FlashBoB: I/O-Efficient Exact Backward-over-Backward for Softmax Attention

Anthony Givans, Michael Crawshaw, Mingrui Liu*
Department of Computer Science
George Mason University, Fairfax, VA 22030
agivans2@gmu.edu, mcrawsha@gmu.edu, mingruil@gmu.edu

Abstract

Transformer models built on the attention mechanism have become a central building block in modern deep learning, yet softmax attention remains a major bottleneck for long-context workloads. While FlashAttention makes the forward and first backward passes I/O-efficient, it does not support *backward-over-backward* (BoB), which enables exact differentiation through the backward pass for applications such as second-order optimization, test-time training, gradient-written memory, and meta-learning. Existing BoB implementations either materialize large intermediate tensors or exhaust GPU memory at long sequence lengths.

We present FLASHBOB, an exact, I/O-efficient algorithm for BoB in softmax attention that keeps computation within on-chip tiles and avoids all $N \times N$ intermediate tensors, where N is the sequence length. The key insight is a hierarchical affine structure in the softmax double backward: two row-wise scalars determine all outputs through affine transformations. This yields a two-pass schedule with bounded on-chip static random-access memory (SRAM) usage and minimal off-chip high-bandwidth memory (HBM) traffic. FLASHBOB achieves $\Theta(N^2d^2/M)$ HBM traffic (d is the head dimension and M is the on-chip SRAM size) and, within the standard FlashAttention-style score-recomputation model, matches the inherited large-cache lower bound for exact forward attention. Empirically, in an isolated attention-layer benchmark on a single A100 80GB GPU, it computes exact attention BoB at $N = 262$K, where prior PyTorch exact baselines exhaust memory by $N = 16$K, and it is up to $9.4\times$ faster than FlashBack. Across complete second-order steps on GPT-2, Pythia, and five native GQA and MoE models, FLASHBOB is 1.6–$3.8\times$ faster than the fastest exact baseline that fits on chip, supports 2–$8\times$ longer contexts, and cuts 1B-token Sophia-H training time and peak memory on FineWeb by $2.2\times$. These results make exact second-order attention practical at long-context sequence lengths where prior implementations cannot run efficiently.

1 Introduction

Transformers are the dominant architecture for sequence modeling [54], and attention is their core operation. The cost of attention grows quadratically with sequence length and, at scale, is dominated by data movement between high-bandwidth memory (HBM) and on-chip SRAM. Two lines of work address this at long context. One modifies the operator: recurrent memories, sparse patterns, low-rank or kernelized attention, and state-space models reduce or replace the full pairwise interaction [8, 3, 59, 26, 55, 7, 23, 44, 43, 17]. The other, FlashAttention [11, 10, 50, 58], preserves exact softmax attention and redesigns the memory schedule to make it I/O-efficient: tiles are recomputed from saved statistics, and the full attention matrix is never materialized in HBM. The key design principle is that for exact attention, the memory schedule is part of the algorithm.

*Corresponding Author. The implementation is at https://github.com/MingruiLiu-ML-Lab/FlashBoB

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

This design principle has so far been limited to the forward and first backward passes of softmax attention. A growing class of methods places gradients inside the model's computation and must differentiate through the attention backward pass itself; we call this primitive *backward-over-backward* (BoB). Such methods arise across modern training paradigms, including second-order optimization via Hessian-vector products for curvature estimation [32, 41, 34, 35, 19, 56], test-time training that differentiates through inner-loop updates [52], gradient-written memory that stores context via per-example gradient descent [27], and meta-learning that differentiates through inner optimization [15, 38, 47]. At the kernel level, all of these rely on the same BoB primitive, yet at long context none of the existing implementations scale. Generic automatic differentiation materializes intermediate tensors of quadratic (sequence-by-sequence) size and exhausts memory at sequence length $N = 8{,}192$ for an isolated GPT-2 Small attention layer (batch 4, 12 heads, $d = 64$, causal, BF16 inputs with FP32 reductions, no GQA/MQA) on a single A100 80GB GPU. GradMem [27] ships custom Hessian-vector product kernels that fuse parts of the second backward and avoid the worst materialization, doubling the feasible context to $N = 8{,}192$ on the same hardware before exhausting memory at $N = 16{,}384$. The closest prior non-materializing kernel, FlashBack [13], fuses the entire BoB into a single GPU kernel parallelized over rows, with several outputs accumulated by concurrent threads writing to the same memory locations through synchronized hardware updates that serialize and stall under contention. Crucially, none of these approaches extend FlashAttention's memory discipline to backward-over-backward.

Extending FlashAttention-style scheduling to backward-over-backward faces two key obstacles. First, the computation requires combining information in both row-wise and column-wise directions, which prevents a single consistent pass over the data. Second, the softmax double backward introduces global dependencies that must be resolved before downstream results can be finalized. Together, these constraints break the simple, single-pass execution pattern that keeps intermediate data in fast on-chip memory: a naive implementation either requires multiple passes or stores large intermediate tensors. Both options sacrifice the memory and I/O efficiency that makes FlashAttention effective.

We resolve these challenges with FLASHBOB, an exact, I/O-efficient algorithm for backward-over-backward in softmax attention. The key idea is to expose a hierarchical affine structure in the softmax double backward, which allows the computation to be reorganized into a small number of passes that avoid materializing $N \times N$ tensors and keep intermediate data in fast on-chip memory. The resulting schedule closely follows the FlashAttention execution model, incurring only minimal overhead beyond the first backward pass while preserving efficient memory access patterns. As a result, FLASHBOB scales to sequence lengths far beyond prior exact implementations [27, 13] and removes the attention-BoB memory bottleneck for methods that require exact second derivatives through softmax attention. Theoretically, it matches a large-cache I/O lower bound under the standard FlashAttention-style score-recomputation model, where score and probability tiles are recomputed from saved row statistics rather than stored as $N \times N$ HBM tensors. Our contributions can be summarized as follows.

- **FlashBoB algorithm.** We introduce an exact two-pass algorithm for softmax-attention BoB that preserves the FlashAttention execution model: no $N \times N$ HBM intermediates, bounded on-chip workspace, and one write per finalized output block. The key identity is a hierarchical affine structure in the softmax double backward (Proposition 4.1 and Figure 1) that collapses the natural four-pass schedule to two passes.

- **I/O analysis and optimality.** We establish a $\Theta(N^2 d^2/M)$ HBM upper bound for FLASHBOB (Theorem 4.3) and a matching large-cache lower bound under the standard FlashAttention-style score-recomputation model (Theorem 4.5). The lower bound is inherited from exact forward attention by embedding it inside a special BoB instance.

- **Kernel and model-level impact.** On an isolated attention layer (A100 80GB), FLASHBOB reaches $N = 262{,}144$, where PyTorch materialization fails at $N = 8{,}192$ and GradMem HVP baselines fail by $N = 16{,}384$. It is up to $10\times$ faster and $28\times$ more memory-efficient than materializing PyTorch, 2.1–$2.4\times$ faster than HVP baselines at $d = 128$ (up to $3.8\times$ at $d = 64$), and up to $9.4\times$ faster than FlashBack. Across ten complete models (GPT-2, Pythia, and five native GQA/MoE architectures), it yields 1.6–$3.8\times$ speedups and fits 2–$8\times$ longer sequences. On 1B-token FineWeb training with Sophia-H, it cuts wall time and peak memory by $2.2\times$ (1.22–$1.70\times$ in a matched hybrid baseline isolating the BoB kernel) while matching the loss trajectory.

2 Related Work

GPU memory hierarchy and kernel fusion. The performance principle behind FlashAttention is part of a broader systems literature: communication between memory levels can dominate arithmetic, recomputation can trade extra work for lower memory pressure, and combining multiple operations into one GPU kernel can avoid expensive reads and writes [20, 1, 5, 22, 25, 46, 6]. GPU programming systems such as CUDA and Triton expose the memory hierarchy well enough to keep small working sets in fast on-chip memory while using specialized hardware for matrix multiplication [39, 53, 33]. FLASHBOB applies this principle to a higher-order derivative: rather than changing the attention operator or introducing an approximation, it changes the execution schedule so that the exact BoB computation avoids unnecessary HBM traffic.

Attention I/O and exact streaming. Long-context attention has been pursued along two lines. Approximate or structured-attention methods change the operator: recurrent memories, sparse patterns, low-rank projections, kernelized attention, and state-space models reduce the explicit quadratic interaction by modifying the computation [8, 3, 59, 26, 55, 7, 23, 44, 43, 17]. Building on online softmax normalization and exact memory-efficient attention [36, 45], the FlashAttention family [11, 10, 50, 58] keeps exact softmax attention and changes the memory schedule: it recomputes attention blocks from saved softmax statistics so the full $N \times N$ attention matrix is never written to HBM. Lower-bound work formalizes these gains in two-level memory and red-blue pebble models [20, 1, 48, 30]. FLASHBOB belongs to this second line: it preserves exact softmax attention and extends FlashAttention's streaming memory schedule to the second derivative.

Implementations of attention backward-over-backward. Existing implementations of exact attention BoB, and more general higher-order derivative tooling [9], fall into three broad categories. Generic reverse-mode automatic differentiation [2] through a PyTorch-style tensor graph [40] provides a natural correctness oracle, but stores quadratic-size intermediate tensors and therefore scales poorly at long context. GradMem [27] provides custom Hessian-vector product kernels (HVP-M, HVP-S) that avoid the worst materialization, but they do not follow the FlashAttention-style streaming schedule. FlashBack [13] is the closest prior non-materializing kernel we are aware of: it fuses the BoB computation into a single GPU kernel, but relies on shared updates to global memory for some outputs and repeatedly recomputes attention blocks. In contrast, FLASHBOB separates the work into two streaming passes that match the natural structure of the BoB outputs, avoid shared global updates, and expose affine fusion that a single-kernel design cannot exploit without additional synchronization or recomputation.

Applications of attention backward-over-backward. Backward-over-backward through attention appears in several long-context training settings. Second-order methods use Hessian-vector products to obtain curvature information without forming the Hessian explicitly, from classical Hessian-free and preconditioned optimization to modern stochastic curvature estimators [41, 34, 35, 19, 56, 32]. Meta-learning differentiates through inner gradient steps [15, 38, 47], while recent test-time training and gradient-written memory methods differentiate through inner-loop adaptation or per-example writing procedures [52, 27]. These settings motivate exact BoB implementations for softmax attention.

3 Background and Preliminaries

Single-Head Attention in Transformer. For a single attention head in a Transformer, the inputs $Q, K, V \in \mathbb{R}^{N \times d}$ are the query, key, and value matrices across N tokens of the sequence [54]. With $\tau = 1/\sqrt{d}$, the forward pass is

$$S = \tau Q K^{\mathsf{T}} + B_{\text{mask}}, \qquad P = \text{softmax}(S), \qquad O = PV, \tag{1}$$

where $B_{\text{mask}} \in \{0, -\infty\}^{N \times N}$ encodes any attention mask (e.g., causal, sliding-window, or block patterns), and the softmax is applied row-wise. For row $i \in [N]$, let $\mathcal{V}_i = \{j \in [N] : (B_{\text{mask}})_{ij} \neq -\infty\}$ denote the set of valid columns; all row sums below are over $\mathcal{V}_i$. Lowercase letters denote row slices: $q_i, k_i, v_i, o_i, do_i \in \mathbb{R}^d$, where do_i denotes the gradient of the loss with respect to o_i and $i = 1, \ldots, N$. Table 6 in Appendix A collects the full notation, including the BoB-specific objects introduced in §4.1. For simplicity, we consider only single-head attention in our derivations, and the analysis extends naturally to multiple heads and to grouped-query or multi-query attention, where H_Q/H_{KV} query heads share one key-value head (§5.2).

Two-level GPU Hierarchy. We work in the standard two-level memory model [20, 1]: HBM is large and slow, SRAM is small and fast, and I/O counts elements moved between them. SRAM

has capacity M elements. We process attention using a tiled (blocked) algorithm: let B_r and B_c denote the row and column tile sizes, respectively, so that blocks of size $B_r \times d$ and $B_c \times d$ from Q, K, V (and associated workspace) are materialized in SRAM during computation. We work in the large-cache regime $M = \Omega(d^2)$ so that these tiles and the necessary local workspaces fit on chip [11, 48]. The goal throughout the paper is to avoid HBM traffic for $N \times N$ intermediates while keeping the arithmetic exact.

Flash Attention. FlashAttention [11, 10] processes attention using a tiled algorithm, partitioning rows into blocks of size B_r and columns into blocks of size B_c. The forward pass stores only the row-wise log-normalizer $L_i = \log \sum_{j \in \mathcal{V}_i} \exp(S_{ij})$ together with the output O, where $\mathcal{V}_i$ denotes the set of valid columns for row i. Later passes recompute probability tiles $P_{ij} = \exp(S_{ij} - L_i)$ inside SRAM, so the full $N \times N$ matrix P is never stored in HBM.

Given $dO = \partial \mathcal{L} / \partial O$, the attention backward computes dP, dV, dS, dQ, dK as

$$dP = dO\, V^{\mathsf{T}}, \qquad dV = P^{\mathsf{T}} dO, \qquad dS_{ij} = P_{ij}(dP_{ij} - D_i), \tag{2}$$

$$dQ = \tau\, dS\, K, \qquad dK = \tau\, dS^{\mathsf{T}} Q, \tag{3}$$

where the row-wise scalar $D_i = \sum_{j \in \mathcal{V}_i} P_{ij}\, dP_{ij} = do_i^{\mathsf{T}} o_i$ is the first-backward row summary.

FlashAttention keeps only the row statistics L (from the forward) and D (from the first backward), together with O; probability tiles are recomputed from L inside SRAM, and P, dP, dS are never stored as full $N \times N$ matrices. We refer to this schedule as the *FlashAttention execution model*: it maintains bounded $O(M)$ on-chip working sets, avoids global atomic adds on outputs, writes no $N \times N$ intermediates to HBM, and recomputes probability tiles from saved row state. FLASHBOB reuses this state and adds only two BoB row-state vectors, (α, E), which are introduced in Section 4.

Motivating applications of backward-over-backward. The forward and first-backward kernels are insufficient for methods that differentiate through attention-containing gradient steps, including Sophia-H Hessian-vector products [32, 41], end-to-end test-time training [52], and gradient-written memory [27]. At the attention boundary, all of these supply upstream matrices $U_Q, U_K, U_V \in \mathbb{R}^{N \times d}$ at the first-backward outputs (dQ, dK, dV). We define this BoB primitive formally in §4.1 and use Sophia-H and GradMem-derived kernels in the experiments.

4 FlashBoB: Algorithm and Analysis

This section develops the FLASHBOB schedule in five steps: we first define the BoB primitive (§4.1); derive closed-form identities (§4.2); present two row-scalar reductions that reduce the natural FlashAttention-style tiled schedule from four sweeps to two passes (§4.2); describe the resulting algorithm (§4.3); and finally analyze its I/O complexity together with a matching score-recomputation lower bound in the large-cache regime (§4.4).

4.1 Backward-over-backward

Let $g(Q, K, V, dO) = (dQ, dK, dV)$ denote the first-backward map of (2)–(3). Given matrices $U_Q, U_K, U_V \in \mathbb{R}^{N \times d}$, each application in §3 requires the vector–Jacobian product of g with (U_Q, U_K, U_V), which is obtained by reverse-mode automatic differentiation applied to the scalar

$$\Phi = \langle dQ, U_Q \rangle + \langle dK, U_K \rangle + \langle dV, U_V \rangle, \tag{4}$$

where $(dQ, dK, dV) = g(Q, K, V, dO)$ and $\langle A, B \rangle = \mathrm{tr}(A^{\mathsf{T}} B)$ denotes the Frobenius inner product. Backward-over-backward (BoB) computes the four resulting gradients

$$\big(\widetilde{d}(Q),\, \widetilde{d}(K),\, \widetilde{d}(V),\, \widetilde{d}(dO)\big) \;:=\; \nabla_{Q,K,V,dO}\, \Phi. \tag{5}$$

For any intermediate variable X in the first-backward graph, $\widetilde{d}(X) := \partial \Phi / \partial X$ is its *BoB gradient*; when g is the gradient of a scalar loss, $\nabla \Phi$ is the HVP applied to the attention block [41].

4.2 Affine Fusion

Reverse-mode automatic differentiation applied to (4) (full derivation in Appendix C) yields the four BoB outputs as

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

$$\widetilde{d}(V) = (P \odot h)^{\mathsf{T}} dO, \qquad \widetilde{d}(dO) = P\, U_V + (P \odot h)\, V, \tag{7}$$

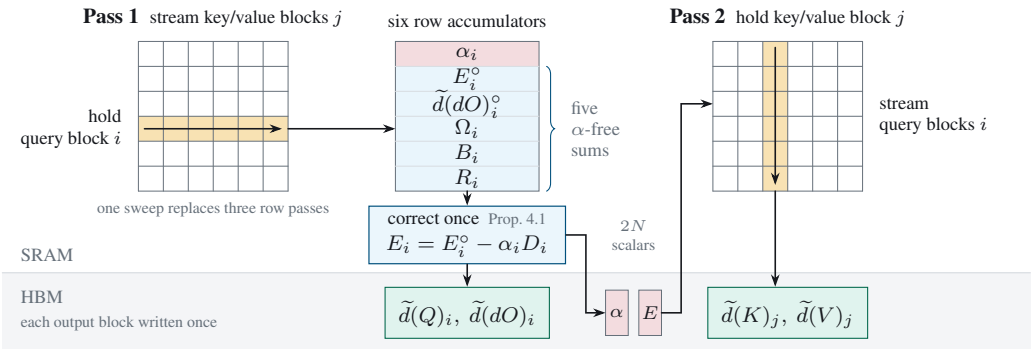


Figure 1: FLASHBOB overview. Pass 1 holds query block i, streams key/value blocks into six row accumulators, and applies the affine corrections of Proposition 4.1 once per row. Only α and E ($2N$ scalars) cross to Pass 2, which holds key/value block j and streams query blocks. No $N \times N$ tensor reaches HBM.

where $\odot$ denotes elementwise multiplication; dP, dS, and D_i are the first-backward intermediates from (2)–(3); and $h_{ij} = F_{ij} - \alpha_i$, $F = \tau(U_Q K^\mathsf{T} + Q U_K^\mathsf{T})$, $C = dO\, U_V^\mathsf{T}$, $\alpha_i = \sum_{j \in \mathcal{V}_i} P_{ij} F_{ij}$, $\widetilde{d}(P)_{ij} = C_{ij} + F_{ij}(dP_{ij} - D_i) - \alpha_i dP_{ij}$, $E_i = \sum_{j \in \mathcal{V}_i} P_{ij} \widetilde{d}(P)_{ij}$, and $\widetilde{d}(S)_{ij} = P_{ij}(\widetilde{d}(P)_{ij} - E_i)$. We write $\alpha = (\alpha_1, \ldots, \alpha_N)$ and $E = (E_1, \ldots, E_N)$. Thus the second derivative introduces two row scalars, α_i and E_i, which are the BoB analogs of the first-backward row scalar D_i.

