TURBOPAIRFORMER: FAST AND STABLE PROTEIN FOLDING MODEL TRAINING WITH AN OPTIMIZED TRIANGLE ATTENTION KERNEL

Anonymous authors
Paper under double-blind review

ABSTRACT

Triangular attention is a core computation in AlphaFold3-style biomolecular models, with computational cost that grows cubically with token count. FlashAttention reduces attention memory traffic through IO-aware tiling and backward recomputation. However, the shared pair bias in triangular attention adds a gradient reduction across attention slices to the usual reductions over queries and keys. The open source backends we examine handle these reductions through repeated probability recomputation, floating point atomics, or full score-gradient storage. Separately, computing the softmax backward correction from BF16-rounded forward outputs incurs a loss of numerical precision. We present TurboPairFormer, a triangular attention implementation for NVIDIA Hopper GPUs that addresses these two issues. Our key tile parallel backward algorithm recomputes each probability tile once and reuses it for the query, key, value, and pair-bias gradients, using ordered partial reductions to achieve deterministic accumulation without floating point atomics or full score gradient storage. Our output residual compensation retains a BF16 approximation of the output rounding residual and uses it to compute the backward correction more accurately in FP32, without changing the BF16 output. With BF16 inputs at crop sizes 384, 640, 768 and head dimensions 16, 32, TurboPairFormer achieves the lowest mean query, key, and pair-bias gradient RMSE against an FP64 reference among eight evaluated implementations; residual compensation mitigates the precision loss from output rounding, reducing these RMSE values by 28–47% in controlled ablations. All four gradients are bitwise identical across five repeated calls in all 600 input cases under fixed execution conditions. Integrated into OpenFold3 together with our triangle multiplication kernels, TurboPairFormer achieves the lowest GPU computation time per optimizer step among eight evaluated backend configurations on 16 H100 GPUs, with speedups of $1.73\times$ over OpenFold3's Triton backend and $1.13\times$ over cuEquivariance at crop size 768.

1 INTRODUCTION

AlphaFold3 predicts complexes containing proteins, nucleic acids, and small molecules within a unified model (Abramson et al., 2024), and open-source implementations such as OpenFold3 support further training and development (The OpenFold3 Team, 2025). These models represent each standard residue or each atom of any other component as a token, and maintain a learned representation for every pair of tokens, repeatedly updated through triangular attention and triangle multiplication. Fixing a token index o, which we call the outer index, selects a slice in which N queries attend to N keys, and all N slices share one pair bias. The total computation scales cubically with token count, so its cost grows rapidly as training crops include more residues and interaction partners.

IO-aware kernels such as FlashAttention fuse attention into one tiled kernel and recompute probabilities in the backward pass instead of storing them (Dao et al., 2022); their successors improve parallelism and overlap data movement with computation on Hopper GPUs (Dao, 2024; Shah et al., 2024). However, these kernels target standard attention and do not handle a bias shared across slices. The open-source triangular-attention kernels we inspect therefore make different compromises in the backward pass (Figure 1b): TriFast (Atkinson, 2025) and Protenix (ByteDance AML AI4Science Team et al., 2025) recompute the probabilities three times, DS4Sci (Song et al., 2023) accumulates

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

the pair-bias gradient with floating-point atomics, and ScaleFold (Zhu et al., 2024) stores the full FP32 score gradient (6.75 GiB per call at crop 768).

The shared bias adds a third backward reduction direction (Figure 1a): dQ sums over keys, dK/dV over queries, and the pair-bias gradient over outer indices. All four reuse the same probability tiles, so recomputing each tile once requires cross-block reductions along axes not covered by a thread block. FlashAttention-3's deterministic mode orders in-place dQ updates with semaphores, and DASH improves its scheduling (Shah et al., 2024; Qiang et al., 2026), but neither computes a bias gradient. The forward pass matters as well: with 64-row query tiles and head dimension 32, a bias tile holds as many bytes as the key and value tiles combined, and recycling and checkpointing run two to five forwards per backward.

Precision is a separate problem. The softmax correction $\delta = \langle dO, O \rangle$ is usually computed from the saved BF16 output, whose rounding error shifts every score gradient in a row in the same direction; a deterministic kernel reproduces this error on every run. Our challenge is to share one probability recomputation across four gradients with fixed accumulation orders and enough parallelism, stage the shared bias without forward stalls, and compute δ accurately.

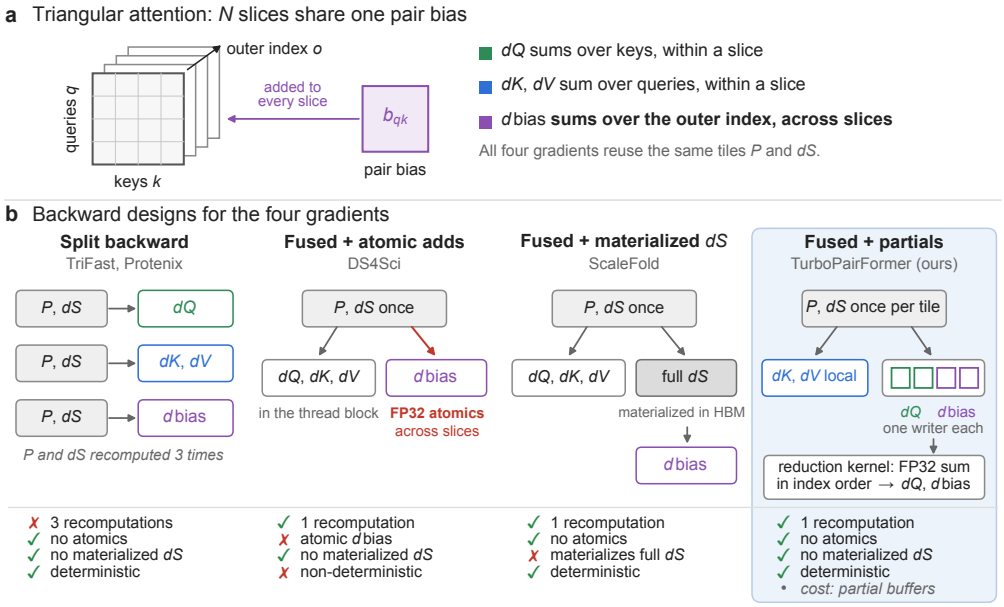


Figure 1: **Triangular attention and backward designs.** (a) The shared pair-bias gradient sums across slices, over the outer index. (b) Split kernels recompute P and dS in separate passes for different gradient groups; fused kernels combine the pair-bias gradient with atomics or a stored score gradient. TurboPairFormer (right) recomputes P and dS once per tile, keeps dK and dV in the thread block, and writes dQ and pair-bias partials with unique writers, which FP32 kernels reduce in index order, at the cost of partial buffers. MegaFold and OpenFold3 Triton recompute twice and use atomics (Appendices E and F).

TurboPairFormer, our triangular-attention kernel for Hopper GPUs, makes four contributions:

1. **Work partitioning for three reduction directions:** We analyze how the shared pair bias trades recomputation against parallelism and storage (Section 2.1 and Appendix I), and assign each thread block a key tile and a group of outer indices, which keeps dK and dV local and leaves two cross-block reductions.
2. **Single recomputation with ordered partial reductions:** Each probability tile serves all four gradients on chip. Query and pair-bias partials have unique writers and are reduced in fixed FP32 orders, without blocks waiting for one another. Gradient bits do not depend on thread-block completion order or on supported workspace schedules; the cost is a query-partial buffer cubic in crop size (Appendix E).

3. **Output-residual compensation:** The forward epilogue saves a BF16 approximation of the output-rounding residual, which the backward pass adds to the BF16 output in FP32 when computing δ (Qiu & Yao, 2026; Henry et al., 2019). This lowers mean query-, key-, and pair-bias-gradient RMSE by 28–47% in controlled ablations, without changing the BF16 output or adding a recomputation.
4. **Forward staging of the shared bias:** The forward kernel loads each bias tile with the Tensor Memory Accelerator alongside the key and value tiles, reads it directly in the score-accumulator register layout, prefetches the key mask, and evaluates probabilities in 16-key chunks just before value aggregation, without changing the floating-point operations or their order.

We evaluate TurboPairFormer on H100 GPUs at crops 384, 640, and 768 and head dimensions 16 and 32. One attention call takes 35.6–45.6% less forward-plus-backward time than cuEquivariance (NVIDIA Corporation, 2026) and ranges from 7.4% faster to 3.6% slower than Protenix (Appendix H.1); the forward changes cut forward time by 25.5–27.5% at $D = 32$ (Appendix H.2). Its mean query-, key-, and pair-bias-gradient RMSE against an FP64 reference is the lowest among eight attention implementations. All four gradients are bitwise identical across five repeated calls in all 600 fixed-input cases (Appendix G.8), whereas cuEquivariance, OpenFold3's Triton attention, and DeepSpeed-Evo pass none; the fastest non-deterministic variants of our kernel take 2.8–4.1% less backward time at $D = 32$ and 11.0–13.7% less at $D = 16$ (Appendix I). Combined with our triangle multiplication kernels, TurboPairFormer has the lowest GPU computation time per step among eight OpenFold3 backend configurations on 16 H100 GPUs, $1.73\times$ faster than OpenFold3's Triton backend and $1.13\times$ faster than cuEquivariance at crop 768. In paired 400-update continuations from a shared step-1000 checkpoint using an earlier build of our kernels, our losses remain finite, whereas OpenFold3's Triton attention produces NaN losses within 10 updates (Section 3.3).

2 TurboPairFormer

TurboPairFormer implements triangular attention and triangle multiplication for OpenFold3 pair updates (TriMul details in Appendix J). The attention kernel combines one probability recomputation, fixed accumulation orders, and output-residual compensation. Figure 1 and Algorithm 1 summarize execution; Table 1 defines the notation.

2.1 Triangular Attention and Its Reduction Axes

OpenFold3 learns features Z for each ordered token pair. Standard amino acids and nucleotides contribute one token each; other components use one token per atom (Abramson et al., 2024). A crop selects N tokens, possibly across chains. Starting-node triangular attention fixes token o: the query pair (o, q) attends to (o, k) in slice $Z_{o,:,:}$. This *outer index* identifies a token, not a chain or geometric plane. Ending-node attention uses the same core after permuting the pair axes (Appendix B).

The projected tensors Q, K, V have logical shape $[B, N_o, H, N, D]$, where $N_o = N$, B is the local batch size, H the head count, and D the head dimension. The pair bias describes (q, k) and is shared across outer indices o. Omitting batch and head indices, the attention core computes

$$\begin{aligned} S_{oqk} &= \frac{\langle Q_{oq:}, K_{ok:}\rangle}{\sqrt{D}} + \mathrm{bias}_{qk} + M_{oqk}, \\ P_{oq:} &= \mathrm{softmax}_k(S_{oq:}), \quad O_{oq:} = \sum_k P_{oqk} V_{ok:}, \end{aligned} \tag{1}$$

with additive mask M. Given the incoming gradient dO, the score gradient dS supplies three of the four input gradients:

$$\begin{aligned} dQ_o &= \frac{dS_o K_o}{\sqrt{D}}, \qquad dK_o = \frac{dS_o^\top Q_o}{\sqrt{D}}, \\ dV_o &= P_o^\top dO_o, \quad d\,\mathrm{bias} = \sum_o dS_o. \end{aligned} \tag{2}$$

These gradients reuse the same intermediates but sum along different axes: keys for dQ, queries for dK/dV, and outer indices for $d\,\mathrm{bias}$. This determines which results a thread block can finish locally and which require a later reduction. Appendix D.1 derives the identities.

2.2 One Recomputation for Four Gradients

Each thread block, or cooperative thread array (CTA), owns one key tile and a group of four outer indices, called an outer group, for one example and head. It traverses query tiles in ascending order, using 64 keys and 32 query rows per tile. Different key tiles provide independent work. Within each owned slice, dK/dV accumulators stay in FP32 registers throughout the traversal.

For each query tile, the thread block recomputes $P = \exp(S - \mathrm{LSE})$ from the scores and saved row log-sum-exp. It computes $dP = dO\,V^\top$ and forms $dS = P \circ (dP - \widehat{\delta})$, where the compensated row correction $\widehat{\delta}$ is described in Section 2.4. The tile P supplies dV, while dS supplies dK, dQ, and the pair bias. Shared memory makes these intermediates available to their consumers before the next tile. Each thread block writes its query contributions as partials indexed by key tile and its bias contributions as partials indexed by outer group. Every partial has one writer. Separate reduction kernels sum these BF16 partials in FP32, without recomputing probabilities or storing the full score gradient. Proposition D.1 establishes the partitioning.

Algorithm 1 TurboPairFormer backward pass

Require: $Q, K, V, dO, O_{16}, E_{16}, \mathrm{bias}, M, \mathrm{LSE}$ in HBM; tile sizes τ_q, τ_k.

Compute $\widehat{\delta} = \mathrm{rowsum}(dO \circ (O_{16} + E_{16}))$ in FP32; store it in HBM.
Set $J_Q = \lceil N/\tau_q \rceil$, $J_K = \lceil N/\tau_k \rceil$, and outer groups $\mathcal{L}_\ell = \{4\ell, \ldots, 4\ell + 3\}$.
for each key tile $t \in \{0, \ldots, J_K - 1\}$ and outer group ℓ in parallel **do**
  Load K_{ot}, V_{ot}, M_{ot} for $o \in \mathcal{L}_\ell$ from HBM to shared memory.
  Initialize $dK_{ot} = dV_{ot} = 0$ in FP32 registers for $o \in \mathcal{L}_\ell$.
  for $i = 0, \ldots, J_Q - 1$ **do**
    Load query tiles Q_{oi}, dO_{oi} and shared bias tile bias_{it} into shared memory.
    Load row statistics $\mathrm{LSE}_{oi}, \widehat{\delta}_{oi}$ for $o \in \mathcal{L}_\ell$.
    for $o \in \mathcal{L}_\ell$ **do**
      Compute $S_o = Q_{oi} K_{ot}^\top / \sqrt{D} + \mathrm{bias}_{it} + M_{ot}$.
      Reconstruct $P_o = \exp(S_o - \mathrm{LSE}_{oi})$.
      Compute $dP_o = dO_{oi} V_{ot}^\top$ and $dS_o = P_o \circ (dP_o - \widehat{\delta}_{oi})$.
      Accumulate $dV_{ot} \leftarrow dV_{ot} + P_o^\top dO_{oi}$.
      Accumulate $dK_{ot} \leftarrow dK_{ot} + dS_o^\top Q_{oi}$.
      Write $dQ^{(t)}_{\mathrm{part},oi} = dS_o K_{ot}$ to HBM in BF16.
    end for
    Write $d\,\mathrm{bias}^{(\ell)}_{\mathrm{part},it} = (dS_{o_0} + dS_{o_1}) + (dS_{o_2} + dS_{o_3})$ to HBM in BF16.
  end for
  Write $dK_{ot}/\sqrt{D}$ and dV_{ot} to HBM in BF16 for $o \in \mathcal{L}_\ell$.
end for
In a reduction kernel, sum $dQ^{(t)}_{\mathrm{part}}$ in increasing t in FP32; scale by $1/\sqrt{D}$ and store dQ in BF16.
In a reduction kernel, sum $d\,\mathrm{bias}^{(\ell)}_{\mathrm{part}}$ in increasing ℓ in FP32; store $d\,\mathrm{bias}$ in BF16.

Algorithm 1 omits batch and head indices; i, t index query and key tiles, and $o_r = 4\ell + r$. Row statistics and the key mask broadcast along the other tile axis; fully masked rows and invalid outer positions have $P_o = 0$. All matrix products use BF16 operands and FP32 accumulation, so P_o and dS_o are rounded when consumed by a matrix product; the bias tree instead uses FP32 dS_o. The main kernel uses two warpgroups (256 threads), WGMMA for S_o, dP_o, and `mma.sync` for gradient contractions[1]. Preprocessing and reductions use separate launches; Appendix D.3 specifies synchronization and instruction order.

2.3 Deterministic Reductions and Workspace Scheduling

Partitioning work across key tiles and outer groups exposes parallelism, but distributes contributions to dQ and the shared pair-bias gradient across thread blocks. We fix their floating-point addition order to make these reductions reproducible. TurboPairFormer separates the production of these contributions from their reduction: each block writes its partials to unique addresses, and a later kernel consumes them in a prescribed order. The addition sequence is therefore independent of block completion order.

[1] NVIDIA PTX ISA 8.8, CUDA 12.9.1 archive (2025): `wgmma.mma_async` and `mma.sync`.

For each dQ element, the reducer adds BF16 partials sequentially in increasing key-tile index t, using an FP32 accumulator. The pair-bias reduction follows the same principle along the outer axis: each four-slice group first forms one BF16 partial through a fixed FP32 addition tree, and the reducer adds these partials in increasing group index ℓ. The dK/dV accumulations remain local to their owning block and follow ascending query-tile order. Final scaling and output conversion occur after the respective reductions, as specified in Algorithm 1. These fixed arithmetic paths require no global floating-point atomic accumulation.

The workspace planner changes how much work is resident in each launch while preserving these arithmetic paths. It partitions examples, query rows, and outer groups into consecutive chunks. After reducing a chunk of bias partials, the next launch reloads the FP32 running sum and continues with the next outer group. If query traversal spans launches, dK/dV likewise resume their FP32 accumulators at the next query tile. A launch boundary thus inserts only a store and reload; it introduces neither a new subtotal nor an extra rounding step. For fixed inputs, binary, device, tile sizes, and outer grouping, supported workspace schedules consequently produce bitwise-identical gradients. Appendix D.4 proves this property; Algorithms 3 and 4 specify the launch boundaries.

In FlashAttention-3's deterministic backward, semaphores order in-place dQ updates (Qiang et al., 2026). TurboPairFormer orders the later reductions, allowing key-tile blocks to write their partials independently. This approach also covers the shared pair-bias gradient. Appendix I compares these designs with FlashAttention-2 and DASH.

The cost is storage for the partials and an additional pass to read them. Reusing the pair-bias buffers limits their resident size, but the current implementation retains all dQ partials, requiring $\Theta(BHDN^3/\tau_k)$ bytes when $N_o = N$. Workspace scheduling does not remove this cubic allocation. Whether it increases peak training memory depends on which other tensors are live during the reduction; Appendix E gives the allocation sizes and lifetime analysis.

2.4 Preserving Precision with an Output Residual

The softmax backward uses a row correction $\delta = \langle dO, O \rangle$, available from the forward output without another probability traversal. Computing it from the saved BF16 output loses information discarded during output conversion. Following analyses of attention precision and multiple-component BF16 representations (Qiu & Yao, 2026; Henry et al., 2019), we save a BF16 approximation to the difference between the FP32 forward result O_{32} and public output O_{16}:

$$E_{16} = \mathrm{RN}_{16}(O_{32} - O_{16}), \qquad \widehat{\delta}_{oq} \approx \langle dO_{oq:}, O_{16,oq:} + E_{16,oq:} \rangle. \tag{3}$$

Here RN_{16} rounds to BF16. The subtraction, reconstruction, and dot product execute in FP32; the approximation sign accounts for floating point evaluation. In our controlled ablations, using $\widehat{\delta}$ in the shared dS reduces the mean RMSE of dQ, dK, and the pair bias gradient, while leaving the public output and dV bitwise unchanged (Section 3.2). The cost is one output-sized BF16 tensor and its memory traffic, without additional probability recomputation. Appendix D.5 specifies the evaluation and analyzes its precision.

Forward pipeline. The forward kernel uses tiled online softmax (Milakov & Gimelshein, 2018), overlapping Tensor Memory Accelerator[2] transfers of keys, values, and pair bias with tensor core computation through alternating shared-memory buffer sets. Probability evaluation is interleaved with value aggregation, and the output epilogue saves the residual alongside the BF16 output. Figure 6 and Algorithm 2 illustrate the pipeline; Appendix C gives the recurrence, staging layout, and hardware mapping.

3 Experiments

We evaluate TurboPairFormer through training performance, backward numerical accuracy against an FP64 reference, training stability, and fine-tuning validation; supplementary hardware measurements, additional accuracy results, component ablations, deterministic replay, and validation details appear in the appendix.

[2]NVIDIA Hopper Tuning Guide, Section 1.4.1.2: Tensor Memory Accelerator, CUDA 12.9.1 archive (2025).

3.1 Training Performance

Setup. We compare TurboPairFormer, Protenix, cuEquivariance, TriFast, MegaFold, DeepSpeed-Evo, ScaleFold, and OpenFold3's Triton backend (also denoted AQLab Triton) at crop sizes 384, 640, and 768. The comparison uses two nodes with eight NVIDIA H100 GPUs per node and 16 samples per optimizer step. Each backend runs one 200-step training trajectory per crop; the reported statistics cover steps 21–200, giving 180 measured steps after excluding the first 20 steps. We report GPU computation time per optimizer step, measured with CUDA events and taking the maximum across ranks before averaging. Data loading, host-to-device transfer, and EMA updates are excluded; Appendix G.1 gives the timing boundaries, input construction, and implementation versions.

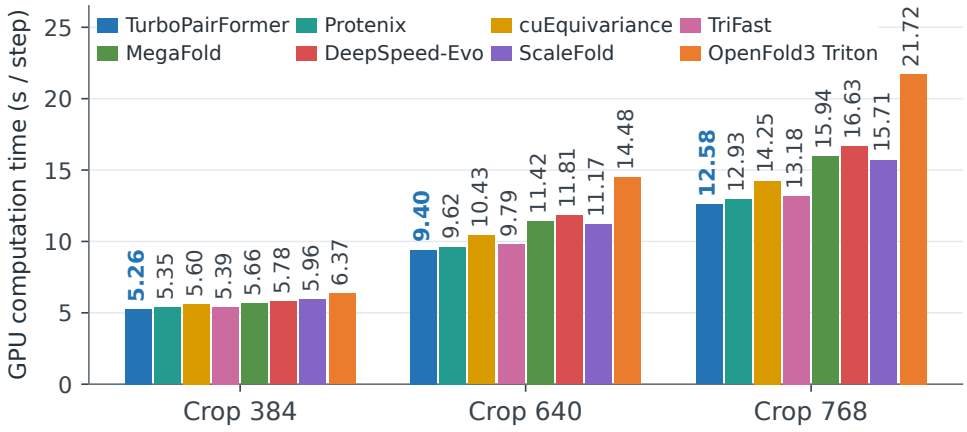


Figure 2: **OpenFold3 training performance on 16xH100 GPUs.** Mean GPU computation time per optimizer step on two nodes with eight H100 GPUs each and 16 samples per step. Statistics cover steps 21–200; data loading, host-to-device transfer, and EMA updates are excluded. Lower is better. TurboPairFormer has the lowest measured mean at all three crop sizes.

Results. **TurboPairFormer achieves the lowest measured GPU computation time at every crop**, taking 5.26, 9.40, and 12.58 seconds per optimizer step at crops 384, 640, and 768 (Figure 2). The corresponding time reductions are 17.4%, 35.1%, and 42.1% relative to OpenFold3 Triton, and 6.1%, 9.9%, and 11.7% relative to cuEquivariance. At crop 768, reductions relative to Protenix, TriFast, MegaFold, DeepSpeed-Evo, and ScaleFold are 2.7%, 4.6%, 21.1%, 24.4%, and 19.9%, respectively; the speedup over OpenFold3 Triton is $1.73\times$.

Supplementary training measurements on eight H200 GPUs, including their separate protocol and timing analysis, appear in Appendix G.4. Component ablations and training-loss comparisons are reported in Appendices J and G.6, respectively.

3.2 Backward Numerical Accuracy

On one H100, we compare eight attention implementations at crops $N \in \{384, 640, 768\}$ and head dimensions $D \in \{16, 32\}$, using ten seeds per shape. All methods receive the same BF16 inputs and incoming gradient, with zero pair bias and mask. Figure 3 reports the mean and sample standard deviation of absolute RMSE for the four public BF16 gradients against an FP64 reference evaluated on the same quantized inputs.

TurboPairFormer has the lowest mean RMSE for dQ, dK, and $d\mathrm{bias}$ in all six shapes, reducing errors relative to cuEquivariance by 31.4–44.4%, 28.7–42.4%, and 16.9–37.9%, respectively. Its dV error is essentially tied with cuEquivariance. Enabling residual compensation lowers the dQ, dK, and $d\mathrm{bias}$ errors in every paired case, by 36.3–47.1%, 28.7–42.4%, and 28.3–43.1% in mean RMSE, while the output and dV stay bitwise identical (Figure 9; Appendix G.7 adds captured-input checks). Appendix G.8 reports five-repeat gradient determinism across 600 fixed inputs.

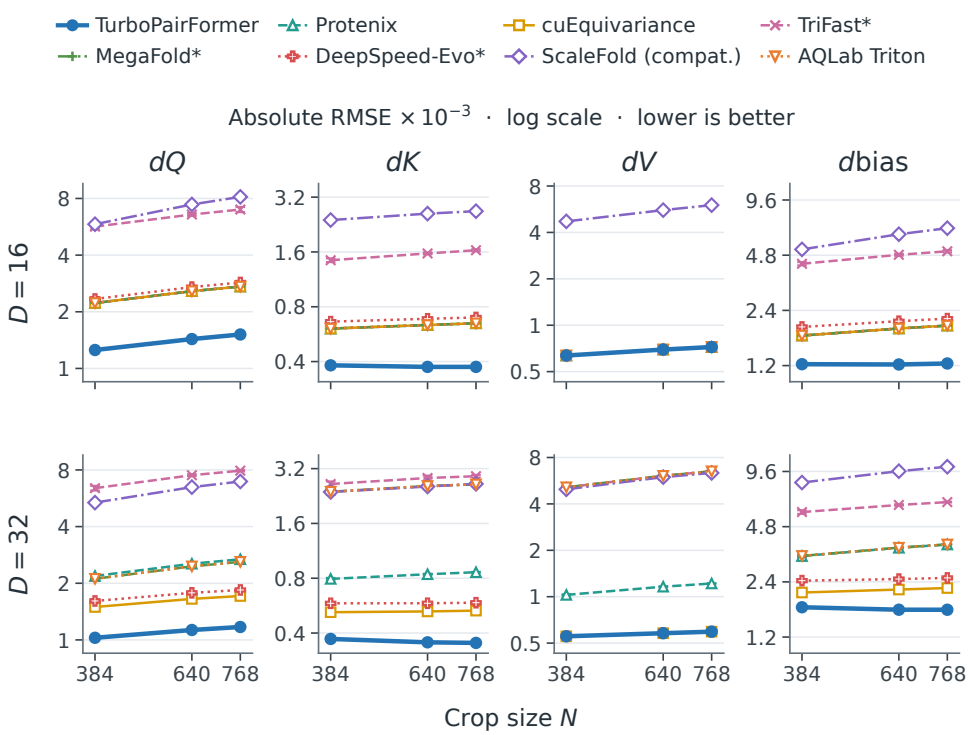


Figure 3: **Attention backward accuracy against FP64.** Rows show $D = 16$ and $D = 32$; columns show dQ, dK, dV, and $d\text{bias}$. Points and error bars report mean absolute RMSE and sample standard deviation over ten seeds. Logarithmic axes show RMSE in units of 10^{-3}. Implementation variants and additional results are described in Appendix G.7.

3.3 Training Stability

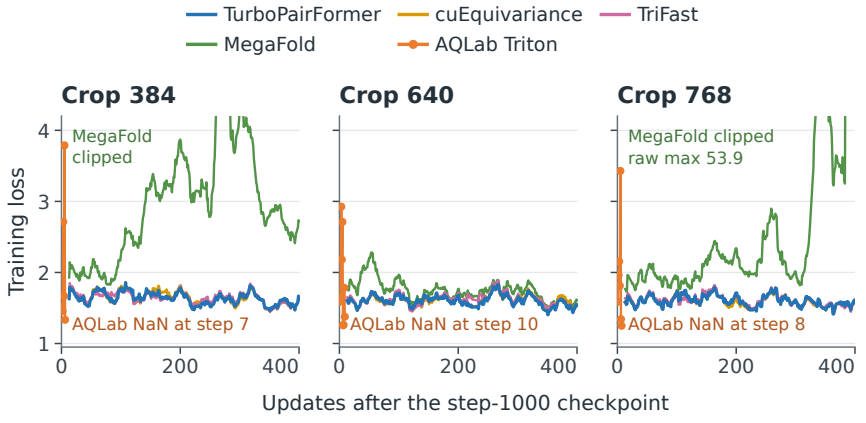


Figure 4: **Training stability after resuming a shared step-1000 checkpoint.** Trailing 25-update mean of the training loss on one H100; every configuration resumes the same checkpoint and trains on the same inputs. TriFast and MegaFold attention run with cuEquivariance TriMul, and AQLab Triton attention with the PyTorch TriMul. AQLab losses are shown per update until they become NaN; the MegaFold curve is clipped.

Five OpenFold3 backend configurations resume the same step-1000 checkpoint, including its optimizer, scheduler, and data-loader state, and train 400 paired updates on identical inputs at crops 384, 640, and 768 on one H100 (Figure 4). Unlike Section 3.1, the AQLab configuration here pairs Triton

attention with the PyTorch TriMul. With OpenFold3's Triton attention (AQLab Triton), the loss becomes NaN after 7, 10, and 8 updates, and with MegaFold attention it spikes to 53.9 at crop 768. TurboPairFormer, cuEquivariance, and TriFast remain finite and closely aligned: over the 400 updates, TurboPairFormer's mean training loss is 1.640, 1.603, and 1.603 at the three crops, compared with 1.646, 1.608, and 1.598 for cuEquivariance. Each crop has one paired trajectory, so the comparison tests numerical stability rather than convergence. These runs used an earlier build of our kernels; Appendix G.6 reports the cumulative losses.

3.4 Fine-Tuning Validation

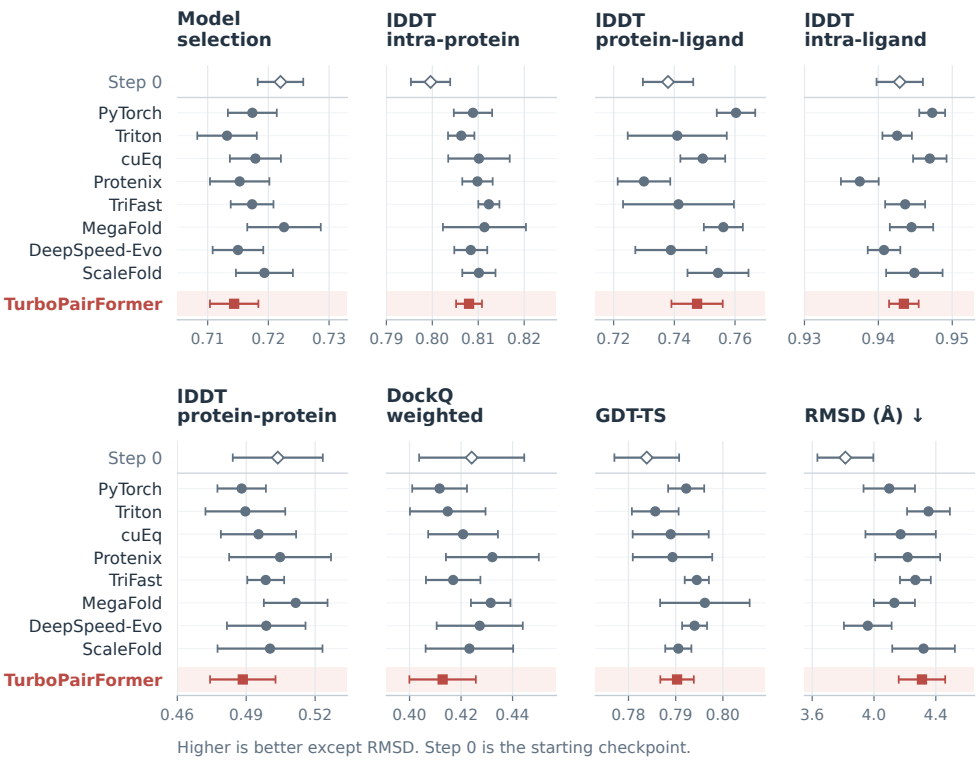


Figure 5: **Preview2 validation after fine-tuning on 16 H100 GPUs.** Points show means and sample SDs over five evaluation seeds on 85 kinase validation complexes after 1,000 global training steps, with step 0 as a reference. Structure metrics average top-1 and best-of-five values; model selection combines lDDT and RASA means (Appendix G.9). Higher is better except for RMSD.

Training setup. We fine-tune OpenFold3 Preview2 (The OpenFold3 Team, 2026) on 9,297 kinase PDB complexes for 1,000 global steps with each of the eight backend configurations of Section 3.1 and with the stock PyTorch configuration, on two nodes with eight H100 GPUs each (settings in Appendix G.9). Each final EMA checkpoint and the Preview2 starting point (step 0) are evaluated on the same 85-complex kinase validation cohort using five evaluation seeds (0–4), which fix the diffusion noise of every target.

Validation results. The model-selection means of the nine fine-tuned checkpoints span 0.713–0.723, with a pooled standard deviation of 0.0045 over evaluation seeds (Figure 5). In paired comparisons over targets, TurboPairFormer's score differs from those of seven other fine-tuned configurations by -0.005 to $+0.001$, with 95% bootstrap intervals that include zero. Its score is 0.0082 below MegaFold's, with an interval of $[-0.0169, -0.0002]$ (Appendix G.9). The measured model-selection scores are broadly similar, with a small advantage for MegaFold in this paired comparison; these results do not establish equivalence across backend configurations.

4 Related Work

Triangular attention and kernel dataflow. AlphaFold2 and AlphaFold3 use triangular updates to pair representations (Jumper et al., 2021; Abramson et al., 2024); FastFold, ScaleFold, and MegaFold accelerate these models through kernel and system optimizations (Cheng et al., 2022; Zhu et al., 2024; La et al., 2025). FlashAttention and its successors establish tiled recomputation and hardware-specific pipelines (Dao et al., 2022; Dao, 2024; Shah et al., 2024; Zadouri et al., 2026). NeuralPLexer3 avoids explicit bias replication (Qiao et al., 2025), FlexAttention compiles PyTorch score modifications (Dong et al., 2025; Paszke et al., 2019), and FlashBias factorizes some structured biases exactly and approximates general biases (Wu et al., 2025). TurboPairFormer retains the dense shared bias. The inspected DS4Sci and ScaleFold kernels already recompute probabilities once, but combine bias gradients through atomic accumulation and full score-gradient storage, respectively (Song et al., 2023; Zhu et al., 2024). Our distinction is combining key-tile parallelism with ordered query and pair-bias partials (Appendix F).

Deterministic gradient reduction. Order-sensitive floating-point summation (Higham, 1993) motivates either order-independent accumulators (Ahrens et al., 2020; Collange et al., 2015) or prescribed reduction sequences. FlashAttention-2's deterministic mode uses FP32 query-gradient buffers per split (Appendix I.2); FlashAttention-3 and FlashAttention-4 order in-place updates with semaphores (Qiang et al., 2026; Zadouri et al., 2026), and DASH improves the scheduling of these ordered updates (Qiang et al., 2026). TurboPairFormer materializes partials for both query and shared pair-bias gradients, allowing key-tile blocks to write independently before fixed FP32 reductions. This avoids waiting for accumulation turns, but requires partial storage; the query-partial buffer remains cubic in crop length (Appendices E and I).

Precision of the backward correction. Mixed-precision training allocates precision selectively (Micikevicius et al., 2018), while multi-component representations retain rounding information in BF16 arithmetic and training states (Henry et al., 2019; Yu et al., 2024). Qiu & Yao (2026) connect attention-output error to the backward correction $\delta = \langle dO, O \rangle$ and adjust the softmax shift to mitigate biased rounding. Attn-QAT computes an auxiliary output from unquantized probabilities for this correction in FP4 attention (Zhang et al., 2026). TurboPairFormer instead extracts a BF16 rounding residual from the existing FP32 output accumulator and reconstructs the saved output for δ. This preserves the public BF16 output and avoids another PV product; it targets output-conversion error, while other rounding errors remain (Appendix D.5).

5 Conclusion

TurboPairFormer combines one probability recomputation for all four triangular-attention gradients, fixed-order partial reductions, output-residual compensation, and shared-bias staging on NVIDIA Hopper GPUs. Under the tested conditions, it achieves bitwise gradient repeatability and the lowest mean query, key, and pair-bias gradient RMSE against an FP64 reference among eight attention implementations. Integrated with our triangle multiplication kernels, it achieves the lowest measured GPU computation time per optimizer step among eight OpenFold3 backend configurations on 16 H100 GPUs. Relative to our deterministic implementation, the fastest tested non-deterministic variants take 2.8–4.1% less backward time at the Pairformer head dimension $D = 32$ and 11.0–13.7% less at $D = 16$. The query-partial buffer grows cubically with crop size; reducing this storage while preserving a fixed accumulation order is the main direction for future work.

AI use statement

Generative AI tools were used only to polish the writing of this paper; the authors are responsible for all content.

Ethics statement

This work studies GPU-kernel optimizations for biomolecular model training using synthetic inputs and existing structural datasets. It involves no human-subject studies, animal experiments, or collection of personal information.

Reproducibility statement

Section 2.1 specifies the operator and gradient ownership graph; Section 2.3 defines the deterministic execution contract; and Section 3 reports the hardware, baselines, timing boundary, and statistical unit. The appendix gives the workspace-invariance argument and the paired-block protocol. The anonymous supplementary material will include the pinned source snapshot with its SHA-256 source and binary identities, the build and ABI manifests, a prebuilt SM90 wheel with a compiler-free clean-install check, the public-facade correctness driver and Compute Sanitizer commands, and the raw timing and energy receipts needed to reproduce every reported table.

References

Josh Abramson, Jonas Adler, Jack Dunger, Richard Evans, Tim Green, Alexander Pritzel, Olaf Ronneberger, Lindsay Willmore, Andrew J. Ballard, Joshua Bambrick, Sebastian W. Bodenstein, David A. Evans, Chia-Chun Hung, Michael O'Neill, David Reiman, Kathryn Tunyasuvunakool, Zachary Wu, Akvilė Žemgulytė, Eirini Arvaniti, Charles Beattie, Ottavia Bertolli, Alex Bridgland, Alexey Cherepanov, Miles Congreve, Alexander I. Cowen-Rivers, Andrew Cowie, Michael Figurnov, Fabian B. Fuchs, Hannah Gladman, Rishub Jain, Yousuf A. Khan, Caroline M. R. Low, Kuba Perlin, Anna Potapenko, Pascal Savy, Sukhdeep Singh, Adrian Stecula, Ashok Thillaisundaram, Catherine Tong, Sergei Yakneen, Ellen D. Zhong, Michal Zielinski, Augustin Žídek, Victor Bapst, Pushmeet Kohli, Max Jaderberg, Demis Hassabis, and John M. Jumper. Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature*, 630(8016):493–500, 2024. doi: 10.1038/s41586-024-07487-w.

Willow Ahrens, James Demmel, and Hong Diep Nguyen. Algorithms for efficient reproducible floating point summation. *ACM Trans. Math. Softw.*, 46(3):22:1–22:49, 2020. doi: 10.1145/3389360. URL https://doi.org/10.1145/3389360.

Liam Atkinson. TriFast: Fused triangle self-attention kernels in Triton. Version 0.1.13 software, 2025. URL https://github.com/latkins/trifast.

Sankar Basu and Björn Wallner. DockQ: A quality measure for protein-protein docking models. *PLOS ONE*, 11(8):e0161879, 2016. doi: 10.1371/journal.pone.0161879. URL https://doi.org/10.1371/journal.pone.0161879.

ByteDance AML AI4Science Team, Xinshi Chen, Yuxuan Zhang, Chan Lu, Wenzhi Ma, Jiaqi Guan, Chengyue Gong, Jincai Yang, Hanyu Zhang, Ke Zhang, Shenghao Wu, Kuangqi Zhou, Yanping Yang, Zhenyu Liu, Lan Wang, Bo Shi, Shaochen Shi, and Wenzhi Xiao. Protenix – advancing structure prediction through a comprehensive AlphaFold3 reproduction. *bioRxiv*, 2025. doi: 10.1101/2025.01.08.631967. URL https://www.biorxiv.org/content/early/2025/01/11/2025.01.08.631967.

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

Shenggan Cheng, Xuanlei Zhao, Guangyang Lu, Jiarui Fang, Zhongming Yu, Tian Zheng, Ruidong Wu, Xiwen Zhang, Jian Peng, and Yang You. FastFold: Reducing AlphaFold training time from 11 days to 67 hours. *arXiv preprint arXiv:2203.00854*, 2022. doi: 10.48550/arXiv.2203.00854.

Caroline Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, 2015. doi: 10.1016/j.parco.2015.09.001.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

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 (NeurIPS)*, volume 35, pp. 16344–16359, 2022.

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

Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pp. 69–76, 2019. doi: 10.1109/ARITH.2019.00019. URL https://arxiv.org/abs/1904.06376.

Nicholas J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993. doi: 10.1137/0914050.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021. doi: 10.1038/s41586-021-03819-2.

Hoa La, Ahan Gupta, Alex Morehead, Jianlin Cheng, and Minjia Zhang. MegaFold: Efficient training of next-generation 3D attention protein models on cross-platform GPUs. *arXiv preprint arXiv:2506.20686*, 2025. doi: 10.48550/arXiv.2506.20686. URL https://arxiv.org/abs/2506.20686v2. Version 2, revised 13 June 2026; version 1 appeared as "MegaFold: System-Level Optimizations for Accelerating Protein Structure Prediction Models".

Valerio Mariani, Marco Biasini, Alessandro Barbato, and Torsten Schwede. lDDT: A local superposition-free score for comparing protein structures and models using distance difference tests. *Bioinformatics*, 29(21):2722–2728, 2013. doi: 10.1093/bioinformatics/btt473. URL https://doi.org/10.1093/bioinformatics/btt473.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations (ICLR)*, 2018.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

NVIDIA Corporation. cuEquivariance documentation. Version 0.10.0, released 2026-04-21, 2026. URL https://docs.nvidia.com/cuda/cuequivariance/changelog.html.

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 Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Xinwei Qiang, Hongmin Chen, Shixuan Sun, Jingwen Leng, Xin Liu, and Minyi Guo. DASH: Deterministic attention scheduling for high-throughput reproducible LLM training. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=bMi5ssfPoM.

Zhuoran Qiao, Feizhi Ding, Thomas Dresselhaus, Mia A. Rosenfeld, Xiaotian Han, Owen Lewis Howell, Aniketh Iyengar, Stephen Opalenski, Anders S. Christensen, Sai Krishna Sirumalla, Frederick R. Manby, Thomas F. Miller, III, and Matthew Welborn. NeuralPLexer3: Accurate biomolecular complex structure prediction with flow models. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2412.10743.

Haiquan Qiu and Quanming Yao. Why low-precision transformer training fails: An analysis on flash attention. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=0jHyEKHDyx.

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

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

Shuaiwen Leon Song, Bonnie Kruft, Minjia Zhang, Conglong Li, Shiyang Chen, Chengming Zhang, Masahiro Tanaka, Xiaoxia Wu, Jeff Rasley, Ammar Ahmad Awan, et al. DeepSpeed4Science initiative: Enabling large-scale scientific discovery through sophisticated AI system technologies. *arXiv preprint arXiv:2310.04610*, 2023. doi: 10.48550/arXiv.2310.04610. URL https://arxiv.org/abs/2310.04610.

Cecilia Summers and Michael J. Dinneen. Nondeterminism and instability in neural network optimization. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pp. 9913–9922, 2021. URL https://proceedings.mlr.press/v139/summers21a.html.

The OpenFold3 Team. OpenFold3-preview. Software archive, version 0.3.1-zenodo, 2025-10-30, 2025. URL https://doi.org/10.5281/zenodo.17485510.

The OpenFold3 Team. OpenFold3-preview2 technical report. Technical report, OpenFold Consortium, 2026. URL https://portal.openfold.omsf.io/reports/of3p2_technical_report.pdf.

Haixu Wu, Minghao Guo, Yuezhou Ma, Yuanxu Sun, Jianmin Wang, Wojciech Matusik, and Mingsheng Long. FlashBias: Fast computation of attention with bias. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/1dc3d70df51a218497529df998a8a8ce-Abstract-Conference.html.

Tao Yu, Gaurav Gupta, Karthick Gopalswamy, Amith R. Mamidala, Hao Zhou, Jeffrey Huynh, Youngsuk Park, Ron Diamant, Anoop Deoras, and Luke Huan. Collage: Light-weight low-precision strategy for LLM training. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 57459–57479. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/yu24d.html.

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.

Adam Zemla. LGA: A method for finding 3D similarities in protein structures. *Nucleic Acids Research*, 31(13):3370–3374, 2003. doi: 10.1093/nar/gkg571. URL https://doi.org/10.1093/nar/gkg571.

Peiyuan Zhang, Matthew Noto, Wenxuan Tan, Chengquan Jiang, Will Lin, Wei Zhou, and Hao Zhang. Attn-QAT: 4-bit attention with quantization-aware training. *arXiv preprint arXiv:2603.00040*, 2026. URL https://arxiv.org/abs/2603.00040.

Feiwen Zhu, Arkadiusz Nowaczynski, Rundong Li, Jie Xin, Yifei Song, Michal Marcinkiewicz, Sukru Burc Eryilmaz, Jun Yang, and Michael Andersch. ScaleFold: Reducing AlphaFold initial training time to 10 hours. In Vivek De (ed.), *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*, pp. 265:1–265:6. ACM, 2024. doi: 10.1145/3649329.3657326. URL https://doi.org/10.1145/3649329.3657326.

APPENDIX

This appendix gives the technical details, proofs, and additional experiments that support the main text. It is organized as follows. Appendix A lists the notation and the kernel implementation parameters. Appendix B describes how biomolecular inputs are tokenized and arranged into pair tensors, and explains the shared pair bias and its gradient. Appendix C covers the forward pipeline: online softmax, TMA staging, Tensor Core execution, and output residual storage. Appendix D derives the backward gradient identities, describes the key tile partitioning and ordered accumulation, and proves that the result does not depend on the workspace schedule. It also bounds the error of the output residual compensation. Appendix E analyzes workspace allocation, asymptotic complexity, and peak training memory. Appendix F compares our kernels with existing attention kernels and their recomputation and accumulation strategies. Appendix G gives the experimental setup and additional end-to-end results, including step latency, paired statistics, training loss, numerical accuracy, deterministic replay, and sixteen-GPU fine-tuning validation. Appendix H reports single-GPU operator measurements for triangular attention and triangle multiplication, plus a forward dataflow ablation. Appendix I explains which gradients need a cross-block reduction, compares ways to make such reductions deterministic, and measures their cost. Appendix J presents the design of the triangle multiplication kernel, a component ablation, and a precision analysis.

A NOTATION AND IMPLEMENTATION PARAMETERS

Table 1 defines the variables used in the method and appendix. A colon in a tensor subscript selects all entries along that axis, and $\top$ denotes transpose. Parenthesized superscripts identify key tiles or outer groups rather than powers.

Table 1: Symbols used in the method and appendix. Tensor shapes are logical; batch and head indices are omitted where stated.

Notation	Meaning
B, N, N_o	Examples per GPU, tokens per crop, and number of fixed-token attention slices. The label o in N_o denotes the outer axis; $N_o = N$.
H, D, C_z	Heads, features per head, and features per pair. D_{head} is another name for D; the labels head and z refer to a head and pair tensor Z.
o, q, k; i, j	Fixed-token, query, and key indices. The triangle example uses $i = o$ and $j = q$. A subscript o alone selects one fixed-token slice.
d; t, ℓ, c, w	Feature coordinate; key tile, outer group, forward 16-key chunk, and backward warpgroup indices. There are four chunks per forward tile and two backward warpgroups. In memory analysis, t denotes an execution point.
A_1, A_2, A_3; B_1, B_2	Residue tokens in chains A and B; the number is the position within that chain. Chain labels are distinct from batch size B and memory function $A(t)$.
Z	Pair features of shape $[B, N, N, C_z]$. $Z_{ij:}$ selects one pair; $Z_{AA}, Z_{AB}, Z_{BA}, Z_{BB}$ select chain blocks. Token subscripts such as $Z_{A_2,:}$ or Z_{A_2,A_1} select a fixed-token slice or an individual pair.
Q, K, V, O	Query, key, value, and attention output tensors, each of shape $[B, N_o, H, N, D]$.

Notation	Meaning
bias, M	Shared pair bias of shape $[B, H, N, N]$ and additive mask. M_{oqk} selects a score mask; the key-only form is M_{ik}.
T, S, P	Unscaled query–key scores, scores after scaling and adding bias and mask, and softmax probabilities. Their tensor indices follow o, q, k.
X, dX	Any tensor and its loss gradient, including dZ. The prefix d means gradient; a subscript d means feature coordinate. Local dK accumulators are unscaled until the final store.
LSE, δ	Saved log-sum-exp and softmax backward correction for each row. Subscript oq selects query row q in fixed-token slice o.
τ_q, τ_k	Query and key tile sizes. Superscript f denotes forward: $\tau_q^{\mathrm{f}} = \tau_k^{\mathrm{f}} = 64$. Backward uses $\tau_q = 32$, $\tau_k = 64$. Here q, k label the tile axes.
$\mathcal{K}_t$, J_K, J_Q	Keys in backward tile t, and numbers of backward key and query tiles; the labels K and Q refer to keys and queries. $\mathcal{K}_t^{\mathrm{f}}$ is the corresponding forward tile.
$\mathcal{L}_\ell$, G, L	Fixed tokens in outer group ℓ, tokens per group ($G = 4$), and number of groups. Groups need not follow chain boundaries.
o_0, o_1, o_2, o_3	Four fixed-token positions within one outer group. $dS_{o_0}, \ldots, dS_{o_3}$ are their contributions to the same bias element; invalid positions contribute zero.
B_c, Q_s, R	Examples per batch chunk, query rows per query chunk, and outer groups per launch. $Q_{\mathrm{span}} = Q_s$. Here c and s label the batch-chunk and query-chunk sizes, not the chunk index c used elsewhere.
$dQ_{\mathrm{part}}^{(t)}$	Unscaled query-gradient partial from key tile t; part means partial. Added subscripts oq : select its output row. The identities use exact partials; algorithms use their stored BF16 values.
$d\,\mathrm{bias}_{\mathrm{part}}^{(\ell)}$	Bias-gradient partial from outer group ℓ. