorch's on all four outputs, both input distributions, both GPUs, and both head dimensions. This is consistent with where each implementation rounds: PyTorch stores every $N \times N$ intermediate in BF16, while FlashBoB-2 accumulates the recomputed tiles in FP32 and rounds only the matmul operands. FlashBoB's RMSE is lower than FlashBoB-2's on every $\widetilde{d}(K)$ and $\widetilde{d}(V)$ measurement, most on $\widetilde{d}(K)$ with outlier inputs, and on 9 of the 16 $\widetilde{d}(Q)$ measurements. The two $\widetilde{d}(dO)$ errors agree to within 12%. RMSE averages over all rows, so the single-key row analyzed above changes FlashBoB-2's $\widetilde{d}(Q)$ RMSE by at most 1.3%.

F.1 SOPHIA-H TRAINING AND MATCHED UPDATES

The final evaluation loss with FlashBoB-2 differs from that of PyTorch's math-backend double backward (*Math*) by 0.0048 on H100 and 0.0018 on B200 (Table 11). Over the 40 evaluation points, FlashBoB-2 stays within 0.01781 of Math on H100 and 0.01257 on B200. On H100 this lies between the two FlashBoB runs (0.01434 and 0.02790). On B200 it is below both (0.02010 and 0.02275) (Figure 11). All runs use the same seed, so these are single-seed results.

Setup. We train GPT-2 Small (12 layers, 12 heads, width 768; 125,226,240 parameters with 2,048 position embeddings) at $N = 2048$ and $d = 64$ on eight GPUs with Sophia-H. Each run completes 1,908 optimizer steps, 191 Hessian updates, and 1,000,341,504 training tokens with the same data, initialization seed, and data order. The attention forward and first-order backward use cuDNN in every run. Only the backend that computes BoB changes: Math, FlashBoB (run twice), or FlashBoB-2 (192 kernel calls per Hessian update). We evaluate on 32 batches every 50 steps and at the end of training, for 40 evaluation points.

Table 11: **FlashBoB-2's largest deviation from Math lies between or below those of two FlashBoB reruns.** Sophia-H final evaluation loss, and the largest absolute difference from the Math run on the same GPU over all 40 evaluation points.

GPU	Backend	Final loss	Largest difference from Math
H100	Math	3.642257	–
H100	FlashBoB, run 1	3.641753	0.02790
H100	FlashBoB-2	3.637502	0.01781
H100	FlashBoB, run 2	3.637761	0.01434
B200	Math	3.637365	–
B200	FlashBoB, run 1	3.639098	0.02275
B200	FlashBoB-2	3.639200	0.01257
B200	FlashBoB, run 2	3.634293	0.02010

Limitations. We did not test statistical equivalence. The learning-rate warm-up is 2,000 steps, longer than the run, so these runs do not cover a complete warm-up and decay schedule.

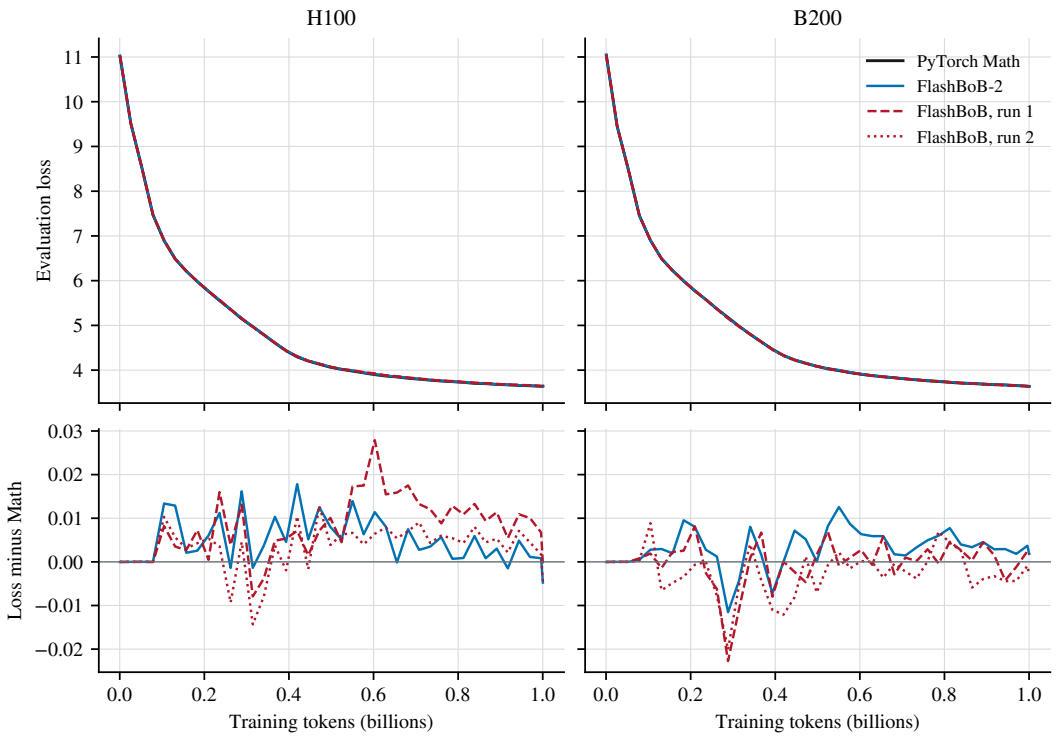

Figure 11: **FlashBoB-2 stays as close to math-backend double backward as reruns of FlashBoB.** Sophia-H evaluation loss over 1B training tokens (top) and its difference from the Math run at each evaluation point (bottom), without smoothing. The two FlashBoB runs use the same seed and show how much a rerun varies.

One matched update. A single matched update compares the backends before their trajectories diverge. FlashBoB-2 (one call per layer) and Math differ by relative ℓ_2 error 0.014469 in the Hessian–vector product, 0.007816 in the FP32 update, and 0.000323 in the BF16 parameters after the update. Both runs start from the same parameters, so the parameter difference comes only from the update, while the norm it is divided by includes the unchanged starting values; the parameter error is therefore smaller than the update error. The cosine similarities are 0.9998955, 0.9999694, and 0.9999999. This separate H100 experiment uses one seeded initial checkpoint, one 2,048-token batch, a fixed Rademacher probe, and an empty Sophia state. A single step does not establish long-run agreement.

G Results Behind Section 5

This appendix gives the measurements behind the ranges of Section 5, under the protocol of Appendix E. Its subsections follow the order of Section 5: BoB speed and efficiency, the sources of the speedup, the attention JVP, and training steps.

G.1 BoB speed and utilization

These measurements support Section 5.1 and Figure 3. They cover the full sweep, PyTorch's double backward, two shape families that occur in models, and hardware counters.

G.1.1 The BoB sweep

Table 12 reports the sweep of Section 5.1. Figure 3 shows all eight lengths.

Table 12: **The speedup over FlashBoB holds from 4K to 256K tokens on both GPUs.** BoB on causal attention given the saved forward state: batch 4, 12 heads, BF16. *Speedup* is FlashBoB's time divided by FlashBoB-2's. Times and speedups are medians over three or four runs of per-run medians. The ranges in the text also include 2K, 8K, 32K, and 128K (Figure 3).

GPU	d	N	FlashBoB (ms)	FlashBoB-2 (ms)	Speedup
H100	64	4K	2.58	1.37	1.87
		16K	35.2	23.0	1.53
		64K	568	369	1.54
		256K	9,443	5,927	1.59
	128	4K	7.34	2.47	2.98
		16K	112	40.9	2.74
		64K	1,747	623	2.81
		256K	28,455	10,367	2.74
B200	64	4K	3.45	1.19	2.90
		16K	49.8	16.8	2.96
		64K	770	264	2.92
		256K	12,469	4,123	3.02
	128	4K	6.88	1.65	4.18
		16K	101	25.8	3.92
		64K	1,592	399	3.99
		256K	25,536	6,170	4.14

G.1.2 PyTorch double backward

PyTorch's double backward is 29.7–$52.0\times$ slower than FlashBoB-2 and runs out of memory at 8K tokens on H100 and 16K on B200 (Table 13). The PyTorch implementation is the math-backend double backward: the SDPA math path differentiated twice through autograd. It is the reference of the Sophia-H runs. PyTorch replays the saved first-order backward graph, which holds $N \times N$ intermediates. With torch 2.8's default, it computes attention in FP32.

All three implementations use the same timed region. The forward pass, the first-order backward, and the first-order quantities are formed outside timing, and each timed call returns all four BoB outputs. Every implementation passes the full FP64 check at $N = 256$ and the sampled check at every timed length. The measurements come from one run per GPU with the timing protocol of Appendix E and unlocked clocks.

Table 13: **PyTorch's math-backend double backward is more than an order of magnitude slower than FlashBoB-2 and runs out of memory first.** BoB given the saved forward state: causal, BF16 inputs, batch 4, 12 heads (ms; one run per GPU). Memory is the peak allocation of one timed call, including inputs. OOM: out of memory.

GPU	d	N	PyTorch	FlashBoB	FlashBoB-2	FlashBoB-2 vs PyTorch	PyTorch memory (GiB)
H100	64	2K	18.40	0.70	0.39	$46.8\times$	9.35
		4K	70.90	2.45	1.36	$52.0\times$	36.63
		8K	OOM	9.13	5.51	–	OOM
	128	2K	23.17	2.04	0.73	$31.6\times$	9.63
		4K	89.92	7.25	2.49	$36.1\times$	37.20
		8K	OOM	26.87	10.70	–	OOM
B200	64	2K	11.39	0.98	0.38	$30.3\times$	9.30
		4K	43.90	3.45	1.19	$37.0\times$	36.58
		8K	170.38	12.93	4.36	$39.1\times$	145.15
		16K	OOM	49.79	17.14	–	OOM
	128	2K	15.92	1.91	0.54	$29.7\times$	9.59
		4K	59.39	7.12	1.71	$34.7\times$	37.16
		8K	230.90	26.16	6.59	$35.0\times$	146.29
		16K	OOM	101.40	26.80	–	OOM

FlashBoB and FlashBoB-2 run every length in the table in under 2.5 GiB, including inputs. PyTorch fails at different stages on the two GPUs. On H100 at 8K, the saved PyTorch graph fits, but the second backward itself runs out of memory. On B200 at 16K, the forward and first-order backward do not fit.

G.1.3 Shapes that occur in models

FlashBoB-2 is also faster than FlashBoB for a fixed token budget and for grouped-query attention (Table 14). The sweep fixes batch and head count while N grows. The first family instead fixes the token budget, $B = \max(1, 16{,}384/N)$. The second family varies how many query heads share each key-value head.

Table 14: **FlashBoB-2 is faster than FlashBoB on both shape families.** Speedup over FlashBoB. H_q:H_{kv} is query heads to key-value heads, so 32:32 is no sharing. Token-budget rows time BoB alone, run at $B = 1$ above 16K, and use a development three-warp B200 schedule. Grouped-query rows time the forward, first-order backward, and BoB together on the final kernels, with batch 2 and 32 query heads.

Family	GPU	d	N	Speedup
Token budget,	B200	64	4K–256K	2.8–3.0×
$B = \max(1, 16{,}384/N)$, $H = 2048/d$	B200	128	4K–256K	3.7–4.2×
Grouped-query, 32:32	H100	64	2K–256K	1.3–1.5×
Grouped-query, 32:8	H100	64	2K–128K	1.5–1.7×
Grouped-query, 32:32	H100	128	2K–16K	2.1×
Grouped-query, 32:8	H100	128	2K–32K	2.1×
Grouped-query, 32:32	B200	64	2K–256K	2.1–2.3×
Grouped-query, 32:8	B200	64	2K–64K	1.3–2.9×
Grouped-query, 32:32	B200	128	2K–16K	2.7–3.1×
Grouped-query, 32:8	B200	128	2K–16K	2.8–3.2×

G.1.4 Hardware profile

On both GPUs, the column pass is not bound by HBM (Table 15). We profiled the final kernels with Nsight Compute: one launch of each, causal, $N = 16$K, batch 4, 12 heads. The cloud instances we used do not allow locking clocks, so we read only ratios and fractions of peak. On H100 at $d = 128$, the top stall of the column pass is the wait on the per-tile barriers (Section 4.3).

On B200, the tensor pipe of the column pass is idle about half the time. The tensor pipe is Nsight Compute's name for the Tensor Cores, the units that execute matmuls. The most frequent stall is long-scoreboard: a warp waits for the result of a memory or synchronization operation. At $d = 128$, a second profile of the column pass with per-instruction source counters locates these stalls. Of the long-scoreboard samples, 83% occur at four mbarrier waits:

- the compute warps waiting for the S and F matmuls (38%);
- the same warps waiting for the $\widetilde{d}(Q)^{\mathsf{T}}$ matmuls (28%);
- the wait before forming $\widetilde{d}(S)$ (9%);
- the wait for the previous tile's reduce-add to release its buffers (8%).

The shared-memory loads account for 1.6% of all samples.

No shared-memory instruction accounts for the reported bank conflicts. Shared memory serves each request in one or more wavefronts, and a bank conflict adds wavefronts. The whole-kernel counter reports 23–36% of shared-memory load and store wavefronts as bank conflicts (Table 15). However, every one of the kernel's 179 shared-memory instructions runs at its ideal wavefront count. At $d = 128$, the reported conflicts equal, to 0.03%, the load wavefronts that no instruction accounts for. They therefore do not come from an access pattern that padding or swizzling would remove. We have not identified their source. These counters do not see `tcgen05.mma` operand reads from shared memory, so they do not measure the operand-supply limit of Appendix D.3.

Table 15: **No column pass is bound by HBM bandwidth, and the B200 kernels keep the tensor pipe less busy than the H100 kernels.** Nsight Compute counters of the final kernels at $N = 16$K. Tensor: tensor pipe active, as a fraction of peak sustained active cycles. Conflicts: the whole-kernel bank-conflict counter as a fraction of shared-memory load wavefronts; per-instruction counters attribute none of it to an access pattern (text). DRAM (HBM): Nsight Compute's DRAM throughput. Top stall reason: the stall reason with the most warp cycles per issued instruction, which is given in parentheses.

GPU	Kernel	Tensor	L1TEX	DRAM (HBM)	Conflicts	Top stall reason
H100	Row, $d = 64$	88.5%	71.6%	76.0%	0.0%	wait (2.06)
H100	Column, $d = 64$	63.3%	80.7%	3.0%	0.2%	long scoreboard (2.29)
H100	Row, $d = 128$	94.4%	47.9%	7.9%	0.1%	wait (2.00)
H100	Column, $d = 128$	75.0%	81.8%	5.8%	0.1%	barrier (2.58)
B200	Row, $d = 64$	40.4%	28.4%	24.7%	39.4%	selected (1.00)
B200	Column, $d = 64$	40.0%	50.0%	1.5%	35.7%	long scoreboard (2.46)
B200	Row, $d = 128$	60.8%	42.6%	53.2%	44.0%	long scoreboard (1.59)
B200	Column, $d = 128$	51.9%	64.9%	2.4%	22.6%	long scoreboard (3.72)

G.2 Sources of the speedup

These measurements support Section 5.2. Appendix G.2.1 gives the evidence for Table 1a, the two-step speedup over FlashBoB. Appendices G.2.2 and G.2.3 give the evidence for Table 1b, where each row compares one of our choices with an alternative.

G.2.1 Fifteen matmuls in Triton

To separate the computation from the schedule, we wrote the fifteen-matmul computation as a Triton kernel (T15). We tuned it separately for each head dimension. At $d = 128$, its column pass processes 128 query rows per streamed block with eight warps and sums $\widetilde{d}(Q)$ with atomic adds. At $d = 64$, it sums $\widetilde{d}(Q)$ with a TMA reduce-add.

On H100, our CuTe DSL kernels give most of the speedup (Table 16). The table compares T15 with FlashBoB and FlashBoB-2 in one series of five independent H100 runs (causal, BF16, batch 4, 12 heads). Every kernel passed the FP64 check on all four outputs in every run. At $d = 128$, the FlashBoB-2 time includes a separate launch that computes D, which the Triton kernels fuse into their row pass. Excluding that launch, T15 takes 1.99, 2.73, and 2.39× as long as FlashBoB-2 at 2K, 8K, and 32K. These ratios come from one paired series of runs. They can differ from ratios derived from Figure 3, which comes from the separate sweep; each comparison is internally matched.

Table 16: **On H100, the step from T15 to our CuTe DSL kernels is larger than the step from FlashBoB to T15.** Time ratios for FlashBoB (18 matmuls, Triton), the fifteen-matmul Triton kernel (T15), and FlashBoB-2.

d	N	FlashBoB / T15	T15 / FlashBoB-2	FlashBoB / FlashBoB-2
64	2K	1.004	1.136	1.144
64	8K	1.035	1.647	1.704
64	32K	1.069	1.465	1.564
128	2K	1.112	1.847	2.053
128	8K	1.123	2.692	3.007
128	32K	1.169	2.384	2.786

B200. The sweep of Section 5.1 includes T15 as a third kernel on B200 (Table 17). It is timed in the same four runs with the same timed region (causal, BF16, batch 4, 12 heads, $d = 128$). Every kernel passed the FP64 check first.

Gap by pass. The column pass holds 80–83% of the gap between T15 and FlashBoB-2 on H100 at $d = 128$ (Table 18). A separate H100 experiment times each pass of T15 and of FlashBoB-

Table 17: **On B200, nearly all of the speedup comes from our CuTe DSL kernels.** Time ratios at $d = 128$: medians over four runs of per-run ratios.

N	2K	4K	8K	16K	32K	64K	128K	256K
FlashBoB / T15	1.047	1.036	1.043	1.044	1.040	1.045	1.062	1.075
T15 / FlashBoB-2	3.401	4.025	4.019	3.758	3.536	3.826	3.900	3.974

2 alone and profiles both with Nsight Compute at 16K ($d = 128$, causal, BF16, batch 4, 12 heads; two launches of three containers each; unlocked clocks). Every kernel first passes the FP64 check. The complete call reproduces the ratios of Table 16: $2.50\times$ at 4K and 2.67–$2.71\times$ at 8K and 16K. The separate D launch adds 0.1–0.25 ms to FlashBoB-2.

The $\widetilde{d}(Q)$ atomics account for only part of the column-pass gap. Removing the $\widetilde{d}(Q)$ matmuls and their atomic adds from the T15 column pass saves 36% of its time. The reduced pass is still 1.87–$1.95\times$ slower than our complete CuTe DSL column pass. A variant with plain stores in place of atomics spills registers and is slower, so it does not isolate the atomics.

The T15 passes execute more instructions and keep the Tensor Cores less busy (Table 19). DRAM (HBM) throughput stays below 8% of peak in all four kernels. The T15 passes execute 2.3–$2.9\times$ as many warp instructions. They keep the Tensor Cores busy in 27–42% of cycles, against 75–94% for FlashBoB-2. The T15 column pass loads operands through registers instead of TMA. It moves $2.3\times$ the shared-memory wavefronts, with 814M bank-conflicted store wavefronts against 4M.

Table 18: **Most of the T15/FlashBoB-2 gap is in the column pass.** Each pass timed alone on H100, $d = 128$ (ms; ranges over two launches). The last column is the column pass's share of the T15/FlashBoB-2 gap. These runs are separate from the sweep, so we compare only ratios within this experiment.

N	T15 row	FlashBoB-2 row	T15 column	FlashBoB-2 column	Column share
4K	1.55	0.74	4.80	1.61	80%
8K	5.52–5.56	2.66–2.67	18.95–19.27	6.22–6.24	82%
16K	20.84–21.08	10.06–10.10	72.68–77.90	24.74–24.87	82–83%

Table 19: **The T15 kernels issue more warp instructions and keep the tensor pipe less busy than FlashBoB-2.** Nsight Compute on H100 at $N = 16$K, $d = 128$, one warm launch of each kernel, unlocked clocks.

Metric	T15 row	FlashBoB-2 row	T15 column	FlashBoB-2 column
Duration (ms)	20.57	10.02	72.35	24.58
Warp instructions (10^9)	6.46	2.22	15.59	6.86
Issue active	33.3%	25.9%	22.3%	31.8%
Tensor pipe active	41.7%	94.3%	26.7%	75.0%
Registers per thread	255	168	255	168
Local-memory sectors, load / store (10^6)	7.7 / 7.7	0	0	0
DRAM (HBM) throughput	3.2%	7.9%	1.1%	5.5%
Shared-memory wavefronts (10^9)	2.34	1.03	10.21	4.37
Bank-conflicted store wavefronts (10^6)	0.9	0.1	814	4.0

G.2.2 THE $7 + 9$ SPLIT

The CuTe DSL $7 + 9$ split takes 1.240–$1.301\times$ as long as $5 + 10$ (Table 20). The measurement is on H100 at fixed $BN = 16{,}384$, $H = 12$, and $d = 128$. Both splits use programmatic dependent launch, which lets a kernel start before the previous kernel in its stream finishes, and the same persistent column-pass launch splitting and work partitioning. The $7 + 9$ column pass drops the $dS\,U_K$ matmul.

The row passes differ in how many query blocks each thread block holds at a time. The $5 + 10$ row pass holds two 64-row query blocks per thread block, and the $7 + 9$ row pass one. The $7 + 9$ row pass already splits each query block across two warpgroups with register allocations of 232 and 248 per thread, beside the producer's 24. Holding two blocks would need $24 + 2(232 + 248) = 984$ registers per thread summed over warpgroups, against the 504 a thread block can allocate. The $7 + 9$ split would therefore need a different design to match. All four kernels use 168 registers per thread, zero-byte stack frames, and no local-memory traffic.

Table 20: **The $7 + 9$ split is slower than $5 + 10$ at every length.** CuTe DSL $5 + 10$ and $7 + 9$ splits on H100, causal BF16, $d = 128$, $BN = 16{,}384$. *Ratio* is $7 + 9$ time divided by $5 + 10$ time; medians over five runs. MAD is the median absolute deviation.

N	$5 + 10$ (μs)	$7 + 9$ (μs)	Ratio	MAD
2K	1368.3	1701.4	1.240	0.0030
4K	2534.7	3148.2	1.238	0.0017
16K	9405.3	12385.6	1.301	0.0030

The slower row pass outweighs the faster column pass. The $7 + 9$ column pass is 5.1–5.4% faster (903.9, 1676.5, and 6158.4 μs against 955.1, 1766.9, and 6507.0 μs). It also avoids clearing the FP32 $\widetilde{d}(Q)$ buffer, which costs about 35 μs in $5 + 10$. Its row pass, however, takes 840.6, 1510.7, and 5629.5 μs against 431.6, 781.2, and 2575.2 μs, or 1.93–2.19$\times$ as long. Absolute times differ from Table 12, which is a separate series of runs. Component times need not sum to the total, because they are timed separately and the total includes the rest of the BoB launch chain.

The $7 + 9$ kernel completes repeated launches and agrees with $5 + 10$. It completed 1,000 consecutive launches, and its outputs stayed within 3.01×10^{-3} relative ℓ_2 of $5 + 10$.

G.2.3 Scheduling measurements

Rows 3, 4, and 6 of Table 1b share one protocol. The Hopper barrier, producer, and Blackwell MMA-warp comparisons each change one choice in the final kernels. They are timed over the complete BoB in five independent runs, with the two versions alternating on two seeded input sets. Both versions of each comparison pass the FP64 check before timing. Row 5, the fused-roles comparison, uses an earlier kernel and its own protocol, described below.

One barrier per tile (H100). Replacing the three barriers with one signal after the last tile makes BoB 4.32% slower at $N = 4$K (MAD 0.0016) and 4.97% slower at 16K (MAD 0.0048). Every run favors three barriers. The false dependency recurs for every streamed query block, which is consistent with the similar ratios at both lengths. The 16K runs reached the H100's 700 W cap, with no thermal or power-brake slowdown reported.

Producer warpgroup (H100). Moving only the loads of the next key and value tiles from the producer into one consumer warpgroup slows the row pass by 1.72% at 4K and 2.70% at 16K. It slows the complete BoB by 0.77% and 0.88%. Row work, register allocations, and shared-memory layout are unchanged. Both binaries use 168 registers per thread and do not spill; the moved loads add TMA instructions. The 16K measurement uses six timed pairs per run.

Fused roles (H100, $d = 64$, earlier kernel). Splitting tile construction and accumulation into two warpgroups is faster at $d = 64$. The fused alternative uses one warpgroup per key block that both forms the tiles and accumulates the outputs. Issuing all recomputation matmuls before consuming any compiles to 168 registers with a 152-byte stack and 80 local loads and 71 local stores, which fails our no-spill requirement. Serializing the recomputation matmuls so that F reuses the S accumulator removes the spills and passes the correctness checks. We timed this version against the split design with counterbalanced paired timing on two rotated input sets larger than L2 (four warm-up and twenty timed iterations). The paired ratio of split to fused time is 0.807 (MAD 0.002) for the complete BoB and 0.747 (MAD 0.002) for the column pass, 1.24$\times$ and 1.34$\times$ as reciprocals. At $d = 64$ the register budget does not force the split, so this comparison measures only its speed. No profile was taken, so the cause of the difference is not identified.

One MMA warp (B200). Spreading the ten column-pass matmuls over three MMA warps, 8, 10, and 11, makes BoB 2.98% slower at 4K and 4.78% slower at 16K. The two binaries issue the same 24 matmul, 20 TMA-load, and 18 TMEM-load instructions, use 168 registers per thread, and do not spill. The three-warp version has 108 synchronization instructions against 100 and synchronizes 384 threads per tile instead of 320. The measurements do not identify which cycles the extra synchronization costs.

Further measurements. Table 21 lists the remaining measurements, several taken on development versions of the kernels.

Table 21: **Each alternative is slower than our choice.** Further scheduling measurements. Each entry is the change in latency when the alternative replaces our choice.

Choice	Effect of the alternative	Setting
Shorter column launches	One launch: $+11\%$ median, up to $+36\%$ ($d = 128$); $+36\%$ ($d = 64$) at 256K, paired within one run	H100; L2 hit rate, one launch against shorter: $55/89\%$ ($d = 128$), $31.6/79.6\%$ ($d = 64$).
Wait at each iteration's end	Leaving a group in flight: $+21.7\%$	H100, $d = 64$, column pass at 4K, development kernel.

Tile placement on Blackwell. Two alternative placements we tried for the tiles produced wrong results. Writing $\widetilde{d}(S)$ and dS into TMEM and dropping the shared-memory copies was 1.5% faster but had a race condition, because the $\widetilde{d}(Q)$ matmuls read those tiles as right operands. Storing the tiles only in TMEM was 3.3% slower and wrote one group of compute warps' dS values into TMEM columns that still held the other group's unread F^{T} values. These failures are specific to those implementations and do not rule out every other layout.

G.3 Attention JVP

These measurements support Section 5.3. Table 22 lists the fused JVP's time and its speedup over both baselines at every length. Table 23 separates the speedup of the row pass into the effect of the fifteen-matmul computation and that of our CuTe DSL kernels. Table 24 repeats the comparison with a causal mask.

Table 22: **The fused JVP is faster than both baselines at every length on both GPUs.** Fused attention JVP, O and $\dot{O}$ together: batch 2, 24 heads, $d = 128$, no mask, BF16. `jvp_flash` is `jvp_flash_attention` (Morehead, 2025) in its fastest configuration; rCM is the kernel of Zheng et al. (2026b). Every kernel computes both outputs in one call with no saved state; ours is Algorithm 4. Speedups are medians of per-run ratios; the abstract and Section 5.3 quote their range over all rows. Peak counts six matmuls. Differentiating through the JVP is not timed.

	H100				B200			
		Speedup over				Speedup over		
N	Ours (ms)	rCM	`jvp_flash`	Peak	Ours (ms)	rCM	`jvp_flash`	Peak
1K	0.13	$1.84\times$	$4.55\times$	62%	0.10	$3.16\times$	$5.45\times$	35%
2K	0.43	$1.97\times$	$4.98\times$	72%	0.33	$3.43\times$	$6.00\times$	42%
4K	1.67	$1.98\times$	$5.03\times$	75%	1.11	$3.54\times$	$6.60\times$	49%
8K	6.51	$2.02\times$	$5.13\times$	77%	4.02	$3.60\times$	$7.12\times$	55%
16K	25.9	$1.97\times$	$5.44\times$	77%	15.5	$3.60\times$	$7.20\times$	57%
32K	108	$1.87\times$	$5.20\times$	74%	64.3	$3.67\times$	$6.94\times$	55%
64K	457	$1.78\times$	$4.92\times$	70%	280	$3.41\times$	$7.34\times$	50%
128K	1,810	$1.82\times$	$4.95\times$	71%	1,152	$3.31\times$	$7.25\times$	49%

Causal mask. A causal mask keeps the speedup over `jvp_flash_attention` and, from 4K tokens, roughly halves our time (Table 24). Table 22 uses no mask because sCM attends in both directions. Many other models use causal attention, so we repeat the comparison with a causal mask.

Table 23: **Each step of the rewrite speeds up the attention JVP on both GPUs.** Same shape as Table 22 ($d = 128$, batch 2, 24 heads, no mask, BF16), N = 1K–32K. The first three columns per GPU take the saved O and L as inputs. *18 → 15*: FlashBoB's row pass (ten matmuls per tile) over the fifteen-matmul row pass (five), both in Triton and autotuned over the same configurations. *Triton → ours*: the fifteen-matmul Triton row pass over our CuTe DSL row pass. *Both*: FlashBoB's row pass over ours. *Fused*: cuDNN's forward plus FlashBoB's row pass, over our fused kernel, which also computes O (the timed region of Table 22). Each entry is the median over three containers of per-container time ratios. All six kernels (these five and cuDNN's forward plus our row pass) run interleaved in each container, and all pass the FP64 check at every length. On H100 the first ratio exceeds the matmul ratio of two. The ten-matmul kernel also holds five FP32 accumulators against two, and the autotuner gives it smaller tiles. –: B200 at 32K, where the fused kernel's time depends on the kernel that ran before it under the 1,000 W power cap.

	H100				B200			
N	18 → 15	Triton → ours	Both	Fused	18 → 15	Triton → ours	Both	Fused
1K	2.92×	1.91×	5.58×	5.10×	1.88×	3.71×	6.98×	6.01×
2K	3.12×	2.03×	6.35×	5.47×	1.97×	3.77×	7.40×	6.28×
4K	3.17×	2.08×	6.61×	5.63×	2.07×	3.85×	7.97×	7.21×
8K	3.22×	2.10×	6.76×	5.71×	2.18×	3.62×	7.87×	7.64×
16K	3.27×	2.09×	6.86×	5.90×	2.21×	3.42×	7.56×	8.42×
32K	3.01×	1.97×	5.93×	5.15×	2.23×	3.24×	7.19×	–

Table 24: **With a causal mask, the fused JVP is 4.0–7.6× faster than `jvp_flash_attention`.** Same shape and timing protocol as Table 22 ($d = 128$, batch 2, 24 heads, BF16), with a causal mask. Peak counts six matmuls over the unmasked entries only. *Causal/full*: our causal time over our time without a mask, both measured in the same container. Medians over two containers of per-container values. rCM is absent because its mask format cannot express a per-token causal mask efficiently (text).

	H100				B200			
N	Ours (ms)	`jvp_flash`	Peak	Causal/full	Ours (ms)	`jvp_flash`	Peak	Causal/full
1K	0.10	4.02×	40%	0.77	0.06	5.58×	28%	0.65
2K	0.28	4.95×	56%	0.64	0.18	6.64×	38%	0.55
4K	0.93	5.48×	67%	0.56	0.61	7.18×	45%	0.55
8K	3.46	5.70×	72%	0.52	2.34	7.23×	47%	0.58
16K	13.2	5.85×	76%	0.51	8.97	7.37×	49%	0.56
32K	54.1	5.76×	74%	0.49	34.8	7.62×	51%	0.53
64K	248	4.97×	64%	0.54	145	7.23×	48%	0.52
128K	1,041	4.78×	61%	0.58	633	6.75×	44%	0.55

Setup. Each container times both kernels twice, with and without a causal mask, interleaved as for Table 22. The speedups without a mask agree with Table 22 to within 2.6%, so the two tables are comparable. Before timing, both causal kernels match an FP64 causal reference at sampled rows and agree with each other, all within 3.3×10^{-3} relative ℓ_2 error.

Results. The fused JVP is 4.02–5.85× faster than `jvp_flash_attention` on H100 and 5.58–7.62× on B200, close to its speedups without a mask (4.55–5.44× and 5.45–7.34×). A causal mask removes about half of the attention entries. From 4K tokens, our causal time is 0.49–0.58× the time without a mask, close to that ideal half. At 1K it is 0.65–0.77×: the partly masked tiles on the diagonal are a larger share of the work, and utilization falls to 40% on H100 and 28% on B200.

Why rCM is absent. rCM's kernel describes a mask as a list of query ranges, each with its own key ranges, and launches one thread block for each query tile that a range covers. A block-causal mask, in which every query of a tile sees the whole diagonal tile, fits this form, but it lets queries see later keys and so computes a different operation. A causal mask gives every query row a different key range. It therefore needs one range per row, and the kernel would launch one full tile for each row. We do not time rCM with a causal mask.

G.4 TRAINING STEPS

These measurements support Section 5.4 and Table 3. Table 25 lists every step measurement, Table 26 decomposes the GradMem steps, and Table 27 times the sCM forward-mode phase.

Workloads. The sCM phase uses a diffusion transformer with hidden width 3072, 24 heads, depth 8, and batch 2. At 256K on B200, only `jvp_flash_attention` was measured. GradMem makes one or three test-time gradient writes per step. TTT uses a 1.468B-parameter model with 1K-token inner chunks.

Table 25: **GradMem steps speed up on both GPUs; TTT steps change little except with full attention at long context on B200.** Optimizer-step speedup over FlashBoB, median of four paired ratios. MAD measures spread and is not a confidence interval. For TTT, the in-step BoB speedup k is the ratio of the two implementations' median BoB times.

GPU	Context	Speedup	MAD
GradMem, one write			
H100	8K	1.196	0.0091
B200	8K	1.350	0.0145
B200	16K	1.617	0.0040
GradMem, three writes			
B200	8K	1.379	0.0035

GPU	Context	Speedup	MAD
TTT, full attention			
H200	8K	0.991	0.0094
H200	16K	1.022	0.0031
H200	32K	1.011	0.0067
H200	64K	1.009	0.0233
B200	8K	0.989	0.0253
B200	16K	1.018	0.0640
B200	32K	1.020	0.0390
B200	64K	1.095	0.0348
B200	128K	1.197	0.0137

GPU	Context	Speedup	MAD
TTT, released 8K window			
H200	32K	1.000	0.0237
H200	128K	0.991	0.0017
B200	32K	1.006	0.0042
B200	128K	0.999	0.0419
B200	256K	0.988	0.0369

Table 26: **BoB's share of the step and its speedup predict the GradMem step speedup closely.** s is BoB's share of the baseline step, k its speedup, and the prediction $1/((1-s)+s/k)$; each is computed within a repeat and the median over repeats is reported. *Measured* is the paired step speedup of Table 25.

Workload	GPU	N	Share s	k	Predicted	Measured
GradMem, one write	H100	8K	24.88%	2.988	1.198	1.196
GradMem, one write	B200	8K	36.04%	4.153	1.376	1.350
GradMem, one write	B200	16K	51.01%	4.340	1.646	1.617
GradMem, three writes	B200	8K	38.70%	3.987	1.408	1.379

Table 27: **The sCM forward-mode phase is faster against both baselines at every measured context.** Speedup of the phase (not a complete training step), median of four paired ratios; `jvp_flash` is `jvp_flash_attention`.

Context	Baseline	Speedup	MAD
H100			
8K	rCM	1.224	0.0027
8K	`jvp_flash`	1.888	0.0011
128K	rCM	1.660	0.0072
128K	`jvp_flash`	4.212	0.0314

Context	Baseline	Speedup	MAD
B200			
8K	rCM	1.589	0.0044
8K	`jvp_flash`	2.368	0.0083
128K	rCM	2.974	0.0041
128K	`jvp_flash`	6.293	0.0046
256K	`jvp_flash`	6.811	0.0412

TTT with full attention. The full-attention variant is a stress test and differs from the released recipe. It raises the longest key span from 7,168 tokens at 8K to 130,048 at 128K. Over the same range, the number of BoB calls grows only from 560 to 635. BoB's share of the step therefore rises from 3.8% to 26.6%.

TTT with the released window. The released 8K sliding window caps the key span at 9,216. It keeps BoB near 4–6% of the step on B200 and about 3% on H200. The windowed step speedups lie between 0.988 and 1.006.

Memory limits. TTT at 256K with full attention exceeds B200 memory with both FlashBoB-2 and FlashBoB. The Hopper TTT steps run on an H200, not an H100. On the H200, full attention at 128K and the window at 256K exceed its memory.