The four outputs split according to how their sums are formed: $(\widetilde{d}(Q), \widetilde{d}(dO))$ and the row scalars (α, E) sum over columns for each fixed row, while $(\widetilde{d}(K), \widetilde{d}(V))$ sum over rows for each fixed column. Under the FlashAttention execution model, this requires a row-wise and a column-wise pass. In addition, the row-wise computations appear to have the sequential dependence $\alpha_i \to \widetilde{d}(P)_{ij} \to E_i \to \widetilde{d}(S)_{ij}$, which suggests four passes: one for α, one for E, one for $(\widetilde{d}(Q), \widetilde{d}(dO))$, and one column-wise pass for $(\widetilde{d}(K), \widetilde{d}(V))$. Both row scalars, however, enter later quantities only *affinely*: α_i enters $\widetilde{d}(P)_{ij}$ through the term $-\alpha_i dP_{ij}$, and E_i enters $\widetilde{d}(S)_{ij}$ through a broadcast subtraction. These affine forms allow the row-wise computations to be fused: one can accumulate α-free row quantities[2] first, and then apply the affine corrections once the row scalars are known, without revisiting the row. Proposition 4.1 makes this explicit by naming six row accumulators, five of which are α-free, and the corrections that finalize the row outputs within each row block (Figure 1).

Proposition 4.1 (Affine fusion)**.** *There exist six row accumulators consisting of α_i together with five α-free quantities $E_i^\circ, B_i, \widetilde{d}(dO)_i^\circ, R_i, \Omega_i$ (defined in Appendix D), all with tilewise computable summands, such that*

$$E_i = E_i^\circ - \alpha_i D_i, \quad \widetilde{d}(dO)_i = \widetilde{d}(dO)_i^\circ - \alpha_i o_i, \quad \widetilde{d}(Q)_i = \tau\big(\Omega_i - \alpha_i R_i - E_i B_i\big). \tag{8}$$

The proof is by direct substitution; see Appendix D. A single-pass schedule for both row-owned and column-owned outputs would need to pay one of three costs: materializing an $N \times N$ tensor in HBM, accumulating column-owned outputs through global atomic adds, or holding $\Theta(Nd)$ partial column accumulators on chip. FlashBack [13] pays the second cost; FLASHBOB uses two passes to avoid all three. The schedule-level argument is given in Appendix G.1.

4.3 Algorithm

Algorithm 1 realizes the two-pass schedule in the block-streaming style of FlashAttention (Figure 1). HBM-resident inputs are $Q, K, V, dO, U_Q, U_K, U_V, O \in \mathbb{R}^{N \times d}$ and the saved row state $L, D \in \mathbb{R}^N$. Pass 1 keeps a query-row block in SRAM, streams over key-value blocks, accumulates the six row accumulators of Proposition 4.1, applies the corrections (8), and writes the auxiliary row quantities (α_i, E_i) together with $(\widetilde{d}(Q)_i, \widetilde{d}(dO)_i)$ once per row block. Pass 2 keeps a key-value block in SRAM, streams over row blocks together with the saved (α_i, E_i), and writes $(\widetilde{d}(K)_j, \widetilde{d}(V)_j)$ once per

[2]We describe a quantity as α-free if it can be computed without first computing α.

Algorithm 1 FLASHBOB: two-pass exact attention BoB. A fully annotated version with explicit tile shapes appears as Algorithm 2 in Appendix F.

Input: Matrices $Q, K, V, dO, U_Q, U_K, U_V, O \in \mathbb{R}^{N\times d}$; row state $L, D \in \mathbb{R}^N$, all in HBM; SRAM size M
Output: BoB outputs $\widetilde{d}(Q), \widetilde{d}(K), \widetilde{d}(V), \widetilde{d}(dO)$ and row scalars α, E

$b \leftarrow \lfloor M/(c_{\text{BoB}}d)\rfloor$; $B_r^{(1)}, B_c^{(2)} \leftarrow b$; $B_c^{(1)}, B_r^{(2)} \leftarrow \min(b, d)$

PASS 1 (ROW-MAJOR): Accum. row summaries $\rightarrow$ apply scalar updates $\rightarrow$ write $(\alpha, E, \widetilde{d}(Q), \widetilde{d}(dO))$

for $i = 1$ **to** $\lceil N/B_r^{(1)}\rceil$ **do**
  Load active row tiles $(Q_i, dO_i, U_{Q,i}, O_i)$ and saved row scalars (L_i, D_i)
  Zero the six row accumulators ▷ Prop. 4.1
  for $j = 1$ **to** $\lceil N/B_c^{(1)}\rceil$ **do**
    Load $(K_j, V_j, U_{K,j}, U_{V,j})$; recompute P_{ij} ▷ from saved L
    Form tile intermediates $dP_{ij}, F_{ij}, C_{ij}, \widetilde{d}(P_{ij})^\circ$ on chip
    Accumulate $\alpha_i,\ E_i^\circ,\ \widetilde{d}(dO)_i^\circ,\ \Omega_i,\ B_i,\ R_i$
  end for
  Apply scalar updates to form $E_i, \widetilde{d}(dO)_i, \widetilde{d}(Q)_i$ ▷ Eq. (8)
  Write $\alpha_i, E_i, \widetilde{d}(Q)_i, \widetilde{d}(dO)_i$ to HBM
end for

PASS 2 (COLUMN-MAJOR): Load saved (α, E) $\rightarrow$ accumulate and write $(\widetilde{d}(K), \widetilde{d}(V))$

for $j = 1$ **to** $\lceil N/B_c^{(2)}\rceil$ **do**
  Load active column tiles $(K_j, V_j, U_{K,j}, U_{V,j})$; zero $\widetilde{d}(K)_j, \widetilde{d}(V)_j$
  for $i = 1$ **to** $\lceil N/B_r^{(2)}\rceil$ **do**
    Load streamed row tiles and saved row scalars $(L_i, D_i, \alpha_i, E_i)$; recompute P_{ij} and tile intermediates
    $\widetilde{d}(V)_j \mathrel{+}= (P_{ij} \odot h_{ij})^\mathsf{T} dO_i$; $\widetilde{d}(K)_j \mathrel{+}= \tau(\widetilde{d}(S)_{ij}^\mathsf{T} Q_i + dS_{ij}^\mathsf{T} U_{Q,i})$
  end for
  Write $\widetilde{d}(K)_j, \widetilde{d}(V)_j$ to HBM
end for

column block. No $N \times N$ intermediate is ever written to HBM. Block sizes follow the FlashAttention-style SRAM budget. With $b = \lfloor M/(c_{\text{BoB}}d)\rfloor$, where $c_{\text{BoB}} > 0$ accounts for all arrays and scalar buffers that must be stored in SRAM while processing one block, Pass 1 keeps a row block in SRAM and streams smaller column blocks, $B_r^{(1)} = b$ and $B_c^{(1)} = \min(b, d)$; Pass 2 reverses the orientation, using $B_c^{(2)} = b$ and $B_r^{(2)} = \min(b, d)$. Under grouped-query or multi-query attention ($H_Q > H_{KV}$), each query head in Pass 1 reads its group's shared (K, V, U_K, U_V) head, while Pass 2 accumulates $(\widetilde{d}(K)_j, \widetilde{d}(V)_j)$ across both row blocks and the H_Q/H_{KV} query heads in the group before writing once to HBM. In our Triton implementation at $d = 128$, autotuning selects $(B_r, B_c) \in \{(16, 16), (32, 16)\}$ with two stages for Pass 1 and $(B_r, B_c) \in \{(16, 16), (16, 32)\}$ with two or three stages for Pass 2, all with four warps (Appendix N.1); Appendix H gives the SRAM budget derivation.

Theorem 4.2 (Correctness, FLOPs, and memory)**.** *Algorithm 1 computes exactly the BoB outputs defined by Equations* (6)–(7)*, requires $\Theta(N^2 d)$ FLOPs, uses $O(N)$ extra HBM beyond inputs, outputs, the saved row state (L, D), and the forward output O, and fits within the fast-memory budget M for block sizes B_r, B_c satisfying $B_r B_c + (B_r + B_c)d = O(M)$.*

The proof is in Appendix D; the only auxiliary HBM beyond inputs, outputs, the saved row state (L, D), and the forward output O is the two length-N vectors α, E.

4.4 I/O complexity and large-cache optimality

We analyze HBM traffic in the standard two-level memory model from §3. SRAM has capacity M elements, and HBM traffic counts elements moved between HBM and SRAM. Let $B_r^{(1)}, B_c^{(1)}$ denote the row and column block sizes in Pass 1, and $B_r^{(2)}, B_c^{(2)}$ those in Pass 2. In this subsection, we focus on the large-cache regime where $M = \Omega(d^2)$.

Theorem 4.3 (HBM traffic)**.** *For attention with N rows and head dimension d, Algorithm 1 performs $Q_{\text{FLASHBOB}} = 12Nd + 4N + \frac{4dN^2}{B_r^{(1)}} + \frac{(3d+4)N^2}{B_c^{(2)}}$ HBM transfers, up to boundary effects.*

Remark. Only the reused block sizes $B_r^{(1)}$ and $B_c^{(2)}$ appear in Theorem 4.3: the streamed dimension cancels inside each pass. The $12Nd + 4N$ term collects input loads, row-metadata reads, and output writes. Every $N \times N$ quantity is recomputed and consumed within SRAM before moving to the next block. The per-pass ledger is given in Appendix H.

Corollary 4.4 (Large-cache upper bound)**.** *Assume $d^2 \lesssim M \lesssim Nd$ and $N \geq d$. With $B_r^{(1)} = B_c^{(2)} = b$, $B_c^{(1)} = B_r^{(2)} = \min(b, d)$, and $b = \Theta(M/d)$,* FLASHBOB *has HBM traffic $Q_{\text{FLASHBOB}} = \Theta(N^2d^2/M)$.*

Remark. Indeed, substituting $b = \Theta(M/d)$ into Theorem 4.3 gives $Q_{\text{FLASHBOB}} = 12Nd + 4N + (7d + 4)N^2/b = 12Nd + 4N + \Theta(N^2d^2/M)$. Under $M \lesssim Nd$, the $\Theta(N^2d^2/M)$ term dominates the stationary $O(Nd)$ traffic.

Theorem 4.5 (Large-cache lower bound under score recomputation)**.** *Assume $M = \Omega(d^2)$, $N \geq d$, $M \lesssim Nd$, and $\Theta(N^2)$ valid query-key interactions. This includes dense attention and causal attention, since causal masking leaves $N(N + 1)/2 = \Theta(N^2)$ valid interactions. In the standard FlashAttention-style score-recomputation model formalized in Appendix E, every exact attention-BoB algorithm in this model requires $\Omega(N^2d^2/M)$ HBM transfers.*

The proof is given in Appendix E. The reduction sets $V = 0$, $dO = 0$, $U_Q = U_K = 0$, and $U_V = W$, giving $F = 0$, $\alpha = 0$, $h = 0$, and $\widetilde{d}(dO) = PW = \text{softmax}(\tau QK^\mathsf{T})W$. Thus the lower bound is inherited from the large-cache exact-attention lower bound of [48, Lemma 3.4]. This is a conditional optimality statement: it applies to exact FlashAttention-style score-recomputation algorithms, not to arbitrary exact algorithms that use a different computational model.

Combining Corollary 4.4 with Theorem 4.5, FLASHBOB has $\Theta(N^2d^2/M)$ HBM traffic in the regime $d^2 \lesssim M \lesssim Nd$, matching the score-recomputation lower bound up to stationary $O(Nd)$ input, output, and row-state traffic.

5 Experiments

We evaluate FLASHBOB at three levels: isolated attention BoB on A100 and B300 (§5.1), full-model second-order steps across GPT-2, Pythia, and five native GQA and MoE models on B200 and B300 (§5.2), and 1B-token Sophia-H pretraining runs on 8×A6000 and 8×A100 (§5.3). For the main causal-attention experiments, we use PyTorch's `nn.scaled_dot_product_attention` fused backend for the forward and first backward, and replace only the attention double backward with our two-pass Triton kernel. Experiments requiring features not covered by PyTorch, such as the sliding-window experiments in Appendix M, use the `flash-attn` package. Correctness, additional FlashBack details, and software environment configurations are in Appendices J, P, and B.

5.1 Attention-layer kernel scaling

We compare four exact attention-BoB implementations: MATH, the PyTorch materializing reference; HVP-M and HVP-S, the GradMem-derived hand-written HVP kernels [27]; and FLASHBOB, the two-pass schedule of Algorithm 1. The HVP baselines avoid full materialization but do not preserve FlashAttention-style tile streaming at the BoB level. Table 1 reports the sequence-length sweeps at $d = 64$ (A100 80GB) and $d = 128$ (B300); the full $d = 64$ sweep from N=256 through N=262,144 appears in Appendix K.

At $d = 64$ on A100, the speedup of FLASHBOB over MATH grows with N (from 1.9× at N=256 to 10.3× at N=4,096), and the memory ratio grows from 2.2× to 28.0×. FLASHBOB is fastest at every tested length, reaching 3.8× over HVP-S at N=8,192. MATH runs out of memory at N=8,192 and both HVP kernels at N=16,384, whereas FLASHBOB continues to N=262,144 at 52.7 GiB peak (53,952 MiB; Appendix K). This N=262,144 measurement is an isolated attention-layer result rather than a full-model training length: at N=32,768, the isolated FLASHBOB kernel uses 6,744 MiB on A100, whereas a complete GPT-2 Small second-order step uses 128,392 MiB on B200 because non-attention activations dominate full-step memory (§5.2). At $d = 128$ on B300 (Table 1, right), FLASHBOB is 2.08–2.36× faster than the fastest surviving baseline across N=2,048–16,384, reduces peak memory by up to 50.6× (4,422 MiB vs. 223,829 MiB at N=16,384), and alone runs at N=32,768 and 65,536. A fixed-N sweep across $d \in \{32, 64, 128\}$ on A100 appears in Appendix N.

FlashBack comparison. FlashBack [13] is the closest prior non-materializing exact BoB attention kernel. Table 2 compares both kernels on A6000 ($d = 64$) and B300 ($d = 128$, with both kernels

Table 1: Isolated attention-layer BoB scaling at $d = 64$ (NVIDIA A100 80GB) and $d = 128$ (NVIDIA B300), with batch 4, 12 heads, causal mask, and BF16 inputs with FP32 reductions. Peak memory reports the lowest-memory baseline that fits (HVP-M) versus FLASHBOB; full four-method memory tables through N=262,144 are in Appendix K. OOM denotes out of memory; best values are bold.

	Head dimension $d = 64$ (NVIDIA A100 80GB)						Head dimension $d = 128$ (NVIDIA B300)					
	Wall-clock time (ms)				Peak mem. (MiB)		Wall-clock time (ms)				Peak mem. (MiB)	
N	MATH	HVP-M	HVP-S	FLASHBOB	Base.	FLASHBOB	MATH	HVP-M	HVP-S	FLASHBOB	Base.	FLASHBOB
256	1.82	1.58	1.66	**0.94**	114	**95**	1.3	1.1	1.7	**0.7**	160	**133**
512	4.95	2.31	2.33	**1.29**	315	**171**	2.1	1.2	1.6	**0.9**	401	**250**
1 024	17.66	7.39	6.41	**2.57**	1 045	**323**	6.8	2.9	3.3	**1.8**	1 207	**485**
2 048	68.80	28.34	22.73	**7.53**	3 805	**630**	24.9	11.6	10.8	**5.2**	3 745	**553**
4 096	265.67	114.89	89.14	**25.80**	14 521	**1 243**	94.2	41.5	39.1	**18.2**	14 401	**1 106**
8 192	OOM	468.93	356.84	**95.11**	55 971	**1 702**	366.3	178.6	160.7	**68.2**	56 498	**2 211**
16 384	OOM	OOM	OOM	**365.73**	OOM	**3 372**	OOM	620.0	602.6	**263.6**	223 829	**4 422**
32 768	OOM	OOM	OOM	**1 449.06**	OOM	**6 744**	OOM	OOM	OOM	**1 037.0**	OOM	**8 844**
65 536	OOM	OOM	OOM	**5 843.14**	OOM	**13 488**	OOM	OOM	OOM	**4 124.2**	OOM	**17 688**

Table 2: Matched isolated-kernel comparison with FlashBack at $d = 64$ (NVIDIA A6000 48GB) and $d = 128$ (NVIDIA B300), using batch 4, 12 heads, causal mask, and BF16 inputs with FP32 reductions. Both kernels are autotuned at each N (at $d = 128$, N=8,192, FlashBack takes 619.00 ms vs. 68.23 ms for FLASHBOB, 9.07×; See Appendix P).

	Head dimension $d = 64$ (NVIDIA A6000 48GB)			Head dimension $d = 128$ (NVIDIA B300)		
N	FlashBack (ms)	FLASHBOB (ms)	Speedup	FlashBack (ms)	FLASHBOB (ms)	Speedup
256	1.94	**0.94**	2.07×	1.22	**0.75**	1.63×
512	6.22	**1.52**	4.09×	3.24	**0.92**	3.52×
1 024	22.16	**3.80**	5.83×	10.69	**1.79**	5.98×
2 048	67.25	**11.73**	5.74×	39.03	**5.19**	7.53×
4 096	262.47	**41.51**	6.32×	156.03	**18.23**	8.56×
16 384	2 421.36	**651.28**	3.72×	2 451.53	**263.65**	9.30×
32 768	9 659.78	**2 682.60**	3.60×	9 754.58	**1 037.03**	9.41×
65 536	38 442.11	**10 729.20**	3.58×	38 902.59	**4 124.16**	9.43×

autotuned over block sizes, stages, and warp counts). FLASHBOB is faster at every length, achieving 2.07–6.32× speedups at $d = 64$ and 1.63–9.43× at $d = 128$ across N=256–65,536 (Appendix P). In a matched MQA benchmark on B300 ($H_Q/H_{KV} = 8/1$, batch 1, N=2,048–131,072), FLASHBOB is 3.96–5.04× faster at $d = 64$ and 5.87–8.90× faster at $d = 128$. Whereas FlashBack's row-parallel single-kernel design accumulates $(\widetilde{d}(K), \widetilde{d}(V))$ through contended global atomic adds, FLASHBOB assigns these outputs to a column-major pass and writes each output tile once.

5.2 Full-model benchmarks

We replace the attention double backward inside complete language models across three architecture groups: GPT-2 Small, Medium, and Large ($d = 64$, MHA on B200), Pythia-160M and Pythia-410M [4] (GPT-NeoX architecture, $d = 128$, MHA on B300), and five released native GQA and MoE models on B300 (Llama 3.2 1B, SmolLM2-135M, SmolLM2-360M, Qwen2.5-1.5B, and Granite 3.1 1B-A400M). Non-attention components are unchanged, so the isolated kernel gains upper-bound the whole-model gains. Tables 3 and 4 report representative configurations; full per-N sweeps for all ten models appear in Appendix L.

FLASHBOB is fastest at every tested configuration. At short N, peak memory is dominated by non-attention activations, so FLASHBOB and the HVP baselines use similar memory; the gap opens once quadratic attention dominates the step. On GPT-2 at the longest sequence lengths where a baseline still fits (N=16,384 for Small and N=8,192 for Medium and Large; Appendix L), FLASHBOB is 2.14–2.18× faster than HVP-S and reduces peak memory by 28.6–56.0%, while continuing to N=32,768 (Small) and N=16,384 (Medium/Large) after all baselines exhaust memory. A sliding-window sweep appears in Appendix M.

Beyond GPT-2: Pythia ($d = 128$) and native GQA/MoE models. On Pythia-160M and Pythia-410M at $d = 128$ on B300 (Table 3, bottom blocks), FLASHBOB achieves 1.98× speedup at N=16,384 (160M) and 1.89× at N=8,192 (410M) over HVP-S, and alone reaches N=32,768 and N=16,384. On the five native GQA and MoE models on B300 (Table 4), FLASHBOB operates directly on the native H_Q/H_{KV} layout without repeating key/value heads (whereas HVP-M/S require

Table 3: End-to-end second-order model steps on MHA architectures: **(a)** GPT-2 ($d = 64$, NVIDIA B200 192 GiB) and **(b)** Pythia (GPT-NeoX, $d = 128$, NVIDIA B300), with batch 4 and BF16 inputs. For GPT-2, Base. reports the lowest-memory baseline that fits (HVP-M); full four-method sweeps are in Appendix L.

(a) GPT-2 ($d = 64$, NVIDIA B200 192 GiB)

		Wall-clock time (ms)				Peak mem. (MiB)	
Model	N	MATH	HVP-M	HVP-S	FLASHBOB	Base.	FLASHBOB
Small	4 096	397.7	417.5	304.0	**190.9**	**17 649**	17 804
	8 192	OOM	1537.5	1017.3	**509.1**	46 965	**33 604**
	32 768	OOM	OOM	OOM	**5368.3**	OOM	**128 392**
Med.	2 048	312.6	343.0	255.9	**177.2**	**16 612**	16 810
	4 096	OOM	1028.5	726.1	**420.9**	30 480	**28 677**
	16 384	OOM	OOM	OOM	**3878.9**	OOM	**99 912**
Large	2 048	562.2	614.6	450.4	**304.3**	**27 666**	28 045
	4 096	OOM	1879.6	1313.4	**740.3**	49 205	**45 457**
	16 384	OOM	OOM	OOM	**7106.5**	OOM	**150 019**

(b) Pythia ($d = 128$, NVIDIA B300)

		Wall-clock time (ms)				Mem. (MiB)
Model	N	MATH	HVP-M	HVP-S	FLASHBOB	FLASHBOB
160M	2 048	152.4	161.4	142.7	**115.7**	**16 702**
	4 096	420.8	437.0	361.7	**239.8**	**32 222**
	16 384	OOM	5069.5	4088.9	**2064.2**	**126 273**
	32 768	OOM	OOM	OOM	**7114.6**	**250 435**
410M	2 048	368.0	393.8	328.7	**242.9**	**25 992**
	4 096	1074.1	1125.0	918.4	**583.7**	**48 974**
	8 192	OOM	4121.5	3121.7	**1652.8**	**94 937**
	16 384	OOM	OOM	OOM	**5280.2**	**186 863**

Table 4: Complete second-order steps on native GQA and MoE models at the longest sequence length completed by both methods (batch 1, BF16, one NVIDIA B300). Max-N gain is from power-of-two capacity sweeps; full per-N sweeps are in Appendix L.

				Wall-clock time (ms)			Peak memory (GiB)			
Model	Attention (H_Q/H_{KV})	d	N	MATH	FLASHBOB	Speedup	MATH	FLASHBOB	Red.	Max-N gain
Llama 3.2 1B	Dense, GQA (32/8)	64	4 096	653.3	**247.1**	2.64×	152.5	**39.8**	73.9%	4×
SmolLM2-135M	Dense, GQA (9/3)	64	8 192	1 208.1	**319.4**	3.78×	249.2	**27.9**	88.8%	8×
SmolLM2-360M	Dense, GQA (15/5)	64	4 096	599.3	**212.3**	2.82×	122.7	**24.5**	80.1%	8×
Qwen2.5-1.5B	Dense, GQA (12/2)	128	4 096	568.5	**365.4**	1.56×	128.6	**59.9**	53.4%	4×
Granite 3.1 1B-A400M	MoE, GQA (16/8)	64	4 096	593.2	**272.4**	2.18×	118.5	**38.7**	67.3%	8×

head replication and FlashBack supports only MQA with $H_{KV} = 1$). At the largest shared sequence length for each model, FLASHBOB is 1.56–3.78× faster than PyTorch MATH, reduces peak memory by 53.4–88.8%, and reaches 4–8× longer maximum sequence lengths in power-of-two capacity sweeps (SmolLM2 runs past 8,192 tokens use RoPE extrapolation and test hardware capacity only).

5.3 Training on FineWeb

We train GPT-2 Small on 1B tokens from FineWeb at N=2048 on 8×NVIDIA A6000 48GB, with per-GPU batch size 2 and gradient accumulation 16 (effective batch 256, 1,908 optimizer steps). Following [32], we estimate the Hessian diagonal once every ten steps. We compare AdamW, Sophia-H with the PyTorch math reference, and Sophia-H with FLASHBOB. The AdamW curve is the best-performing AdamW reference from a small learning-rate sweep, while the two Sophia-H runs use matched optimizer, model, data-streaming, precision, seed, batch, and Hessian-estimation settings; the only intended difference between them is the attention BoB backend used during Hessian estimation. Full training hyperparameters are given in Appendix O.

PyTorch's `scaled_dot_product_attention` does not support double backward in any accelerated backend, so an exact second-order pipeline using the standard PyTorch path must fall back to the `math` backend for the attention path. Because the graph must remain compatible with double backward, the math-reference Sophia-H run loses the optimized attention path even on ordinary gradient steps. In contrast, FLASHBOB preserves the optimized forward and first backward and replaces only the attention double backward, reducing per-step non-Hessian time from ≈4.9 s to ≈2.3 s.

FLASHBOB accelerates the Hessian-estimation event by 2.45× and the full training loop by 2.17× end to end, while reducing peak memory by 2.22× (from 30 GB to 14 GB per GPU). The two Sophia-H loss curves nearly overlap throughout training (Figure 2), with matched final losses of 3.6674 (PyTorch math) and 3.6662 (FLASHBOB), compared with 3.6759 for AdamW. In elapsed time, FLASHBOB shifts the Sophia-H curve substantially left toward the AdamW time-to-loss curve while preserving the final Sophia-H loss.

Separating kernel and integration gains. The 2.17× full-loop speedup combines a faster BoB kernel on Hessian steps with retaining FlashAttention on ordinary gradient steps. To isolate the kernel contribution, we implemented a matched hybrid baseline that uses FlashAttention for the forward and

Table 5: GPT-2 Small training on 1B FineWeb tokens at $N = 2048$ (1,908 steps, effective batch 256). **(a)** Standard PyTorch math-reference versus FLASHBOB Sophia-H with Hessian estimation every 10 steps on 8×A6000 48GB. **(b)** Matched hybrid baseline (FlashAttention forward and first backward in both arms; only the attention BoB differs) across Hessian frequencies on 8×A100 80GB.

(a) Standard vs. FLASHBOB Sophia-H (8×A6000)

Metric	Math ref.	FLASHBOB	Ratio
Hessian estimation / event (ms)	13,500	**5,500**	2.45×
Total Hessian time (min)	42.8	**17.4**	2.45×
Total non-Hessian time (hr)	2.59	**1.23**	2.10×
Full training wall time (hr)	3.30	**1.52**	2.17×
Peak memory per GPU (MiB)	30,644	**13,830**	2.22×

(b) Matched hybrid baseline (8×A100 80GB)

Hessian interval	Hybrid (hr)	FLASHBOB (hr)	Speedup
Every 10 steps	0.77	**0.63**	1.22×
Every 5 steps	1.12	**0.79**	1.42×
Every 2 steps	2.11	**1.24**	1.70×

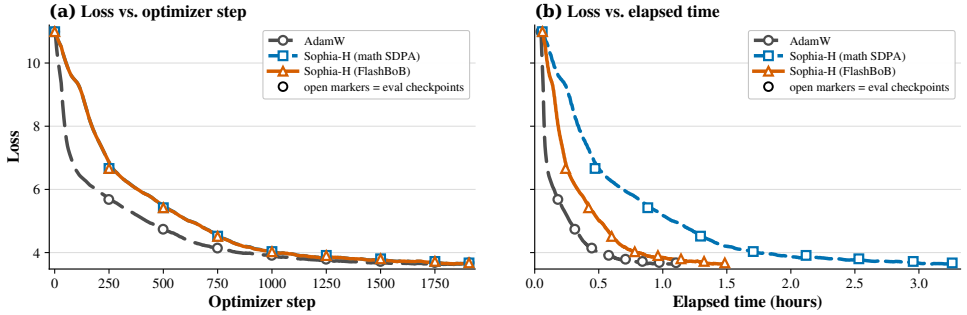


Figure 2: Single-seed systems parity check. Final losses are 3.6759 for AdamW, 3.6674 for Sophia-H with the PyTorch math backend, and 3.6662 for Sophia-H with FLASHBOB; the elapsed-time panel shows that FLASHBOB preserves the Sophia-H trajectory while moving exact second-order training closer to AdamW in time-to-loss.

first backward on all steps and differs only in the attention BoB implementation used during Hessian estimation (PyTorch `math` versus FLASHBOB). Across six matched 1B-token GPT-2 Small runs on 8×A100 80GB with Hessian estimation every 10, 5, and 2 steps (Table 5, right), the hybrid baseline requires 0.77, 1.12, and 2.11 hours, whereas FLASHBOB requires 0.63, 0.79, and 1.24 hours (1.22×, 1.42×, and 1.70× end-to-end speedup; See Appendix O).

6 Discussion

Long-context learning methods such as second-order optimization, test-time training, gradient-written memory, and meta-learning differentiate through gradient computations. When these computations contain softmax attention, they require backward-over-backward (BoB). FLASHBOB extends FlashAttention's tile-streaming execution model to this operation, computing exact BoB without materializing $N \times N$ intermediates. Across complete-model second-order steps, it achieves 1.6–3.8× speedups over the available exact baselines and supports 2–8× longer sequences in the tested memory-capacity sweeps. The $N = 262{,}144$ result is an isolated attention-layer measurement on one A100 80GB GPU, rather than a full-model training context. On FineWeb, FLASHBOB preserves the Sophia-H loss trajectory while reducing training wall time by 2.17× and peak memory by 2.22× relative to the PyTorch math-backend pipeline. The matched hybrid baseline shows 1.22–1.70× end-to-end gains when only the attention BoB implementation changes. Evaluating these benefits within test-time training and gradient-written memory remains future work. It also makes a new design point practical: attention-aware post-training quantization based on exact Hessian-vector products [51, 12], where existing methods use approximations or layer-local surrogates [37, 18, 24]; we have not evaluated this direction.

Limitations. Our Triton implementation does not yet use architecture-specific asynchronous pipelines, warp specialization, or tensor-core scheduling for Hopper and Blackwell. The released kernels support standard multi-head attention and native grouped-query and multi-query attention (GQA/MQA). The GQA/MQA path currently supports BF16, $N_Q = N_{KV}$, $H_Q/H_{KV} \leq 32$, and causal masking without sliding windows; rectangular attention is supported for multi-head attention. Extensions to flexible attention APIs and other attention normalizations [49] require additional implementation or derivation. We view the two-pass algorithm as a foundation for these specializations.

Acknowledgments and Disclosure of Funding

We are especially grateful to Modal Labs for their generous provision of GPU compute resources and support throughout this research. Their contribution was invaluable in enabling the extensive experimental evaluation of FLASHBOB, including the additional benchmarks and validation conducted during the review process. We sincerely appreciate Modal's generosity and commitment to supporting academic research. This work is supported by NSF grants #2436217, #2425687, #2601681.

References

[1] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[2] 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`.

[3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[4] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, Usvsn Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. Pythia: A suite for analyzing large language models across training and scaling. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 2397–2430. PMLR, 2023. URL `https://proceedings.mlr.press/v202/biderman23a.html`.

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016. URL `https://arxiv.org/abs/1604.06174`.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594. USENIX Association, 2018.

[7] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[9] Felix Dangel, Frederik Kunstner, and Philipp Hennig. BackPACK: Packing more into backprop. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJlrF24twB`.

[10] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, 2024.

[11] 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 35*, pages 16344–16359, 2022.

[12] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/d77c703536718b95308130ff2e5cf9ee-Abstract.html`.

[13] Logan Engstrom and Axel Feldmann. Flashback: A FlashAttention backwards-over-backwards. https://github.com/lengstrom/flashback, 2024. GitHub repository. Accessed: 2026-03-01.

[14] Jiarui Fang and Shangchun Zhao. USP: A unified sequence parallelism approach for long context generative AI, 2024. URL https://arxiv.org/abs/2405.07719.

[15] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135, 2017.

[16] Daniel Y. Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. FlashFFTConv: Efficient convolutions for long sequences with tensor cores. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=gPKTTAfYBp.

[17] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=tEYskw1VY2.

[18] Ziyi Guan, Hantao Huang, Yupeng Su, Hong Huang, Ngai Wong, and Hao Yu. APTQ: Attention-aware post-training mixed-precision quantization for large language models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024. doi: 10.1145/3649329.3658498.

[19] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 2956–2964, 2018.

[20] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[21] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. DeepSpeed Ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023. URL https://arxiv.org/abs/2309.14509.

[22] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[23] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, 2020.

[24] Junhan Kim, Ho-Young Kim, Eulrang Cho, Chungman Lee, Joonyoung Kim, and Yongkweon Jeon. BoA: Attention-aware post-training quantization without backpropagation. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 30132–30152. PMLR, 2025. URL https://proceedings.mlr.press/v267/kim25j.html.

[25] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Vfs_2RnOD0H.

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

[27] Yuri Kuratov, Matvey Kairov, Aydar Bulatov, Ivan Rodkin, and Mikhail Burtsev. Grad-Mem: Learning to write context into memory with test-time gradient descent. *arXiv preprint arXiv:2603.13875*, 2026. doi: 10.48550/arXiv.2603.13875.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles*, pages 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

[29] Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Xuezhe Ma, Ion Stoica, Joseph E. Gonzalez, and Hao Zhang. DISTFLASHATTN: Distributed memory-efficient attention for long-context LLMs training. In *First Conference on Language Modeling*, 2024. doi: 10.48550/arXiv.2310.03294. URL https://openreview.net/forum?id=pUEDkZyPDl.

[30] Xiaoyu Li, Yingyu Liang, Zhenmei Shi, Zhao Song, Song Yue, and Jiahao Zhang. On fine-grained I/O complexity of attention backward passes. *arXiv preprint arXiv:2410.09397*, 2024. doi: 10.48550/arXiv.2410.09397.

[31] Hao Liu, Matei Zaharia, and Pieter Abbeel. RingAttention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=WsRHpHH4s0.

[32] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. In *International Conference on Learning Representations*, 2024.

[33] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance and precision. In *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops*, pages 522–531. IEEE, 2018. doi: 10.1109/IPDPSW.2018.00091.

[34] James Martens. Deep learning via hessian-free optimization. In *International Conference on Machine Learning*, pages 735–742, 2010.

[35] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International Conference on Machine Learning*, pages 2408–2417, 2015.

[36] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax, 2018. URL https://arxiv.org/abs/1805.02867.

[37] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7197–7206. PMLR, 2020. URL https://proceedings.mlr.press/v119/nagel20a.html.

[38] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[39] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. doi: 10.1145/1365490.1365500.

[40] 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 Z. 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, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[41] Barak A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160, 1994.

[42] Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems 37*, 2024. doi: 10.52202/079017-0970. Datasets and Benchmarks Track.

[43] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocoń, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanisław Woźniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: Reinventing RNNs for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[44] Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y. Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, 2023.

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

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–530. ACM, 2013. doi: 10.1145/2491956.2462176.

[47] Aravind Rajeswaran, Chelsea Finn, Sham Kakade, and Sergey Levine. Meta-learning with implicit gradients. In *Advances in Neural Information Processing Systems 32*, 2019.

[48] Barna Saha and Christopher Ye. The I/O complexity of attention, or how optimal is Flash Attention? *arXiv preprint arXiv:2402.07443*, 2024. doi: 10.48550/arXiv.2402.07443.

[49] Michael E. Sander, Pierre Ablin, Mathieu Blondel, and Gabriel Peyré. Sinkformers: Transformers with doubly stochastic attention. In *Proceedings of the 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 3515–3530. PMLR, 2022. URL `https://proceedings.mlr.press/v151/sander22a.html`.

[50] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024. doi: 10.48550/arXiv.2407.08608.

[51] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Q-BERT: Hessian based ultra low precision quantization of BERT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020. doi: 10.1609/aaai.v34i05.6409.

[52] 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. doi: 10.48550/arXiv.2512.23675.

[53] 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*, pages 10–19, 2019.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, 2017.

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

[56] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W. Mahoney. ADAHESSIAN: An adaptive second order optimizer for machine learning. In *AAAI Conference on Artificial Intelligence*, volume 35, pages 10665–10673, 2021. doi: 10.1609/aaai.v35i12.17275.

[57] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flash-Infer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. doi: 10.48550/arXiv.2501.01005. URL `https://proceedings.mlsys.org/paper_files/paper/2025/hash/dbf02b21d77409a2db30e56866a8ab3a-Abstract-Conference.html`.

[58] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026. doi: 10.48550/arXiv.2603.05451.

[59] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems 33*, 2020.

A Notation

We collect the main symbols used throughout the paper in Table 6. Shapes use the square, single-head attention setting of Section 3; batch and head dimensions are omitted. Rectangular attention and grouped-query attention are described in Sections 3 and 4.3. We use t_i for the i-th row of a matrix T; row scalars L, D, α, and E are indexed by query row. For an input or intermediate X, $\widetilde{d}(X)$ denotes $\partial\Phi/\partial X$, where Φ is the BoB scalar defined in (4).

Table 6: Notation used throughout the paper. Tile-level intermediates ($F, C, \widetilde{d}(P), \widetilde{d}(S)$) are introduced in Section 4.

Symbol	Shape	Meaning
Q, K, V, dO	$\mathbb{R}^{N\times d}$	Forward inputs, first-backward upstream
B_{mask}	$\{0, -\infty\}^{N\times N}$	Additive attention mask
O	$\mathbb{R}^{N\times d}$	Forward output, $O = PV$
L	$\mathbb{R}^{N}$	Saved row log-normalizer, $L_i = \log\sum_{j\in\mathcal{V}_i}\exp(S_{ij})$
dQ, dK, dV	$\mathbb{R}^{N\times d}$	First-backward outputs
D	$\mathbb{R}^{N}$	Saved first-backward row summary, $D_i = do_i^{\mathsf{T}} o_i$
U_Q, U_K, U_V	$\mathbb{R}^{N\times d}$	BoB input matrices
$\widetilde{d}(Q), \widetilde{d}(K), \widetilde{d}(V), \widetilde{d}(dO)$	$\mathbb{R}^{N\times d}$	BoB outputs
α, E	$\mathbb{R}^{N}$	BoB row scalars (Section 4)
M	scalar	On-chip SRAM capacity in scalar elements
B_r, B_c, T_r, T_c	scalar	Row/column tile sizes and tile counts

B Software environment

Table 7 reports the software stack used for the experiments. We report the CUDA version from the PyTorch/kernel stack rather than the maximum CUDA compatibility version reported by `nvidia-smi`. We separately report the `nvcc` compiler release and the `nvidia-smi` CUDA compatibility version because these quantities can differ from `torch.version.cuda`. The A6000 experiments used Python 3.10.20, while the A100, B200, and B300 experiments used Python 3.10.13.

Measurement protocol. Unless noted otherwise, kernel and model benchmarks run 25 warmup iterations followed by 100 CUDA-synchronized timed iterations and report the median CUDA-event latency (p_{50}). Sliding-window experiments use `triton.testing.do_bench` with `warmup=25` and

Table 7: Software stack used for experiments. Under CUDA versions, Runtime denotes `torch.version.cuda`, `nvcc` denotes the compiler release used for custom kernels, and `smi` denotes the maximum CUDA version reported by `nvidia-smi`. N/A indicates that standalone `flash-attn` was not required.

	Python packages			CUDA versions			
Hardware	PyTorch	Triton	`flash-attn`	Runtime	`nvcc`	`smi`	Driver
A6000 (1× / 8×)	2.9.1+cu128	3.5.1	2.8.4	12.8	12.8	13.0	580.76.05
A100 80GB (1× / 8×)	2.11.0.dev20260123 +cu126	3.6.0	2.8.3	12.6	12.6	12.6	560.35.05
B200 192GB	2.11.0+cu130	3.6.0	N/A	13.0	13.1	13.2	595.58.03
B300	2.11.0+cu130	3.6.0	N/A	13.0	13.1	13.2	595.58.03

`rep=100`, where both arguments specify millisecond budgets rather than fixed iteration counts; entries written as $x \pm \delta$ report the median and half the 20th-to-80th percentile range,

$$x = p_{50}, \qquad \delta = \frac{1}{2}(p_{80} - p_{20}),$$

rather than a standard deviation. Multi-GPU training results on $8\times$A6000 and $8\times$A100 report full end-to-end wall-clock time; full-model and training timings are single-run measurements and do not estimate run-to-run variation.

Peak memory is measured via `torch.cuda.max_memory_allocated` and therefore includes all tensors held live during the benchmark, not only kernel workspace. In particular, the $d = 64$ A100 sweep in Table 1 retains the MATH reference outputs for $N \leq 4{,}096$, whereas the head-dimension sweep in Appendix N releases them before timing.

FlashBack additionally used JAX 0.6.2 and `jaxlib` 0.6.2; we do not compare its JAX/XLA peak-memory measurements directly with PyTorch's allocated-memory statistic (Appendix P). The main causal-attention experiments use PyTorch's fused scaled-dot-product attention (SDPA) for the forward and first backward; FLASHBOB replaces only the double backward. SDPA need not select the same underlying fused kernel for every GPU and input shape. The standalone `flash-attn` package was used for the sliding-window experiments on A100 (Appendix M).

The GradMem reference implementations, HVP-M and HVP-S, were evaluated at commit `3f40f11`. FlashBack was evaluated at commit `62a0e0e` with its own autotuned configurations. The Sophia-H baseline was evaluated at commit `738c32b`, the last commit in which Sophia-H was present. Random seeds were fixed per benchmark. Between configurations, we called `torch.cuda.empty_cache()` and reset peak memory statistics before each trial. Measurements used the same PyTorch CUDA allocator configuration across compared methods unless otherwise noted.

B.1 Example: Hessian-vector products through attention

A common backward-over-backward workload computes a Hessian-vector product (HVP) of a model loss $\mathcal{L}$ with respect to its parameters θ. Given a direction z of the same shape as θ, we first compute $g = \nabla_\theta \mathcal{L}$ with an autograd graph and then compute $H_\theta z = \nabla_\theta \langle g, z \rangle$. This differentiates through the attention backward without constructing the Hessian. Within the model, replace the scaled-dot-product attention call with `sdpa_bob`; all other model operations remain unchanged:

```
import torch
from bob import sdpa_bob

# Inside the model's attention block (q, k, v: [B, H, N, d]):
y = sdpa_bob(q, k, v, is_causal=True)

# Inside a curvature-estimation step for a model using this block:
params = tuple(p for p in model.parameters() if p.requires_grad)
loss = loss_fn(model(inputs), targets)
g = torch.autograd.grad(loss, params, create_graph=True)
```

```
z = [torch.randn_like(p) for p in params]
phi = sum((gi.float() * zi.float()).sum() for gi, zi in zip(g, z))
Hv = torch.autograd.grad(phi, params)
```

Here `model`, `inputs`, `targets`, and `loss_fn` are supplied by the training program; the model's attention modules call `sdpa_bob`. The returned `Hv` contains one Hessian-vector product for each parameter tensor. A Hutchinson-style diagonal estimate can further use the elementwise product $z \odot (H_\theta z)$ when z has independent, zero-mean, unit-variance entries. This is the curvature primitive underlying second-order optimizer examples, not a complete Sophia-H training step. Our repository provides a runnable model-parameter HVP example at `https://github.com/MingruiLiu-ML-Lab/FlashBoB/blob/main/examples/hessian_vector_product.py`.

B.2 Software and dataset licenses

We acknowledge the following software, models, and datasets used in our experiments or as baselines. The listed licenses describe the referenced distributions; code, model weights, and datasets may have different terms. Consult the upstream releases for their applicable licenses and conditions.

- **PyTorch** (BSD-3-Clause), `https://github.com/pytorch/pytorch`.
- **Triton** (MIT), `https://github.com/triton-lang/triton`.
- **flash-attn** (BSD-3-Clause), `https://github.com/Dao-AILab/flash-attention`.
- **JAX** and **jaxlib** (Apache-2.0), `https://github.com/jax-ml/jax`.
- **FineWeb-derived training data**. The training runs use `VisionTheta/fineweb-1B`, a FineWeb-derived 1B-token dataset stream. The underlying FineWeb dataset [42] is distributed under the Open Data Commons Attribution License v1.0 (ODC-By) at `https://huggingface.co/datasets/HuggingFaceFW/fineweb`; use is also subject to CommonCrawl's Terms of Use.
- **GPT-2**, **Pythia**, **Llama 3.2**, **SmolLM2**, **Qwen2.5**, and **Granite 3.1** model architectures and configurations: GPT-2 (MIT), Pythia (Apache-2.0), Llama 3.2 (Llama 3.2 Community License), SmolLM2 (Apache-2.0), Qwen2.5 (Apache-2.0), and Granite 3.1 (Apache-2.0).
- **Sophia** reference implementation [32] (`https://github.com/Liuhong99/Sophia`). The repository did not display a top-level LICENSE file at commit `738c32b`; we used the code only as a research baseline. The repository documents that its training code is derived from nanoGPT (MIT) and levanter (Apache-2.0).
- **GradMem** reference HVP kernels [27], evaluated at commit `3f40f11`. Used solely as a research baseline under the terms of the upstream repository.
- **FlashBack** [13] (`https://github.com/lengstrom/flashback`), evaluated at commit `62a0e0e`. The repository did not display a top-level LICENSE file at this commit; we used the code only as a research baseline.
- **NVIDIA CUDA Toolkit and drivers**: used under the NVIDIA CUDA Toolkit End User License Agreement.

C Full BoB derivation

This appendix derives every BoB identity in §4.2 from reverse-mode automatic differentiation applied to the first-backward graph of softmax attention. The derivation is deliberately step-by-step: each node is processed in turn, every local Jacobian is computed from scratch from its defining formula, and every chain-rule sum is written explicitly before being simplified. Readers familiar with the chain rule and the multi-child rule have all the prerequisites.

Notation summary

Throughout this appendix we use the following two conventions in parallel. For any node X in the first-backward graph, $dX = \partial L/\partial X$ is its first-order gradient (already computed by the ordinary attention backward), and $\widetilde{d}(X) = \partial\Phi/\partial X$ is its BoB gradient (what we are computing here). The

first-order gradients dQ, dK, dV, dP, dS, dO together with the saved row scalar D are inputs to BoB; the BoB gradients $\widetilde{d}(Q), \widetilde{d}(K), \widetilde{d}(V), \widetilde{d}(dO), \widetilde{d}(P), \widetilde{d}(S), \widetilde{d}(dP), \widetilde{d}(dS)$ are intermediates or outputs.

The first-backward outputs dQ, dK, dV are the entry points of the BoB walk; reverse-mode begins with their BoB gradients

$$\widetilde{d}(dQ) = U_Q, \qquad \widetilde{d}(dK) = U_K, \qquad \widetilde{d}(dV) = U_V, \tag{9}$$

which follow directly from differentiating the BoB scalar $\Phi = \langle dQ, U_Q\rangle + \langle dK, U_K\rangle + \langle dV, U_V\rangle$ of (4).

For matrices A, B of the same shape, $\langle A, B\rangle = \sum_{i,j} A_{ij}B_{ij}$ is the Frobenius inner product. Lowercase letters denote row slices: $q_i, k_j, v_j, do_i, u_{Q,i}, u_{K,j}, u_{V,j} \in \mathbb{R}^d$. Whenever a row vector $D, \alpha, E \in \mathbb{R}^N$ appears inside an $N \times N$ expression, it is broadcast across columns.

Reverse-mode chain rule

The only fact used in this appendix is the chain rule for a node X with children $Y^{(1)}, \ldots, Y^{(m)}$ in the first-backward graph (meaning that each $Y^{(r)}$ is computed from X during the first-backward pass): the BoB gradient at X is the sum of contributions through each child, with each contribution given by the local Jacobian acting on the BoB gradient at the child. In components,

$$\widetilde{d}(X_{ij}) \;=\; \sum_{r=1}^{m} \sum_{u,v} \widetilde{d}(Y^{(r)}_{uv}) \, \frac{\partial Y^{(r)}_{uv}}{\partial X_{ij}}. \tag{10}$$

Every equation that follows is one application of (10).

Step 1: $F = \widetilde{d}(dS)$

The node dS has two children in the first-backward graph, $dQ = \tau\, dS\, K$ and $dK = \tau\, dS^{\mathsf{T}} Q$. Applying (10),

$$\widetilde{d}(dS_{ij}) \;=\; \sum_a \widetilde{d}(dQ_{ia}) \, \frac{\partial dQ_{ia}}{\partial dS_{ij}} \;+\; \sum_a \widetilde{d}(dK_{ja}) \, \frac{\partial dK_{ja}}{\partial dS_{ij}}. \tag{11}$$

The local Jacobians come from the formulas $dQ_{ia} = \tau \sum_j dS_{ij} K_{ja}$ and $dK_{ja} = \tau \sum_i dS_{ij} Q_{ia}$, giving

$$\frac{\partial dQ_{ia}}{\partial dS_{ij}} = \tau\, K_{ja}, \qquad \frac{\partial dK_{ja}}{\partial dS_{ij}} = \tau\, Q_{ia}. \tag{12}$$

Substituting and using $\widetilde{d}(dQ_{ia}) = U_{Q,ia}$ and $\widetilde{d}(dK_{ja}) = U_{K,ja}$ from (9),

$$\widetilde{d}(dS_{ij}) \;=\; \tau \sum_a U_{Q,ia}\, K_{ja} \;+\; \tau \sum_a U_{K,ja}\, Q_{ia} \;=\; \tau\, u_{Q,i}^{\mathsf{T}} k_j \;+\; \tau\, q_i^{\mathsf{T}} u_{K,j}. \tag{13}$$

We name this quantity F. In matrix form,

$$F \;:=\; \widetilde{d}(dS) \;=\; \tau\big(U_Q\, K^{\mathsf{T}} + Q\, U_K^{\mathsf{T}}\big). \tag{14}$$

Step 2: $\widetilde{d}(dP)$ **and the row scalars** α

The node dP has children in the first-backward graph through the formula $dS_{ik} = P_{ik}(dP_{ik} - D_i)$ together with $D_i = \sum_r P_{ir}\, dP_{ir}$. Holding P fixed (its BoB gradient is handled separately in Steps 4 and 5), dP_{ij} enters dS_{ik} through two routes: a direct route (dP_{ij} appears as dP_{ik} when $k = j$) and an indirect route through the row scalar D_i (every k in row i uses D_i, which depends on dP_{ij}). Both contribute to the chain rule, so we compute the local Jacobian carefully:

$$\frac{\partial dS_{ik}}{\partial dP_{ij}} \;=\; P_{ik}\left(\frac{\partial dP_{ik}}{\partial dP_{ij}} - \frac{\partial D_i}{\partial dP_{ij}}\right). \tag{15}$$

The two pieces are

$$\frac{\partial dP_{ik}}{\partial dP_{ij}} = \delta_{kj}, \qquad \frac{\partial D_i}{\partial dP_{ij}} = \frac{\partial}{\partial dP_{ij}} \sum_r P_{ir}\, dP_{ir} = P_{ij}, \tag{16}$$

so

$$\frac{\partial dS_{ik}}{\partial dP_{ij}} \;=\; P_{ik}(\delta_{kj} - P_{ij}). \tag{17}$$

Applying (10) with the single child dS and $\widetilde{d}(dS_{ik}) = F_{ik}$,

$$\widetilde{d}(dP_{ij}) \;=\; \sum_k F_{ik}\, P_{ik}(\delta_{kj} - P_{ij}) \;=\; \underbrace{F_{ij}\, P_{ij}}_{k=j \text{ term}} \;-\; \underbrace{P_{ij} \sum_k P_{ik} F_{ik}}_{\text{row-sum term}}. \tag{18}$$

The row-sum factor $\sum_k P_{ik}F_{ik}$ is a row reduction; we name it α_i:

$$\alpha_i \;:=\; \sum_{k \in \mathcal{V}_i} P_{ik}\, F_{ik}. \tag{19}$$

Substituting,

$$\widetilde{d}(dP_{ij}) \;=\; P_{ij}(F_{ij} - \alpha_i). \tag{20}$$

Step 3: $\widetilde{d}(V)$ and the first contribution to $\widetilde{d}(dO)$

The node V has only one child in the first-backward graph, $dP \;=\; dO\, V^{\mathsf{T}}$; that is, $dP_{ij} \;=\; \sum_a dO_{ia}\, V_{ja}$. Applying (10) for the BoB gradient at V_{ja},

$$\widetilde{d}(V_{ja}) \;=\; \sum_i \widetilde{d}(dP_{ij})\, \frac{\partial dP_{ij}}{\partial V_{ja}} \;=\; \sum_i \widetilde{d}(dP_{ij})\, dO_{ia} \;=\; \sum_i P_{ij}(F_{ij} - \alpha_i)\, dO_{ia}. \tag{21}$$

In matrix form, with $h := F - \alpha\, \mathbf{1}^{\mathsf{T}}$ broadcast over columns,

$$\widetilde{d}(V) \;=\; (P \odot h)^{\mathsf{T}}\, dO. \tag{22}$$

The same node $dP = dO\, V^{\mathsf{T}}$ also sends one of two contributions into $\widetilde{d}(dO)$. For the BoB gradient at dO_{ia},

$$\frac{\partial dP_{ij}}{\partial dO_{ia}} = V_{ja}, \qquad \text{so the contribution from } dP \text{ is} \quad \sum_j \widetilde{d}(dP_{ij})\, V_{ja} = \sum_j P_{ij}(F_{ij} - \alpha_i)\, V_{ja}. \tag{23}$$

We will combine this with the second contribution to $\widetilde{d}(dO)$ in Step 4.

Step 4: C, the contribution to $\widetilde{d}(P)$ from the dV child, and the rest of $\widetilde{d}(dO)$

The node P has two children in the first-backward graph, dV and dS. We process the dV child here (it is algebraically simple) and defer the dS child to Step 5 (where the second row scalar will appear).

The first-backward formula is $dV_{ja} = \sum_i P_{ij}\, dO_{ia}$. The local Jacobian is $\partial dV_{ja}/\partial P_{ij} = dO_{ia}$, and the BoB gradient at dV is $\widetilde{d}(dV_{ja}) = U_{V,ja}$ from (9). The dV-child contribution to $\widetilde{d}(P_{ij})$ is

$$\sum_a U_{V,ja}\, dO_{ia} \;=\; dO_i^{\mathsf{T}} u_{V,j}. \tag{24}$$

We name this quantity C_{ij}. In matrix form,

$$C \;:=\; dO\, U_V^{\mathsf{T}}. \tag{25}$$

The same node $dV = P^{\mathsf{T}} dO$ also sends the second contribution into $\widetilde{d}(dO)$. For the BoB gradient at dO_{ia},

$$\frac{\partial dV_{ja}}{\partial dO_{ia}} = P_{ij}, \qquad \text{so the contribution from } dV \text{ is} \quad \sum_j P_{ij}\, U_{V,ja}. \tag{26}$$

Combining with the contribution from Step 3,

$$\widetilde{d}(dO_{ia}) \;=\; \sum_j P_{ij}\, U_{V,ja} \;+\; \sum_j P_{ij}(F_{ij} - \alpha_i)\, V_{ja}, \quad \text{or} \quad \widetilde{d}(dO) = P\, U_V + (P \odot h)\, V. \tag{27}$$

Step 5: contribution to $\widetilde{d}(P)$ from the dS child

This is the second contribution to $\widetilde{d}(P)$. The variable P_{ij} reaches dS_{ik} along two routes through the formula $dS_{ik} = P_{ik}(dP_{ik} - D_i)$: the outer factor P_{ik} (when $k = j$) and the row scalar D_i (which depends on every P_{ir}). Computing the local Jacobian by the product rule, with dP now held fixed,

$$\frac{\partial dS_{ik}}{\partial P_{ij}} \;=\; \frac{\partial P_{ik}}{\partial P_{ij}}(dP_{ik} - D_i) \;+\; P_{ik}\,\frac{\partial (dP_{ik} - D_i)}{\partial P_{ij}} \tag{28}$$

$$\;=\; \delta_{kj}(dP_{ik} - D_i) \;-\; P_{ik}\,dP_{ij}, \tag{29}$$

where in the second line we used $\partial P_{ik}/\partial P_{ij} = \delta_{kj}$ and $\partial D_i/\partial P_{ij} = dP_{ij}$. Multiplying by $\widetilde{d}(dS_{ik}) = F_{ik}$ and summing over k,

$$\sum_k F_{ik}\,\frac{\partial dS_{ik}}{\partial P_{ij}} \;=\; \underbrace{\sum_k F_{ik}\,\delta_{kj}(dP_{ij} - D_i)}_{\text{outer-factor route}} \;-\; \underbrace{\sum_k F_{ik}\,P_{ik}\,dP_{ij}}_{\text{row-scalar route through } D_i} \tag{30}$$

$$\;=\; F_{ij}(dP_{ij} - D_i) \;-\; \alpha_i\,dP_{ij}, \tag{31}$$

where the last step used the α_i definition from (19). Adding the C_{ij} contribution from (25) closes the BoB gradient at P:

$$\widetilde{d}(P_{ij}) \;=\; \underbrace{C_{ij} + F_{ij}(dP_{ij} - D_i)}_{\widetilde{d}(P_{ij})^{\circ}} - \alpha_i\,dP_{ij}. \tag{32}$$

Equation (32) also defines the α-free part $\widetilde{d}(P_{ij})^{\circ}$ of $\widetilde{d}(P_{ij})$.

Step 6: $\widetilde{d}(S)$ and the row scalars E

The node S has one child, $P = \mathrm{softmax}(S)$. We first derive the softmax Jacobian from scratch by the quotient rule, then apply (10).

For row i, $P_{ik} = \exp(S_{ik})/Z_i$ with $Z_i = \sum_r \exp(S_{ir})$. By the quotient rule,

$$\frac{\partial P_{ik}}{\partial S_{ij}} \;=\; \frac{[\partial \exp(S_{ik})/\partial S_{ij}]\,Z_i \;-\; \exp(S_{ik})\,[\partial Z_i/\partial S_{ij}]}{Z_i^2}. \tag{33}$$

The two numerator pieces are $\partial \exp(S_{ik})/\partial S_{ij} = \delta_{kj}\exp(S_{ik})$ and $\partial Z_i/\partial S_{ij} = \exp(S_{ij})$, so

$$\frac{\partial P_{ik}}{\partial S_{ij}} \;=\; \frac{\delta_{kj}\exp(S_{ik})}{Z_i} \;-\; \frac{\exp(S_{ik})\exp(S_{ij})}{Z_i^2} \;=\; P_{ik}(\delta_{kj} - P_{ij}). \tag{34}$$

Applying (10) at S_{ij} with the single child P,

$$\widetilde{d}(S_{ij}) \;=\; \sum_k \widetilde{d}(P_{ik})\,\frac{\partial P_{ik}}{\partial S_{ij}} \;=\; \sum_k \widetilde{d}(P_{ik})\,P_{ik}(\delta_{kj} - P_{ij}) \;=\; P_{ij}\,\widetilde{d}(P_{ij}) \;-\; P_{ij}\sum_k P_{ik}\,\widetilde{d}(P_{ik}). \tag{35}$$

The row-sum factor is the second row scalar:

$$E_i \;:=\; \sum_{k\in\mathcal{V}_i} P_{ik}\,\widetilde{d}(P_{ik}). \tag{36}$$

Substituting,

$$\widetilde{d}(S_{ij}) \;=\; P_{ij}(\widetilde{d}(P_{ij}) - E_i), \tag{37}$$

the same shape as the first-backward formula $dS_{ij} = P_{ij}(dP_{ij} - D_i)$.

Step 7: $\widetilde{d}(Q)$ and $\widetilde{d}(K)$

The node Q has two children in the first-backward graph: S (through the indirect path $Q \to S \to P \to \cdots$) and dK (through the direct path, since $dK = \tau\,dS^{\mathsf{T}}Q$ uses Q directly). The contribution from each child is computed below.

Indirect contribution through S. The forward formula is $S_{ij} = \tau\, q_i^{\mathsf{T}} k_j + (B_{\text{mask}})_{ij}$, so the local Jacobian is $\partial S_{ij}/\partial Q_{ia} = \tau\, K_{ja}$. The contribution to $\widetilde{d}(Q_{ia})$ from S is

$$\sum_j \widetilde{d}(S_{ij})\,\tau\,K_{ja} \;=\; \tau \sum_j \widetilde{d}(S_{ij})\,K_{ja}, \quad \text{or in matrix form,} \quad \tau\,\widetilde{d}(S)\,K. \tag{38}$$

Direct contribution through dK. The first-backward formula is $dK_{ja} = \tau \sum_i dS_{ij}\, Q_{ia}$, so the local Jacobian is $\partial dK_{ja}/\partial Q_{ib} = \tau\, dS_{ij}\,\delta_{ab}$. The contribution to $\widetilde{d}(Q_{ib})$ from dK is

$$\sum_{j,a} \widetilde{d}(dK_{ja})\,\tau\,dS_{ij}\,\delta_{ab} \;=\; \tau \sum_j dS_{ij}\,U_{K,jb}, \quad \text{or in matrix form,} \quad \tau\,dS\,U_K. \tag{39}$$

Adding the two contributions,

$$\widetilde{d}(Q) \;=\; \tau\big(\widetilde{d}(S)\,K + dS\,U_K\big). \tag{40}$$

The same calculation with K in place of Q gives

$$\widetilde{d}(K) \;=\; \tau\big(\widetilde{d}(S)^{\mathsf{T}} Q + dS^{\mathsf{T}} U_Q\big). \tag{41}$$

Final assembled identities

Collecting Equations (14), (19), (25), (32), (36), (37), (22), (27), (40), and (41),

$$\begin{aligned}
F &= \tau(U_Q\,K^{\mathsf{T}} + Q\,U_K^{\mathsf{T}}), & \alpha_i &= \textstyle\sum_j P_{ij}\,F_{ij}, & C &= dO\,U_V^{\mathsf{T}},\\
\widetilde{d}(P_{ij})^\circ &= C_{ij} + F_{ij}(dP_{ij} - D_i), & \widetilde{d}(P_{ij}) &= \widetilde{d}(P_{ij})^\circ - \alpha_i\,dP_{ij}, & E_i &= \textstyle\sum_j P_{ij}\,\widetilde{d}(P_{ij}),\\
\widetilde{d}(S) &= P \odot (\widetilde{d}(P) - E\,\mathbf{1}^{\mathsf{T}}), & \widetilde{d}(Q) &= \tau(\widetilde{d}(S)\,K + dS\,U_K), & \widetilde{d}(K) &= \tau(\widetilde{d}(S)^{\mathsf{T}} Q + dS^{\mathsf{T}} U_Q),\\
\widetilde{d}(V) &= (P \odot h)^{\mathsf{T}} dO, & \widetilde{d}(dO) &= P\,U_V + (P \odot h)\,V, & h &= F - \alpha\,\mathbf{1}^{\mathsf{T}}.
\end{aligned}$$

These are the closed-form BoB identities used in Section 4.2.

D Proofs of Proposition 4.1 and Theorem 4.2

Proof of Proposition 4.1. For (8), substitute $\widetilde{d}(P_{ij}) = \widetilde{d}(P_{ij})^\circ - \alpha_i\,dP_{ij}$ into $E_i = \sum_{j\in\mathcal{V}_i} P_{ij}\,\widetilde{d}(P_{ij})$:

$$E_i = \sum_{j\in\mathcal{V}_i} P_{ij}\,\widetilde{d}(P_{ij})^\circ - \alpha_i \sum_{j\in\mathcal{V}_i} P_{ij}\,dP_{ij} = E_i^\circ - \alpha_i D_i.$$

For (8), expand $h_{ij} = F_{ij} - \alpha_i$ in $\widetilde{d}(dO)_i = \sum_{j\in\mathcal{V}_i} P_{ij}\,u_{V,j} + \sum_{j\in\mathcal{V}_i} P_{ij}\,h_{ij}\,v_j$:

$$\widetilde{d}(dO)_i = \underbrace{\sum_{j\in\mathcal{V}_i} P_{ij}\,u_{V,j} + \sum_{j\in\mathcal{V}_i} P_{ij}\,F_{ij}\,v_j}_{\widetilde{d}(dO)_i^\circ} - \alpha_i \underbrace{\sum_{j\in\mathcal{V}_i} P_{ij}\,v_j}_{o_i}.$$

For (8), substitute $\widetilde{d}(P_{ij}) = \widetilde{d}(P_{ij})^\circ - \alpha_i dP_{ij}$ and $dS_{ij} = P_{ij}(dP_{ij} - D_i)$ into $\widetilde{d}(Q)_i = \tau \sum_{j\in\mathcal{V}_i}(P_{ij}(\widetilde{d}(P_{ij}) - E_i)k_j + dS_{ij}u_{K,j})$:

$$\widetilde{d}(Q)_i = \tau\Big(\underbrace{\sum_{j\in\mathcal{V}_i} P_{ij}\,\widetilde{d}(P_{ij})^\circ k_j + \sum_{j\in\mathcal{V}_i} P_{ij}(dP_{ij} - D_i)u_{K,j}}_{\Omega_i} - \alpha_i \underbrace{\sum_{j\in\mathcal{V}_i} P_{ij}\,dP_{ij}\,k_j}_{R_i} - E_i \underbrace{\sum_{j\in\mathcal{V}_i} P_{ij}\,k_j}_{B_i}\Big).$$

□

Proof of Theorem 4.2. By Proposition 4.1, the row-side outputs decompose as $E_i = E_i^\circ - \alpha_i D_i$, $\widetilde{d}(dO)_i = \widetilde{d}(dO)_i^\circ - \alpha_i o_i$, and $\widetilde{d}(Q)_i = \tau(\Omega_i - \alpha_i R_i - E_i B_i)$, where each of α, E°, $\widetilde{d}(dO)^\circ$, Ω, B, and R can be accumulated while streaming the columns without first knowing α_i; here "α-free"

means independent of the completed value of α (Proposition 4.1). The inner loop of Pass 1 is the blockwise realization of these column reductions, visiting each (i, j) tile exactly once per row. D_i is loaded from the saved row state (L, D). Pass 2 loads (α_i, E_i) from HBM and every tilewise intermediate is then the direct blockwise realization of its defining equation from Section 4.2. The column accumulations $\widetilde{d}(V)_j = \sum_i (P_{ij} \odot h_{ij})^\mathsf{T} dO_i$ and $\widetilde{d}(K)_j = \tau \sum_i (\widetilde{d}(S_{ij})^\mathsf{T} Q_i + dS_{ij}^\mathsf{T} U_{Q,i})$ are the exact tile-sum forms of (6) and (7).

For the FLOPs and memory bounds: Pass 1 forms the four $B_r^{(1)} \times B_c^{(1)}$ tiles $dP_{ij}, F_{ij}, C_{ij}, \widetilde{d}(P_{ij})^\circ$ via matrix products of $N \times d$ operands and accumulates six column reductions of total output dimension $O(B_r^{(1)} d)$. Summed over all $\lceil N/B_r^{(1)} \rceil \cdot \lceil N/B_c^{(1)} \rceil$ tile visits, the matrix-multiply work is $\Theta(N^2 d)$ and the elementwise softmax work is $O(N^2)$. Pass 2 has the same per-tile work in column-major orientation, also totalling $\Theta(N^2 d)$. Auxiliary HBM beyond inputs, outputs, the saved row state (L, D), and the forward output O consists only of the two N-vectors (α, E), hence $O(N)$. The on-chip footprint per tile visit is bounded by the tile-budget analysis of Section 4.3. These identities establish algebraic correctness; floating-point reduction order can differ between implementations. □

E Large-cache lower bound

This section proves Theorem 4.5. We use the standard two-level I/O model from Section 3: SRAM has capacity M scalar elements, HBM is unbounded, and I/O counts scalar transfers between them. The lower bound is stated for the standard FlashAttention-style score-recomputation model for exact tiled attention. In this model, the full score matrix $S = \tau Q K^\mathsf{T}$ and probability matrix $P = \mathrm{softmax}(S)$ are not given as HBM inputs. Score blocks are recomputed from the bilinear forms

$$S_{ij} = \tau q_i^\mathsf{T} k_j = \tau \sum_{\ell=1}^{d} Q_{i\ell} K_{j\ell},$$

probability blocks are obtained by exact row-wise softmax normalization, and the algorithm may receive the usual $O(N)$ saved row statistics, such as the row log-normalizer L_i. This is the execution model used by exact FlashAttention-style kernels and by the attention I/O lower bounds invoked below.

The proof has one inherited component and one new component. The inherited component is the large-cache I/O lower bound for exact forward softmax attention under the standard score-recomputation model. In particular, the large-cache lower bound of [48, Lem. 3.4] shows that, for $M = \Omega(d^2)$, exact attention evaluated through the standard matrix-product score computation has I/O complexity $\Omega(N^2 d^2/M)$. This is where the $N^2 d^2/M$ term enters the argument. The new component in this paper is the embedding: we show that a special BoB instance outputs exactly $\mathrm{softmax}(\tau Q K^\mathsf{T}) W$. Therefore, an asymptotically faster exact BoB algorithm in the same score-recomputation model would imply an asymptotically faster exact forward-attention algorithm, contradicting the inherited lower bound.

Intuitively, the factor $N^2 d^2/M$ arises as follows. The attention score computation contains $\Omega(N^2 d)$ scalar multiplication vertices of the form $Q_{i\ell} K_{j\ell}$. In the large-cache regime, [48, Lem. 3.3] bound the number of such level-1 score vertices in each M-partition part by $O(M^2/d)$. Hence any M-partition needs $\Omega(N^2 d^2/M^2)$ parts, and the red-blue pebble partition lemma converts this into $\Omega(N^2 d^2/M)$ HBM transfers.

Lemma E.1 (Forward-attention lower bound with row statistics)**.** *Under the standard FlashAttention-style score-recomputation model, exact forward softmax attention*

$$Y = \mathrm{softmax}(\tau Q K^\mathsf{T}) W$$

with $Q, K, W \in \mathbb{R}^{N \times d}$, $M = \Omega(d^2)$, $N \geq d$, $M \lesssim Nd$, and $\Theta(N^2)$ valid query-key interactions requires $\Omega(N^2 d^2/M)$ HBM transfers.

Proof. We invoke the large-cache score-computation I/O lower bound of [48, Lem. 3.4]. The cited lemma concerns unmasked attention. The causal case contains an unmasked rectangular subproblem: every key in the first half of the sequence is visible to every query in the second half. This gives

$\Theta(N^2)$ score interactions even after masking, so the score-vertex partition argument applies to that subproblem.

The scope of the reduction is important. We require score tiles to be computed through the standard bilinear products of Q and K; full score or probability matrices are neither given as inputs nor stored in HBM. The usual $O(N)$ row normalizers may be supplied, but the execution model still requires score tiles to be recomputed through the stated products. In particular, this is not a lower bound for arbitrary algorithms given an unrestricted, precomputed attention matrix or other nonlinear checkpoints. The inherited partition bound therefore gives $\Omega(N^2d^2/M)$ HBM transfers for $M = \Omega(d^2)$, which dominates the $O(N)$ row-state reads when $M \lesssim Nd$. This establishes the claimed lower bound within the stated score-recomputation model. □

Lemma E.2 (Forward-attention embedding in BoB)**.** *For any $Q, K, W \in \mathbb{R}^{N\times d}$, there is a valid BoB instance whose output $\widetilde{d}(dO)$ is exactly*

$$\widetilde{d}(dO) = \mathrm{softmax}(\tau QK^{\mathsf{T}})W.$$

Proof. Choose

$$V = 0, \qquad dO = 0, \qquad U_Q = 0, \qquad U_K = 0, \qquad U_V = W.$$

The required row state is valid: $O = PV = 0$, $D_i = dO_i^{\mathsf{T}} O_i = 0$, and the row normalizers are determined by Q and K. On this instance,

$$F = \tau(U_Q K^{\mathsf{T}} + QU_K^{\mathsf{T}}) = 0, \qquad \alpha_i = \sum_j P_{ij}F_{ij} = 0, \qquad h_{ij} = F_{ij} - \alpha_i = 0.$$

Using the BoB identity for the upstream-output derivative,

$$\widetilde{d}(dO) = PU_V + (P \odot h)V,$$

we obtain

$$\widetilde{d}(dO) = PW.$$

Thus this BoB instance computes exact forward softmax attention with value matrix W. □

Proof of Theorem 4.5. The argument uses the same score-recomputation model as Lemma E.1: the algorithm is not given S or P as HBM inputs, does not store them as $N \times N$ HBM tensors, and evaluates the score subgraph through the standard bilinear products $S_{ij} = \tau q_i^{\mathsf{T}} k_j$. Suppose, for contradiction, that there were an exact BoB algorithm with

$$o\left(\frac{N^2d^2}{M}\right)$$

HBM transfers under this model. Given an arbitrary forward-attention instance (Q, K, W), construct the BoB instance from Lemma E.2 by setting $V = 0$, $dO = 0$, $U_Q = U_K = 0$, and $U_V = W$. Running the assumed BoB algorithm and reading its $\widetilde{d}(dO)$ output would compute

$$\mathrm{softmax}(\tau QK^{\mathsf{T}})W.$$

This construction only relabels the forward-attention value matrix W as the BoB upstream matrix U_V, while V, dO, U_Q, U_K are fixed zero inputs and the required row state is determined by Q and K. Thus it gives an exact forward-attention algorithm in the same score-recomputation model with asymptotically smaller HBM traffic than Lemma E.1 allows. This contradiction proves that any exact BoB algorithm in the stated score-recomputation model requires

$$\Omega\left(\frac{N^2d^2}{M}\right)$$

HBM transfers. □

Algorithm 2 FLASHBOB: fully annotated schedule.

Input: $Q, K, V, dO, U_Q, U_K, U_V, O \in \mathbb{R}^{N\times d}$; row state $L, D \in \mathbb{R}^N$ in HBM, SRAM size M, $\tau = 1/\sqrt{d}$
Output: $\widetilde{d}(Q), \widetilde{d}(K), \widetilde{d}(V), \widetilde{d}(dO) \in \mathbb{R}^{N\times d}$ and $\alpha, E \in \mathbb{R}^N$

$b \leftarrow \lfloor M/(c_{\text{BoB}} d)\rfloor$; $B_r^{(1)}, B_c^{(2)} \leftarrow b$; $B_c^{(1)}, B_r^{(2)} \leftarrow \min(b, d)$

PASS 1 (ROW-MAJOR). Finalize $(\alpha, E, \widetilde{d}(Q), \widetilde{d}(dO))$ in a single sweep via affine fusion.

for $i = 1$ **to** $\lceil N/B_r^{(1)}\rceil$ **do**
  Load $Q_i, dO_i, U_{Q,i}, O_i \in \mathbb{R}^{B_r^{(1)}\times d}$ and saved row scalars $L_i, D_i \in \mathbb{R}^{B_r^{(1)}}$ ▷ active row block
  $\alpha_i, E_i^\circ \leftarrow 0 \in \mathbb{R}^{B_r^{(1)}}$; $\widetilde{d}(dO)_i^\circ, \Omega_i, B_i, R_i \leftarrow 0 \in \mathbb{R}^{B_r^{(1)}\times d}$ ▷ six row accumulators; five are α-free
  for $j = 1$ **to** $\lceil N/B_c^{(1)}\rceil$ **do**
    Load $K_j, V_j, U_{K,j}, U_{V,j} \in \mathbb{R}^{B_c^{(1)}\times d}$ ▷ streamed column block
    Recompute $S_{ij}, P_{ij} \in \mathbb{R}^{B_r^{(1)}\times B_c^{(1)}}$ from (Q_i, K_j, L_i) ▷ tilewise softmax
    $dP_{ij} \leftarrow dO_i\, V_j^\mathsf{T}$ ▷ upstream to P via dV path
    $F_{ij} \leftarrow \tau(U_{Q,i} K_j^\mathsf{T} + Q_i U_{K,j}^\mathsf{T})$ ▷ BoB gradient at dS, eq. (14)
    $C_{ij} \leftarrow dO_i\, U_{V,j}^\mathsf{T}$ ▷ contribution from dV path
    $\widetilde{d}(P_{ij})^\circ \leftarrow C_{ij} + F_{ij} \odot (dP_{ij} - D_i)$ ▷ α-free base of $\widetilde{d}(P)$, eq. (32)
    $\alpha_i \mathrel{+}= \text{rowsum}(P_{ij} \odot F_{ij})$ ▷ row scalar for affine update
    $E_i^\circ \mathrel{+}= \text{rowsum}(P_{ij} \odot \widetilde{d}(P_{ij})^\circ)$ ▷ α-free component of E_i
    $\widetilde{d}(dO)_i^\circ \mathrel{+}= P_{ij} U_{V,j} + (P_{ij} \odot F_{ij}) V_j$ ▷ α-free component of $\widetilde{d}(dO)_i$
    $\Omega_i \mathrel{+}= (P_{ij} \odot \widetilde{d}(P_{ij})^\circ) K_j + (P_{ij} \odot (dP_{ij} - D_i)) U_{K,j}$ ▷ α-free component of $\widetilde{d}(Q)_i$
    $B_i \mathrel{+}= P_{ij} K_j$; $R_i \mathrel{+}= (P_{ij} \odot dP_{ij}) K_j$ ▷ coefficients for update
  end for
  $E_i \leftarrow E_i^\circ - \alpha_i \odot D_i$ ▷ eq. (8)
  $\widetilde{d}(dO)_i \leftarrow \widetilde{d}(dO)_i^\circ - \alpha_i \odot O_i$ ▷ eq. (8)
  $\widetilde{d}(Q)_i \leftarrow \tau(\Omega_i - \alpha_i \odot R_i - E_i \odot B_i)$ ▷ eq. (8)
  Write $\alpha_i, E_i, \widetilde{d}(Q)_i, \widetilde{d}(dO)_i$ to HBM ▷ (α, E) consumed in Pass 2
end for

PASS 2 (COLUMN-MAJOR). Consume saved (α, E); accumulate and write $(\widetilde{d}(K), \widetilde{d}(V))$.

for $j = 1$ **to** $\lceil N/B_c^{(2)}\rceil$ **do**
  Load $K_j, V_j, U_{K,j}, U_{V,j} \in \mathbb{R}^{B_c^{(2)}\times d}$ ▷ active column block
  Initialize $\widetilde{d}(K)_j, \widetilde{d}(V)_j \leftarrow 0 \in \mathbb{R}^{B_c^{(2)}\times d}$
  for $i = 1$ **to** $\lceil N/B_r^{(2)}\rceil$ **do**
    Load $Q_i, dO_i, U_{Q,i} \in \mathbb{R}^{B_r^{(2)}\times d}$ and $L_i, D_i, \alpha_i, E_i \in \mathbb{R}^{B_r^{(2)}}$ ▷ streamed row block + saved scalars
    Recompute $S_{ij}, P_{ij} \in \mathbb{R}^{B_r^{(2)}\times B_c^{(2)}}$ from (Q_i, K_j, L_i)
    $dP_{ij} \leftarrow dO_i\, V_j^\mathsf{T}$; $F_{ij} \leftarrow \tau(U_{Q,i} K_j^\mathsf{T} + Q_i U_{K,j}^\mathsf{T})$; $C_{ij} \leftarrow dO_i\, U_{V,j}^\mathsf{T}$
    $h_{ij} \leftarrow F_{ij} - \alpha_i$; $dS_{ij} \leftarrow P_{ij} \odot (dP_{ij} - D_i)$ ▷ tile-wise quantities
    $\widetilde{d}(P_{ij}) \leftarrow C_{ij} + F_{ij} \odot (dP_{ij} - D_i) - \alpha_i \odot dP_{ij}$ ▷ full $\widetilde{d}(P)$, eq. (32)
    $\widetilde{d}(S_{ij}) \leftarrow P_{ij} \odot (\widetilde{d}(P_{ij}) - E_i)$ ▷ second softmax-backward
    $\widetilde{d}(V)_j \mathrel{+}= (P_{ij} \odot h_{ij})^\mathsf{T} dO_i$ ▷ column update for $\widetilde{d}(V)_j$
    $\widetilde{d}(K)_j \mathrel{+}= \tau(\widetilde{d}(S_{ij})^\mathsf{T} Q_i + dS_{ij}^\mathsf{T} U_{Q,i})$ ▷ column update for $\widetilde{d}(K)_j$
  end for
  Write $\widetilde{d}(K)_j, \widetilde{d}(V)_j$ to HBM
end for

F FlashBoB: Fully Annotated Pseudocode

Algorithm 2 reproduces the schedule of Algorithm 1 with explicit tile shapes and inline commentary on the role of each quantity. Shapes are given for the divisible dense case; boundary tiles are padded and masked as in FlashAttention. For causal attention, masked tile pairs are skipped and invalid elements within boundary tiles have zero probability. All intermediates P_{ij}, dP_{ij}, F_{ij}, C_{ij}, $\widetilde{d}(P)^\circ_{ij}$, $\widetilde{d}(P)_{ij}$, $\widetilde{d}(S)_{ij}$ are formed only as $B_r \times B_c$ tiles in SRAM/registers and are discarded after the tile update; no full $N \times N$ intermediate is materialized in HBM.

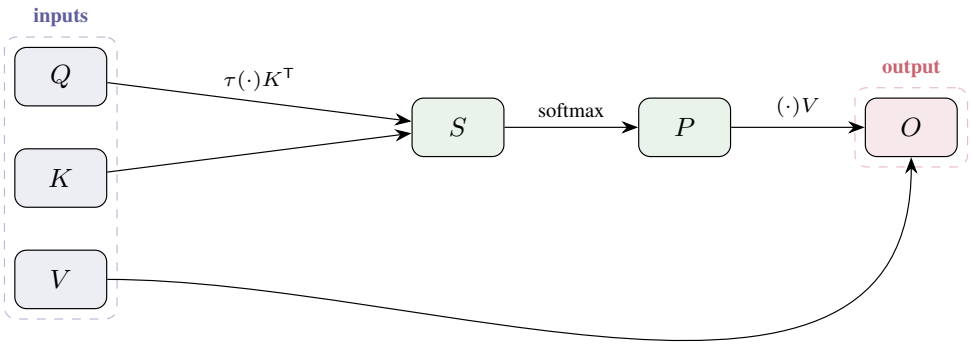


Figure 3: Forward dependency graph. Three inputs flow through two $N \times N$ intermediates (S and P) to a single $N \times d$ output.

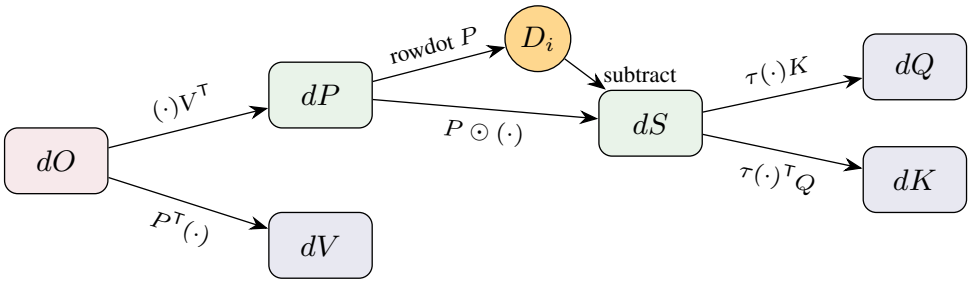


Figure 4: First-backward dependency graph. One upstream (dO) produces three outputs via two $N \times N$ intermediates, with a single row-scalar detour through D_i.

G Dependency graphs and schedule overlays

This appendix collects the dependency graphs underlying the derivation. Three simple graph ideas appear repeatedly. A *fan-out* means one quantity is computed once and then sent to several later expressions. A *reduction detour* means an $N \times N$ quantity is first compressed to one scalar per row and that row scalar is then used later. A *two-factor product* means a node is formed by multiplying information coming from two streamed operands. Figures 3 and 4 show these patterns in the forward and first-backward graphs, and Figure 5 shows how they reappear in BoB. Figures 6 and 7 then overlay the conservative three-pass and affine-fused two-pass schedules on the BoB graph.

In all graphs, blue nodes are inputs, green nodes are intermediates, red nodes are outputs or incoming gradients from the next derivative level, and yellow circles are one-scalar-per-row quantities. Dashed gray nodes are recomputed tilewise quantities that are never materialized in HBM.

G.1 Three-pass versus two-pass: HBM savings and empirical comparison

Throughout this appendix, the *FlashAttention execution model* means an exact tiled attention execution with bounded $O(M)$ on-chip working sets, no global atomic adds on output tensors, no $N \times N$ HBM intermediates, and probability tiles recomputed from saved row state (L, D) rather than read from HBM.

The four BoB outputs do not finish in the same traversal direction (§4.2), forcing one row-major and one column-major sweep. A single pass cannot satisfy the FlashAttention execution model: the column-side outputs $(\widetilde{d}(K), \widetilde{d}(V))$ reduce over rows for fixed column, so accumulating them during a row-major traversal would require materializing an $N \times N$ intermediate in HBM, using global

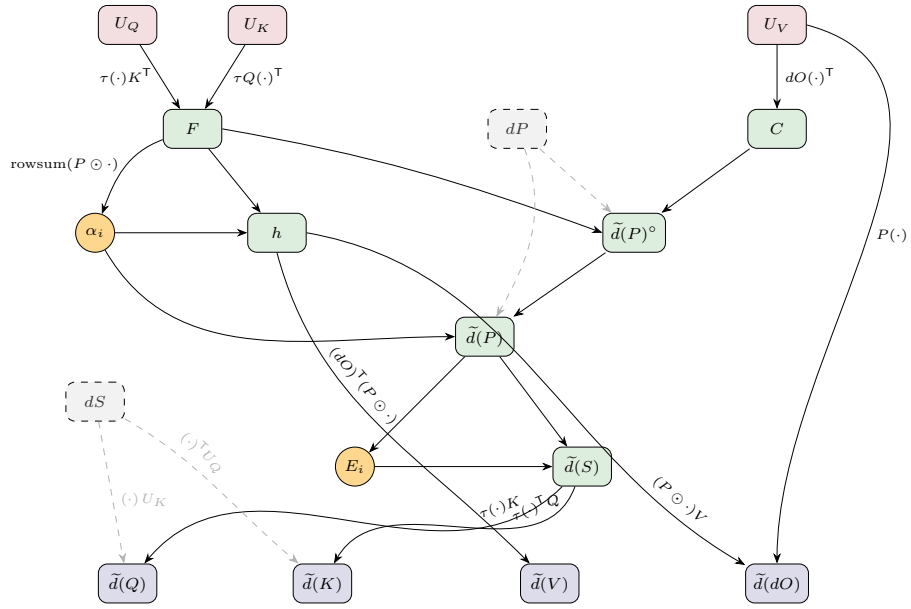


Figure 5: BoB dependency graph. Three upstream matrices produce four outputs through five $N \times N$ intermediates and two row scalars. The softmax backward identity appears twice (at $\alpha_i \to h$ and at $E_i \to \widetilde{d}(S)$).

Table 8: Direct comparison between the three-pass baseline schedule and the affine-fused two-pass kernel on NVIDIA A100 (GPT-2 Small shape, batch 4, BF16 inputs with FP32 reductions).

N	3-PASS BOB (ms)	FLASHBOB (ms)	Ratio
1 024	2.70	2.57	1.05×
4 096	27.30	25.80	1.06×
16 384	389.14	365.73	1.06×
65 536	6 227.59	5 843.14	1.07×
262 144	100 696.16	94 853.72	1.06×

atomics for column-indexed outputs, or holding $\Theta(Nd)$ partial column accumulators on chip. Each path gives up the discipline that made FlashAttention fast.

The three-pass schedule (Figure 6) is itself the result of one level of affine fusion: its second pass uses the delayed E-correction to finalize $\widetilde{d}(Q)$ in a single row-major sweep. The two-pass FLASHBOB schedule (Figure 7) extends this with a second level of affine fusion that eliminates the dedicated α-only sweep (Pass 1 of the three-pass schedule). The HBM saving is bounded by the streamed reads of the eliminated sweep, which loads exactly two $B \times d$ tensors per tile pair (K_j and $U_{K,j}$, the operands of F_{ij}). This is $2dN^2/B$ out of the three-pass total of $(10d + 4)N^2/B$ streamed traffic, about $2d/(10d + 4) \approx 19\%$ of the streamed budget at d=64. With compute unchanged across schedules, the wall-time advantage scales the streamed-traffic ratio by the memory-bound fraction of the kernel.

The measured 5–7% advantage at long context is consistent with the predicted streamed-traffic ratio. We view this as additional evidence for the schedule choice; the formal optimality argument is the large-cache I/O bound of Theorem 4.5.

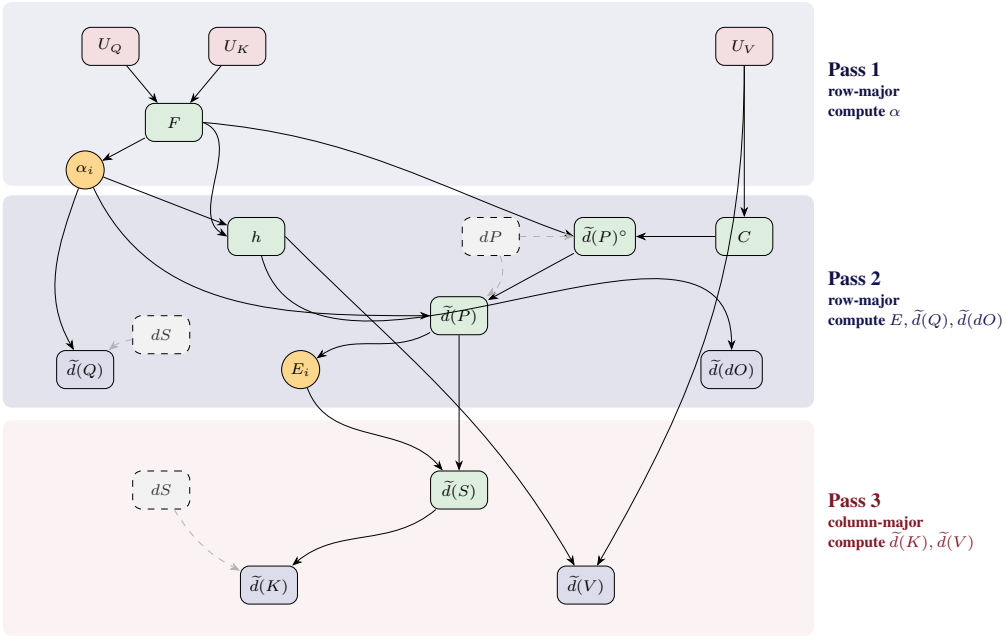


Figure 6: Three-pass schedule overlaid on the BoB graph. Pass 1 exists only to accumulate α_i. Proposition 4.1 collapses Passes 1 and 2 into one.

H Per-pass I/O ledger and proof of Theorem 4.3

Pass	Stationary loads	Streamed loads	Writes
1 (row)	$Q_i, U_{Q,i}, do_i, O_i, L_i, D_i$	$K_j, U_{K,j}, V_j, U_{V,j}$	$\widetilde{d}(Q)_i, \widetilde{d}(dO)_i, \alpha_i, E_i$
2 (col)	$K_j, V_j, U_{K,j}, U_{V,j}$	$Q_i, U_{Q,i}, do_i, L_i, D_i, \alpha_i, E_i$	$\widetilde{d}(K)_j, \widetilde{d}(V)_j$

Pass 1 element counts. Stationary reads: $\frac{N}{B_r^{(1)}}(4B_r^{(1)}d + 2B_r^{(1)}) = 4Nd + 2N$. Streamed reads: $\frac{N}{B_r^{(1)}}\frac{N}{B_c^{(1)}} \cdot 4B_c^{(1)}d = \frac{4dN^2}{B_r^{(1)}}$. Writes: $\frac{N}{B_r^{(1)}}(2B_r^{(1)}d + 2B_r^{(1)}) = 2Nd + 2N$.

Pass 2 element counts. Stationary reads: $\frac{N}{B_c^{(2)}} \cdot 4B_c^{(2)}d = 4Nd$. Streamed reads: $\frac{N}{B_c^{(2)}}\frac{N}{B_r^{(2)}}(3B_r^{(2)}d + 4B_r^{(2)}) = \frac{(3d+4)N^2}{B_c^{(2)}}$. Writes: $\frac{N}{B_c^{(2)}} \cdot 2B_c^{(2)}d = 2Nd$.

Summing all reads and writes,

$$\begin{aligned} Q_{\text{FlashBoB}} &= (4Nd + 2N) + \frac{4dN^2}{B_r^{(1)}} + (2Nd + 2N) + 4Nd + \frac{(3d+4)N^2}{B_c^{(2)}} + 2Nd \\ &= 12Nd + 4N + \frac{4dN^2}{B_r^{(1)}} + \frac{(3d+4)N^2}{B_c^{(2)}}, \end{aligned}$$

proving Theorem 4.3. Setting $B_r^{(1)} = B_c^{(2)} = b$ gives $Q_{\text{FlashBoB}} = 12Nd + 4N + (7d+4)N^2/b$. □

The constant c_{BoB} in $b = \lfloor M/(c_{\text{BoB}}d) \rfloor$ is fixed by the on-chip footprint of one tile visit. The footprint contains a constant number of row-vector blocks, column-vector blocks, interaction-shaped $B_r \times B_c$ tiles, and row/column scalar buffers. We can write the requirement as

$$M \;\geq\; c_1 B_r d + c_2 B_c d + c_3 B_r B_c + c_4(B_r + B_c),$$

for implementation-dependent constants $c_1, c_2, c_3, c_4 > 0$. The $c_3 B_r B_c$ term accounts for the constant number of interaction-shaped intermediates formed during a tile visit, such as probability-like, score-adjoint-like, and temporary product tiles. This constant-factor accounting does not change

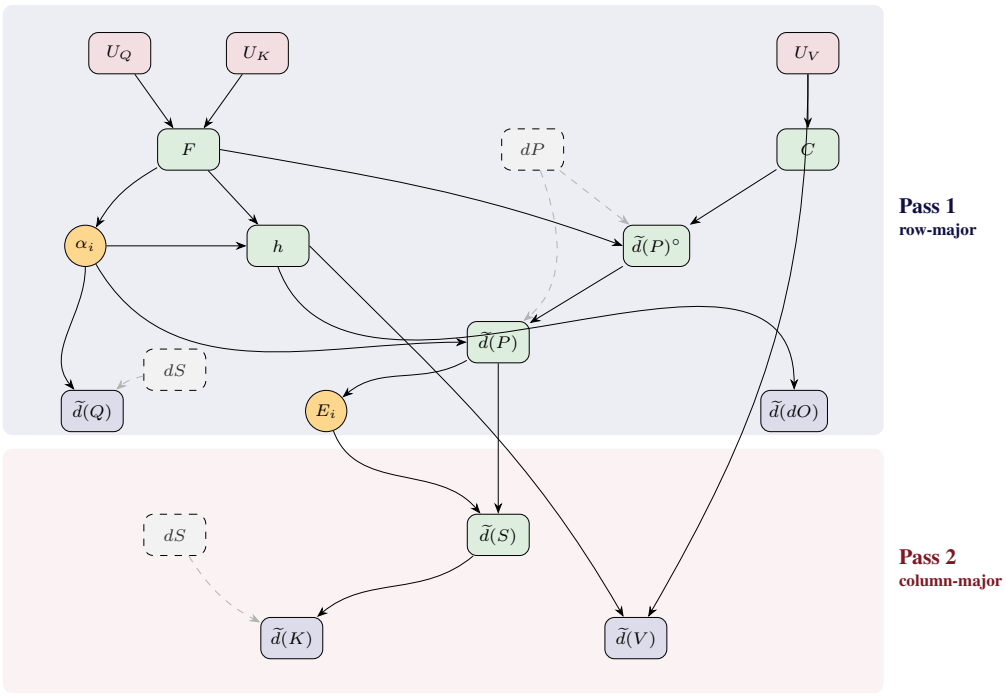


Figure 7: Affine-fused two-pass schedule (FLASHBOB) overlaid on the BoB graph. One horizontal cut separates row-side nodes (above) from column-side nodes (below). Pass 1 finalizes $\widetilde{d}(Q)$ and $\widetilde{d}(dO)$ by accumulating the six row accumulators from Proposition 4.1 and applying scalar corrections at the end of the sweep; Pass 2 consumes the saved (α, E) and finalizes $\widetilde{d}(K)$ and $\widetilde{d}(V)$ as natural column reductions.

the asymptotic choice $B_r, B_c = \Theta(M/d)$ in the large-cache regime used by Theorem 4.3; it only makes explicit that the SRAM budget hides more than one interaction tile.

Under causal masking, each pass visits only tile pairs that intersect the lower-triangular attention pattern. When row and column tiles have the same size and $T_r = T_c = T$, there are $T(T+1)/2$ visited pairs instead of T^2; the resulting fraction is $\rho = (1 + 1/T)/2$. For the unequal row and column tile sizes used by Algorithm 1, the boundary-tile count depends on both tile counts, but the leading quadratic streamed traffic approaches one half of the dense count at fixed tile sizes as N grows. The stationary and output-write terms of Theorem 4.3 are unchanged.

I Extensions

The derivation of Section 4 extends cleanly to three settings that appear in practice: variable-length packed sequences, dropout (derived analytically below; the current code release does not enable dropout or arbitrary user-provided attention bias), and mixed precision.

Variable-length packing and sparse masks require only that each row has a well-defined set of allowed columns and that masked entries have zero probability. For packed variable-length attention with total length N, define $\mathcal{V}$ as the set of block pairs with at least one unmasked entry and restrict the inner streamed loop over column blocks to those j with $(i, j) \in \mathcal{V}$. Under a causal mask, $|\mathcal{V}|$ is block-triangular, so roughly half of all tile visits are skipped. For each pass p, let $\mathcal{T}^{(p)}$ denote the visited tile pairs under its row/column tiling. The dense tile-pair count $T_r^{(p)} T_c^{(p)}$ is replaced by $|\mathcal{T}^{(p)}|$; traffic also depends on the streamed tile widths and boundary tiles.

Analytical dropout extension. For dropout after the softmax, let $R_{ij} \in \{0, 1\}$ be a fixed keep mask and $g_{ij} = R_{ij}/(1 - p_{\text{drop}})$. The forward probability used for values is $\widehat{P} = g \odot P$, so $O = \widehat{P}V$ and $dV = \widehat{P}^\mathsf{T} dO$. The upstream of the *undropped* softmax is $A = g \odot (dOV^\mathsf{T})$; hence the first-backward

row statistic is $D_i = \sum_j P_{ij} A_{ij} = dO_i^\mathsf{T} o_i$. The BoB row summaries and intermediate probabilities then satisfy

$$\alpha_i = \sum_j P_{ij} F_{ij}, \qquad C = g \odot (dOU_V^\mathsf{T}),$$
$$\widetilde{d}(P_{ij}) = C_{ij} + F_{ij}(A_{ij} - D_i) - \alpha_i A_{ij}, \qquad E_i = \sum_j P_{ij} \widetilde{d}(P_{ij}),$$
$$\widetilde{d}(S_{ij}) = P_{ij}(\widetilde{d}(P_{ij}) - E_i), \qquad h_{ij} = F_{ij} - \alpha_i.$$

The column- and row-owned outputs involving the value pathway become

$$\widetilde{d}(V) = (g \odot P \odot h)^\mathsf{T} dO,$$
$$\widetilde{d}(dO) = (g \odot P) U_V + (g \odot P \odot h) V.$$

Thus the outer softmax Jacobian still uses the undropped P, while paths through the dropout operation are multiplied by g. These are analytical identities. The released implementation does not support dropout; an implementation would need to regenerate the same mask in both passes, for example using a counter-based random-number generator.

The element counts in Theorem 4.3 are precision-agnostic, so mixed-precision execution requires no algorithmic change. A standard implementation stores $Q, K, V, dO, U_Q, U_K, U_V$ in BF16 or FP16 and accumulates all reductions (L, D, α, E, E°) in FP32, matching the FlashAttention backward [10].

J Additional numerical agreement data

We evaluate the maximum absolute and relative discrepancy of each BoB output against the PyTorch MATH autograd reference across sequence lengths $N \in \{128, 256, 512, 1024, 2048, 4096, 8192\}$. The MATH reference differentiates PyTorch's SDPA `math` backend twice via `torch.autograd.grad` with $N \times N$ intermediates materialized in HBM and appears in Table 9 with zero discrepancy as the baseline. We also verified that evaluating the closed-form identities of §4.2 directly in PyTorch matches autograd double backward to floating-point precision. We compare four non-materializing implementations against this reference: the two-pass FLASHBOB schedule (Algorithm 1), the intermediate three-pass schedule (3-PASS BOB; Appendix G.1), and the two GradMem-derived HVP kernels (HVP-M and HVP-S). All runs use BF16 inputs with FP32 accumulators on a single NVIDIA A6000 in the GPT-2 Small attention configuration (batch 4, 12 heads, $d = 64$, causal mask). We also verified numerical agreement for $d = 128$ and native GQA shapes ($H_Q > H_{KV}$); Table 9 reports the $d = 64$ sweep.

Across the entries in Table 9, the largest absolute discrepancy is 2.38×10^{-5} and the largest reported relative discrepancy is 4.76×10^{-6}. The differences are consistent with different floating-point reduction orders. These values describe numerical agreement with the autograd oracle; they do not imply bitwise agreement.

K Extended isolated-kernel scaling

Table 10 reports the full isolated attention-layer BoB scaling sweep at $d = 64$ on NVIDIA A100 80GB, corresponding to Table 1 (left panel). Once the materializing MATH backend ($N = 8{,}192$) and the GradMem-derived HVP baselines ($N = 16{,}384$) exhaust memory, FLASHBOB continues to execute exact attention BoB at $N = 262{,}144$ on a single A100 80GB GPU. This is an isolated attention-layer capacity measurement; it does not establish that a complete model can train at that sequence length.

L Full model benchmark sweeps: GPT-2, Pythia, and native GQA/MoE

This appendix reports per-N sweeps of complete model-level second-order steps. They measure the model step, including non-attention operations, rather than the attention-BoB kernel in isolation. The sweeps support Table 3 in §5.2: GPT-2 Small, Medium, and Large on NVIDIA B200 (Tables 11–13),

Table 9: Maximum absolute and relative discrepancy of each BoB output against the PyTorch MATH autograd double-backward reference, for $N \in \{128, \ldots, 8192\}$. Math reference shown for completeness as the all-zero baseline. In addition to the main methods from Table 1, we include a 3-pass BoB implementation as an auxiliary correctness check. Hardware: NVIDIA A6000 48GB; precision: BF16 inputs with FP32 reductions.

		Max absolute discrepancy				Max relative discrepancy			
N	Backend	$\widetilde{d}(Q)$	$\widetilde{d}(K)$	$\widetilde{d}(V)$	$\widetilde{d}(dO)$	$\widetilde{d}(Q)$	$\widetilde{d}(K)$	$\widetilde{d}(V)$	$\widetilde{d}(dO)$
128	MATH	0	0	0	0	0	0	0	0
	HVP-M	2.38e-6	5.25e-6	1.43e-6	9.54e-7	5.74e-7	8.08e-7	3.31e-7	1.97e-7
	HVP-S	2.38e-6	5.25e-6	1.43e-6	9.54e-7	5.74e-7	8.08e-7	3.31e-7	1.97e-7
	FLASHBOB	3.34e-6	3.58e-6	1.91e-6	2.38e-6	8.03e-7	5.51e-7	4.42e-7	4.92e-7
	3-PASS BOB	2.62e-6	5.01e-6	1.91e-6	1.67e-6	6.31e-7	7.72e-7	4.42e-7	3.44e-7
256	MATH	0	0	0	0	0	0	0	0
	HVP-M	1.31e-6	3.93e-6	1.43e-6	1.19e-6	3.74e-7	1.02e-6	3.84e-7	2.96e-7
	HVP-S	1.31e-6	3.93e-6	1.43e-6	1.19e-6	3.74e-7	1.02e-6	3.84e-7	2.96e-7
	FLASHBOB	2.15e-6	4.53e-6	1.43e-6	1.91e-6	6.12e-7	1.18e-6	3.84e-7	4.74e-7
	3-PASS BOB	2.15e-6	3.58e-6	1.43e-6	1.19e-6	6.12e-7	9.31e-7	3.84e-7	2.96e-7
512	MATH	0	0	0	0	0	0	0	0
	HVP-M	1.31e-6	6.20e-6	1.43e-6	7.15e-7	3.51e-7	1.21e-6	3.80e-7	1.78e-7
	HVP-S	1.31e-6	6.20e-6	1.43e-6	7.15e-7	3.51e-7	1.21e-6	3.80e-7	1.78e-7
	FLASHBOB	2.86e-6	5.96e-6	1.31e-6	1.91e-6	7.66e-7	1.17e-6	3.49e-7	4.74e-7
	3-PASS BOB	1.91e-6	5.96e-6	1.55e-6	1.37e-6	5.11e-7	1.17e-6	4.12e-7	3.40e-7
1 024	MATH	0	0	0	0	0	0	0	0
	HVP-M	1.79e-6	1.12e-5	1.91e-6	7.45e-7	3.57e-7	2.44e-6	5.25e-7	2.18e-7
	HVP-S	1.79e-6	1.12e-5	1.91e-6	7.45e-7	3.57e-7	2.44e-6	5.25e-7	2.18e-7
	FLASHBOB	2.38e-6	6.20e-6	1.67e-6	1.85e-6	4.75e-7	1.35e-6	4.60e-7	5.40e-7
	3-PASS BOB	1.91e-6	5.72e-6	1.67e-6	1.07e-6	3.80e-7	1.25e-6	4.60e-7	3.14e-7
2 048	MATH	0	0	0	0	0	0	0	0
	HVP-M	2.15e-6	1.34e-5	2.15e-6	1.19e-6	5.29e-7	2.87e-6	5.74e-7	2.94e-7
	HVP-S	2.15e-6	1.34e-5	2.15e-6	1.19e-6	5.29e-7	2.87e-6	5.74e-7	2.94e-7
	FLASHBOB	3.34e-6	1.12e-5	1.91e-6	2.15e-6	8.23e-7	2.41e-6	5.10e-7	5.29e-7
	3-PASS BOB	3.70e-6	1.03e-5	1.91e-6	1.43e-6	9.11e-7	2.20e-6	5.10e-7	3.52e-7
4 096	MATH	0	0	0	0	0	0	0	0
	HVP-M	2.06e-6	1.72e-5	1.67e-6	7.15e-7	3.54e-7	3.29e-6	3.06e-7	1.47e-7
	HVP-S	2.06e-6	1.72e-5	1.67e-6	7.15e-7	3.54e-7	3.29e-6	3.06e-7	1.47e-7
	FLASHBOB	3.34e-6	1.26e-5	2.15e-6	1.97e-6	5.75e-7	2.42e-6	3.94e-7	4.04e-7
	3-PASS BOB	2.15e-6	1.00e-5	2.15e-6	1.31e-6	3.69e-7	1.92e-6	3.94e-7	2.69e-7
8 192	MATH	0	0	0	0	0	0	0	0
	HVP-M	2.33e-6	2.38e-5	1.67e-6	1.07e-6	6.33e-7	4.76e-6	4.12e-7	2.79e-7
	HVP-S	2.33e-6	2.38e-5	1.67e-6	1.07e-6	6.33e-7	4.76e-6	4.12e-7	2.79e-7
	FLASHBOB	4.29e-6	2.11e-5	2.15e-6	2.41e-6	1.17e-6	4.22e-6	5.30e-7	6.28e-7
	3-PASS BOB	2.74e-6	2.00e-5	1.91e-6	1.43e-6	7.47e-7	4.00e-6	4.71e-7	3.72e-7

Pythia-160M and Pythia-410M on NVIDIA B300 (Table 14), and five released native GQA and MoE models on NVIDIA B300 (Table 15).

Table 12: GPT-2 Medium full sweep, batch 4, BF16 inputs with FP32 reductions. Hardware: NVIDIA B200 (192 GiB HBM); OOM denotes exceeding the 192 GiB budget.

	Wall time (ms)				Peak memory (MiB)			
N	MATH	HVP-M	HVP-S	FLASHBOB	MATH	HVP-M	HVP-S	FLASHBOB
512	88.4	102.8	104.0	81.9	10 443	7 851	7 900	7 900
1 024	125.6	125.2	120.9	98.8	20 562	10 769	10 869	10 869
2 048	312.6	343.0	255.9	177.2	55 203	16 612	16 810	16 810
4 096	OOM	1028.5	726.1	420.9	–	30 480	30 876	28 677
8 192	OOM	3944.0	2557.1	1196.6	–	74 523	75 315	52 429
16 384	OOM	OOM	OOM	3878.9	–	–	–	99 912

Table 10: Extended attention-layer BoB scaling on NVIDIA A100 80GB. Shape: GPT-2 Small attention, batch 4, 12 heads, head dimension 64, causal mask, BF16 inputs with FP32 reductions. OOM denotes out of memory; best wall-clock time and lowest peak memory are bold.

	Wall-clock time (ms)				Peak memory (MiB)			
N	MATH	HVP-M	HVP-S	FLASHBOB	MATH	HVP-M	HVP-S	FLASHBOB
256	1.82	1.58	1.66	**0.94**	212	114	116	**95**
512	4.95	2.31	2.33	**1.29**	668	315	318	**171**
1 024	17.66	7.39	6.41	**2.57**	2 375	1 045	1 051	**323**
2 048	68.80	28.34	22.73	**7.53**	8 957	3 805	3 817	**630**
4 096	265.67	114.89	89.14	**25.80**	34 794	14 521	14 546	**1 243**
8 192	OOM	468.93	356.84	**95.11**	–	55 971	56 020	**1 702**
16 384	OOM	OOM	OOM	**365.73**	–	–	–	**3 372**
32 768	OOM	OOM	OOM	**1 449.06**	–	–	–	**6 744**
65 536	OOM	OOM	OOM	**5 843.14**	–	–	–	**13 488**
131 072	OOM	OOM	OOM	**23 528.87**	–	–	–	**26 976**
262 144	OOM	OOM	OOM	**94 853.72**	–	–	–	**53 952**

Table 11: GPT-2 Small full sweep, batch 4, BF16 inputs with FP32 reductions. Hardware: NVIDIA B200 (192 GiB HBM); OOM denotes exceeding the 192 GiB budget.

	Wall time (ms)				Peak memory (MiB)			
N	MATH	HVP-M	HVP-S	FLASHBOB	MATH	HVP-M	HVP-S	FLASHBOB
512	46.5	48.0	49.5	43.0	4 932	3 956	3 981	3 983
1 024	58.7	58.6	57.5	48.6	9 605	5 913	5 956	5 957
2 048	134.9	146.3	114.6	84.4	24 091	9 830	9 915	9 924
4 096	397.7	417.5	304.0	190.9	79 058	17 649	17 804	17 804
8 192	OOM	1537.5	1017.3	509.1	–	46 965	47 267	33 604
16 384	OOM	4974.7	3421.6	1576.4	–	147 483	148 082	65 194
32 768	OOM	OOM	OOM	5368.3	–	–	–	128 392

Table 13: GPT-2 Large full sweep, batch 4, BF16 inputs with FP32 reductions. Hardware: NVIDIA B200 (192 GiB HBM); OOM denotes exceeding the 192 GiB budget.

	Wall time (ms)				Peak memory (MiB)			
N	MATH	HVP-M	HVP-S	FLASHBOB	MATH	HVP-M	HVP-S	FLASHBOB
512	131.3	135.4	139.3	122.3	19 720	14 856	14 956	14 965
1 024	215.4	216.6	192.7	154.4	37 496	19 120	19 335	19 342
2 048	562.2	614.6	450.4	304.3	100 554	27 666	28 040	28 045
4 096	OOM	1879.6	1313.4	740.3	–	49 205	49 968	45 457
8 192	OOM	7322.2	4721.8	2162.8	–	110 927	112 414	80 303
16 384	OOM	OOM	OOM	7106.5	–	–	–	150 019

Table 14: End-to-end second-order steps on Pythia (GPT-NeoX architecture) at $d = 128$. Batch 4, BF16 with FP32 reductions, NVIDIA B300. Speedup is relative to the fastest baseline that fits at the same sequence length. N restarts for each model block.

Model	N	MATH (ms)	HVP-M (ms)	HVP-S (ms)	**FLASHBOB (ms)**	FLASHBOB memory (MiB)	Speedup
Pythia-160M	512	87.9	91.0	101.8	**79.1**	5 062	1.11×
	1 024	103.2	106.0	106.2	**99.3**	8 942	1.04×
	2 048	152.4	161.4	142.7	**115.7**	16 702	1.23×
	4 096	420.8	437.0	361.7	**239.8**	32 222	1.51×
	8 192	OOM	1 563.1	1 195.0	**659.0**	63 263	1.81×
	16 384	OOM	5 069.5	4 088.9	**2 064.2**	126 273	1.98×
	32 768	OOM	OOM	OOM	**7 114.6**	250 435	–
Pythia-410M	512	93.7	91.5	92.2	**81.7**	8 763	1.12×
	1 024	161.1	158.5	148.4	**126.3**	14 502	1.17×
	2 048	368.0	393.8	328.7	**242.9**	25 992	1.35×
	4 096	1 074.1	1 125.0	918.4	**583.7**	48 974	1.57×
	8 192	OOM	4 121.5	3 121.7	**1 652.8**	94 937	1.89×
	16 384	OOM	OOM	OOM	**5 280.2**	186 863	–

Table 16: GPT-2 Small with sliding-window attention, $w = N/4$. Hardware: NVIDIA A100 80GB. Wall time reports the median $\pm$ half of the 20th-to-80th percentile spread. OOM denotes out of memory.

		Wall time (ms)			Peak memory (MiB)		
N	w	MATH	FLASHBOB	Speedup	MATH	FLASHBOB	Mem. ratio
2,048	512	175.376 ± 0.111	136.254 ± 0.125	$1.29\times$	8,366.84	5,007.09	$1.67\times$
4,096	1,024	432.310 ± 0.104	218.646 ± 0.113	$1.98\times$	21,809.62	9,019.05	$2.42\times$
8,192	2,048	$1{,}330.003 \pm 0.089$	457.850 ± 0.061	$2.90\times$	66,801.47	17,040.55	$3.92\times$
16,384	4,096	OOM	$1{,}136.572 \pm 0.172$	–	OOM	33,104.94	–

Table 15: Complete model-level second-order calculations on native GQA and MoE models. Batch size 1, BF16, one NVIDIA B300. Latencies are synchronized CUDA-event medians. Peak memory is PyTorch `max_memory_allocated`, measured identically for both methods.

Model (H_Q/H_{KV})	d	N	**FLASHBOB (ms)**	MATH (ms)	Speedup	**FLASHBOB (GiB)**	MATH (GiB)	Reduction
Llama 3.2 1B (32/8)	64	1 024	**68.681**	96.664	1.41×	**16.489**	22.987	28.3%
	64	2 048	**116.920**	220.275	1.88×	**23.755**	51.946	54.3%
	64	4 096	**247.073**	653.269	2.64×	**39.761**	152.469	73.9%
SmolLM2-135M (9/3)	64	1 024	**110.231**	113.387	1.03×	**4.211**	7.753	45.7%
	64	2 048	**105.555**	146.113	1.38×	**7.601**	21.576	64.8%
	64	4 096	**147.941**	360.238	2.44×	**14.381**	69.893	79.4%
	64	8 192	**319.350**	1 208.060	3.78×	**27.942**	249.214	88.8%
SmolLM2-360M (15/5)	64	1 024	**100.320**	116.980	1.17×	**7.712**	13.981	44.8%
	64	2 048	**122.291**	213.072	1.74×	**13.298**	38.021	65.0%
	64	4 096	**212.327**	599.298	2.82×	**24.470**	122.664	80.1%
Qwen2.5-1.5B (12/2)	128	1 024	**111.551**	116.835	1.05×	**21.780**	25.944	16.0%
	128	2 048	**176.361**	224.462	1.27×	**34.482**	51.476	33.0%
	128	4 096	**365.433**	568.491	1.56×	**59.886**	128.555	53.4%
Granite 3.1 1B-A400M (16/8, MoE)	64	1 024	**113.701**	125.384	1.10×	**15.517**	20.222	23.3%
	64	2 048	**151.667**	229.960	1.52×	**23.167**	42.935	46.0%
	64	4 096	**272.416**	593.166	2.18×	**38.729**	118.548	67.3%

Power-of-two capacity sweeps extend the longest feasible sequence length from 4K to 16K for Llama 3.2 1B and Qwen2.5-1.5B, from 8K to 64K for SmolLM2-135M, and from 4K to 32K for SmolLM2-360M and Granite 3.1 1B-A400M. These yield $4\times$ or $8\times$ capacity gains over the PyTorch math baseline. SmolLM2 evaluations beyond its released 8,192-token context use rotary-position extrapolation and establish hardware feasibility only, not validated long-context model quality. Table 15 reports matched-length timing and memory; capacity endpoints do not establish equivalent model accuracy.

M Sliding-window attention

For the multi-head-attention configuration evaluated here, sliding-window attention (SWA) is supported by restricting the inner-loop column range: for each row tile, the streamed column tiles are restricted to those overlapping the window. The affine-fusion structure of Proposition 4.1 is unchanged (α_i and the five α-free accumulators $E_i^\circ, B_i, \widetilde{d(dO)}_i^\circ, R_i, \Omega_i$ are accumulated over the now-shorter rowwise supports), so the two-pass algorithm applies to the restricted attention pattern. This experiment does not establish SWA support in the current GQA/MQA release path.

Table 16 reports an end-to-end second-order step on GPT-2 Small with window $w = N/4$, comparing FLASHBOB against the PyTorch MATH reference. We report only $N \geq 2048$: at shorter lengths, non-attention work and launch overhead dominate the step, and the gains there are in the measurement noise floor.

The speedup and memory ratios grow with N, as in the dense setting.

Table 17: Head-dimension sensitivity for isolated attention BoB on NVIDIA A100 80GB ($N = 4{,}096$, batch 4, 12 heads, causal mask, BF16 inputs with FP32 reductions). Unlike the $N \leq 4{,}096$ rows of Table 1, this sweep releases the MATH reference outputs before timing, which accounts for the 384–400 MiB lower peak memory at $d = 64$. Time and memory ratios use FLASHBOB at the same d as the denominator.

d	Method	Time (ms)	Peak mem. (MiB)	Time ratio	Mem. ratio
32	FLASHBOB	**14.214**	**423.0**	1.00×	1.00×
	HVP-M	111.312	14,005.4	7.83×	33.11×
	HVP-S	82.393	14,010.0	5.80×	33.12×
64	FLASHBOB	**25.676**	**843.0**	1.00×	1.00×
	HVP-M	114.726	14,137.4	4.47×	16.77×
	HVP-S	89.032	14,154.0	3.47×	16.79×
128	FLASHBOB	**81.694**	**1,683.0**	1.00×	1.00×
	HVP-M	120.029	14,401.4	1.47×	8.56×
	HVP-S	103.995	14,442.0	1.27×	8.58×

N Head-dimension sensitivity and autotuned tile configurations

The A100 isolated-kernel sweep in Table 1 (left) uses GPT-2-style attention heads with $d = 64$, while Table 1 (right) evaluates $d = 128$ across sequence lengths on B300. Since both the I/O analysis and the kernel implementation are parametric in d, Table 17 additionally tests $d \in \{32, 64, 128\}$ at fixed $N = 4096$ on A100 80GB. Increasing d makes the problem harder for a FlashAttention-style schedule: each tile carries larger Q, K, V, dO-like blocks, the feasible tile size decreases for a fixed on-chip memory budget, and register/SRAM pressure increases.

Extending the optimized FLASHBOB kernels to the larger $d = 256$ regime is an engineering target rather than a change to the algorithm: the two-pass schedule and the I/O analysis of Section 4.4 are parametric in d, and Theorem 4.3 applies unchanged. The binding constraints at $d = 256$ are implementation-level. First, register pressure: the $B_r \times d$ row-resident tiles ($Q_i, dO_i, U_{Q,i}, O_i$) and the four $B_r \times d$ row accumulators of Proposition 4.1 ($\widetilde{d}(dO)^\circ, \Omega, B, R$) compete for the same register file inside Pass 1, and the per-thread register requirement grows with d for a fixed tile shape. Second, shared-memory pressure: the simultaneously live interaction-shaped tiles ($P_{ij}, dP_{ij}, F_{ij}, C_{ij}, \widetilde{d}(P)^\circ_{ij}$) scale as $B_r B_c$ and force a smaller B_c at larger d, which raises the streamed-tile re-load count proportionally. Third, tensor-core tiling: the native FlashAttention-3 pipelines on Hopper assume specific (B_r, B_c, d) shape tuples that do not transfer verbatim to the BoB schedule, so a $d = 256$ port would require re-deriving the warp specialization and asynchronous load schedules used by FlashAttention-3 and FlashAttention-4. We view a $d = 256$ specialization as the clearest follow-up systems work.

Table 17 shows that FLASHBOB remains the fastest and most memory-efficient method at every tested head dimension on A100. The runtime advantage is largest at smaller head dimensions: FLASHBOB is 7.83× faster than HVP-M and 5.80× faster than HVP-S at $d = 32$. On A100 at $N = 4096$, the speedup narrows at $d = 128$ (1.47× over HVP-M and 1.27× over HVP-S), whereas on B300 (Table 1), FLASHBOB achieves 2.08–2.36× speedups across $N = 2048$–16,384. The memory advantage remains substantial: on A100 at $N = 4096$, the HVP baselines allocate roughly 14 GiB across the sweep, while FLASHBOB uses 423 MiB at $d = 32$, 843 MiB at $d = 64$, and 1,683 MiB at $d = 128$, an 8.6× reduction even at $d = 128$.

N.1 Triton autotuning search space and tile configurations by d

Both passes autotune over a shared candidate family, fixing `num_warps=4` and CTA swizzle group size $\in \{4, 8\}$, and pruning candidates before compilation when either of two static hardware budgets is exceeded: (i) more than 16,384 live FP32 accumulator elements per CTA (128 registers per thread at four warps), or (ii) more than 95 KiB of estimated shared memory for staged operand tiles. Pass 1 holds four $B_r \times d$ vector accumulators and four staged column tiles, whereas Pass 2 holds two $B_c \times d$ accumulators and three staged row tiles. In addition, at $d \geq 128$, candidates with $B_r B_c > 512$ or Pass 2 $B_c > 32$ are pruned. Every surviving tile dimension is a multiple of 16 and therefore eligible

for Tensor Core MMA instructions when lowered via `tl.dot`. Table 18 lists the surviving (B_r, B_c) and pipeline stage configs for each d.

Table 18: Autotuned Triton tile shapes (B_r, B_c) and pipeline depths (`num_stages`) surviving static register and shared-memory pruning for each head dimension d (all with 4 warps and CTA group size $\in \{4, 8\}$).

	Pass 1 (row-major)		Pass 2 (column-major)	
d	Tile shapes (B_r, B_c)	Stages	Tile shapes (B_r, B_c)	Stages
32	$(16,16), (32,16), (32,32), (64,16), (64,32), (128,32)$	$2, 3$	$(16,16), (16,32), (32,32), (64,32)$	$2, 3$
	$(32,64)$	2		
64	$(16,16), (32,16), (64,16)$ $(32,32), (64,32)$	$2, 3$ 2	$(16,16), (16,32), (32,32)$	$2, 3$
128	$(16,16), (32,16)$	2	$(16,16), (16,32)$	$2, 3$

O FineWeb training reproducibility

This section reports the full configuration for the 1B-token FineWeb training experiments in Section 5.3: the main three-way comparison on 8×A6000 48GB (AdamW, Sophia-H with the PyTorch math reference, and Sophia-H with FLASHBOB) and the six-run matched hybrid baseline sweep across Hessian intervals on 8×A100 80GB. Within each matched pair, the Sophia-H runs use the same model, dataset stream, seed, optimizer hyperparameters, batch geometry, precision, token budget, and Hessian-estimation interval. In the original A6000 comparison, the PyTorch math arm uses math attention on ordinary steps as well as Hessian steps. In the A100 hybrid comparison, both arms use fused attention on ordinary steps and differ only in the attention BoB implementation during Hessian estimation. The hybrid comparison is therefore the controlled estimate of the BoB substitution's contribution to training wall time.

We use the Hugging Face dataset identifier `VisionTheta/fineweb-1B`, which is a 1B-token prepared subset derived from the FineWeb corpus of Penedo et al. [42]. The license and source attribution therefore follow the upstream FineWeb dataset, while the exact artifact used for these experiments is the `VisionTheta/fineweb-1B` Hugging Face dataset. We report the dataset identifier explicitly to make the data order and preprocessing reproducible.

All FineWeb curves in Figure 2 are single runs with seed 17. The AdamW curve is included as a first-order optimizer reference, not as a backend-equivalence experiment. To avoid comparing against a weak AdamW baseline, we selected the plotted AdamW curve from a small learning-rate sweep, described below.

The Sophia-H learning-rate schedule is intentionally kept as used in the best observed Sophia-H configuration from our tuning runs. Although the configured warmup length of 2000 steps exceeds the 1908-step 1B-token run, this setting gave the best observed Sophia-H behavior and is held fixed across all Sophia-H arms.

AdamW reference. Figure 2 also includes an AdamW reference curve. This run is not used to test backend equivalence; it is included as a familiar first-order optimizer reference for the same 1B-token FineWeb setup. To avoid comparing against a poorly tuned AdamW baseline, we ran a small learning-rate sweep over peak learning rates $\{3.0\times10^{-4}, 4.5\times10^{-4}, 6.0\times10^{-4}, 8.0\times10^{-4}, 1.0\times10^{-3}\}$. Each sweep point used AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, weight decay 0.1, gradient clipping 1.0, linear warmup followed by cosine decay, and minimum learning rate 0.1 times the peak learning rate. The AdamW curve in Figure 2 uses the best-performing configuration from this sweep.

P Detailed comparison with FlashBack

This appendix expands on the main-text comparison in Table 2. FlashBack [13] fuses the entire BoB into a single GPU kernel parallelized over rows, runs two sequential inner loops over column tiles per

Table 19: Shared data, batching, and evaluation configuration for the FineWeb GPT-2 Small training runs.

Setting	Value
Dataset	`VisionTheta/fineweb-1B`
Dataset name	`default`
Dataset mode	streaming
Tokenizer	`gpt2`
Sequence length	2048
Training token budget	1,000,000,000 tokens
Maximum optimizer steps	1908
Seed	17
Precision	bfloat16
Number of GPUs	8 (8×A6000 48GB main; 8×A100 80GB hybrid baseline)
Per-GPU batch size	2 sequences
Gradient accumulation steps	16
Effective global batch size	256 sequences
Local tokens per optimizer step	65,536
Global tokens per optimizer step	524,288
Evaluation interval	every 250 optimizer steps
Evaluation batches	32
Evaluation batch size	2 per GPU
Prefetch factor	2

Table 20: GPT-2 Small model configuration used for the FineWeb training runs.

Setting	Value
Architecture	GPT-2 Small
Layers	12
Attention heads	12
Embedding dimension	768
Context length	2048
Vocabulary size	50,257
Bias terms	enabled
Total parameters	125,226,240
Attention parameters	28,348,416
Non-attention parameters	96,877,824
Embedding parameters	40,170,240
Attention parameter fraction	22.64%

row block, and accumulates the column-owned outputs $(\widetilde{d}(K), \widetilde{d}(V))$ through global atomic adds. FLASHBOB instead splits the work into a row-major Pass 1 that finalizes $(\widetilde{d}(Q), \widetilde{d}(dO), \alpha, E)$ via affine fusion (Proposition 4.1), and a column-major Pass 2 that finalizes $(\widetilde{d}(K), \widetilde{d}(V))$ as a natural column reduction. Both schedules recompute the probability tile P twice across the BoB; they differ in how that recomputation work is distributed, how column-owned outputs are accumulated, and how much state is simultaneously live on chip.

Aspect	FlashBack	FLASHBOB
Kernel launches	1 (single fused)	2 (row-major + column-major)
$\widetilde{d}(K), \widetilde{d}(V)$ writes	`atomic_add`	natural column reduction (no atomics)
Affine fusion of α	no	yes
Row-side work in first sweep	scalar summaries only	all row outputs $(\widetilde{d}(Q), \widetilde{d}(dO), E)$
On-chip row scalars	kept on chip (no HBM round trip)	written and re-read via (α, E)

Configuration sweeps and measurement details. For both methods, we autotune over candidate kernel configurations and report the lowest steady-state latency at each sequence length N, excluding compilation and autotuning time. On NVIDIA B300 at $d = 128$, we swept FlashBack over block sizes in $\{16, 32\}$, pipeline stages in $\{1, 2, 3\}$, and warp counts in $\{4, 8\}$; block size 32 with 1 stage and 4 warps is fastest across all tested lengths. The B300 sweep ($d = 128$) evaluates all powers of two from

Table 21: Sophia-H optimizer and Hessian-estimation configuration. These settings are identical for the PyTorch math-backend, hybrid baseline, and FLASHBOB Sophia-H runs.

Setting	Value
Optimizer	Sophia-H
Learning rate	6.0×10^{-4}
Minimum learning rate	3.0×10^{-5}
β_1	0.96
β_2	0.99
ϵ	10^{-12}
γ	0.01
Weight decay	0.2
Gradient clipping	1.0
Hessian-estimation interval	every 10 steps (main); every $\{10, 5, 2\}$ steps (hybrid sweep)
Hutchinson samples	1
Hutchinson batch size	32
Warmup steps	2000
Maximum optimizer steps	1908

Table 22: Sophia-H backend arms compared in Section 5.3.

Configuration	Ordinary steps (fwd/bwd)	Hessian steps (BoB)	Hardware
Sophia-H (math ref.)	PyTorch `math` SDPA	PyTorch `math` autograd	8×A6000 48GB
Sophia-H (hybrid)	FlashAttention SDPA	PyTorch `math` autograd	8×A100 80GB
Sophia-H (FLASHBOB)	FlashAttention SDPA	FLASHBOB two-pass kernel	8×A6000 / 8×A100

$N = 256$ to 65,536, whereas the A6000 sweep ($d = 64$) evaluates the same range except $N = 8{,}192$ (Table 2 omits $N = 8{,}192$ to align the two columns, and on B300 at $N = 8{,}192$ FlashBack requires 619.00 ms versus 68.23 ms for FLASHBOB). We did not run A6000 comparisons at $N = 131{,}072$ or $N = 262{,}144$; the longer-sequence FLASHBOB results in Table 10 were measured on A100 80GB.

We report shared-hardware wall time rather than FlashBack peak memory in Table 2 because the public FlashBack implementation uses JAX/XLA, whose allocator reservation is not directly comparable to PyTorch's `torch.cuda.max_memory_allocated`.

Matched MQA comparison. Because FlashBack supports native multi-query attention ($H_{KV} = 1$), we also compared both methods directly at the saved-state second-order attention boundary on one NVIDIA B300 using batch size 1, $H_Q/H_{KV} = 8/1$, and matched BF16 inputs and outputs across $N = 2{,}048$–$131{,}072$: FLASHBOB is $3.96\times$–$5.04\times$ faster than FlashBack at $d = 64$ and $5.87\times$–$8.90\times$ faster at $d = 128$.

The complete A6000 and B300 timing sweeps appear in Table 2.

Q Broader long-context systems context

Long-context systems address complementary bottlenecks at different levels of execution. Sequence-parallel training partitions attention across GPUs and coordinates communication between devices [21, 31, 29, 14]. Inference systems instead optimize KV-cache management and attention execution during serving [28, 57]. Other long-sequence kernels, such as FlashFFTConv [16], restructure distinct sequence operators to exploit the GPU memory hierarchy.

FLASHBOB addresses a different operation: exact backward-over-backward through softmax attention on a single GPU. Its two-pass schedule avoids $N \times N$ HBM intermediates and accumulates row- and column-owned outputs without global atomics. Because the kernel executes independently on each local attention head, standard data parallelism, tensor parallelism that shards attention heads, pipeline parallelism, and head-sharded sequence parallelism with all-to-all communication (such as DeepSpeed-Ulysses [21]) leave FLASHBOB unchanged. Ring and context parallelism [31], which shard the sequence dimension across P ranks, require two scalar ring all-reduces across the sequence group between Pass 1 and Pass 2 to sum the partial row scalars (α_i, E_i) ($4N/P$ FP32 scalars per rank, which can be overlapped with streaming the key-value blocks) before Pass 2 accu-

Table 23: AdamW learning-rate sweep used to select the reference AdamW curve in Figure 2. All runs use AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, weight decay 0.1, gradient clipping 1.0, linear warmup followed by cosine decay, and minimum learning rate 0.1 times the peak learning rate.

Run	Peak LR	Min LR	Role
1	3.0×10^{-4}	3.0×10^{-5}	Conservative baseline
2	4.5×10^{-4}	4.5×10^{-5}	Stable intermediate setting
3	6.0×10^{-4}	6.0×10^{-5}	Canonical anchor
4	8.0×10^{-4}	8.0×10^{-5}	Aggressive setting
5	1.0×10^{-3}	1.0×10^{-4}	Highest-LR stress test

Table 24: Selected AdamW reference-run configuration used for the FineWeb plot. This was the best-performing run from the sweep in Table 23.

Setting	Value
Optimizer	AdamW
Attention backend	FlashAttention backend
Seed	17
Learning-rate schedule	linear warmup followed by cosine decay
Peak learning rate	1.0×10^{-3}
Minimum learning rate	1.0×10^{-4}
Warmup steps	150
Maximum optimizer steps	1908
β_1	0.9
β_2	0.95
ϵ	10^{-8}
Weight decay	0.1
Gradient clipping	1.0
Precision	bfloat16
Compilation	disabled
Per-GPU batch size	2 sequences
Gradient accumulation steps	16
Effective global batch size	256 sequences
Global tokens per optimizer step	524,288
Training token budget	1,000,000,000 tokens
Evaluation interval	every 250 optimizer steps
Evaluation batches	32

mulates $(\widetilde{d}(K), \widetilde{d}(V))$; we have not implemented this distributed ring schedule or inference-oriented paged KV layouts. Finally, PyTorch's Fully Sharded Data Parallel (`FSDP`) currently lacks native double-backward hooks for parameter resharding during a second backward pass, so our multi-GPU second-order training runs in Section 5.3 use Distributed Data Parallel (`DDP`).

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Yes. All claims made in the abstract and introduction were taken from experimental results, which formed the basis of our contributions.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We make sure to state all the current limitations of the project, so that all stakeholders can be privy of them. We believe that some of these limitations are exciting future work, so we ensured that we stated them exactly.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: We state all assumptions and ensure that we have a complete proof to accompany our theoretical results. We also ensure that the proofs are easy to read and self-contained.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We specify all that is needed to reproduce all our main claims, including a very detailed pseudocode of our Triton kernel. We also specify the exact setup and software configuration needed to reproduce our results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Our training dataset is publicly available, and we release code artifacts that detail, verbatim, how to reproduce all experiments.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We detail all that is needed to replicate our experiments, in addition to the actual experimentation code as well.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: We do not report error bars and/or statistical significance of our experiments, due to computational cost.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We stated, verbatim, all the GPUs we used, along with the exact software stack needed to replicate all our experiments. In addition, we provide code to make this process easier for all stakeholders.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: To the best of our knowledge, the work uses standard public datasets and does not involve human subjects or other procedures that would conflict with NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

Justification: We do not discuss broader positive and/or negative societal impacts, since this work primarily focuses on optimizing throughput for neural network training.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: We do not release new high-risk pretrained models, scraped datasets, or other assets that would require a responsible-release safeguard discussion.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We cite all existing works that we use in our experiments, in order to preserve fairness.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We will release fully documented code, open-sourcing our Triton kernels and experimentation suite. We ensured that the code is documented, making it easy for all stakeholders to interact with the codebase.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: Our work neither involves crowdsourcing nor human-subject research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

Justification: Our work does not involve crowdsourcing or human-subject research, so IRB approval is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: LLMs are not an important or non-standard component of our proposed method, or experimental methodology. We only use LLMs for writing, editing, and/or formatting purposes.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.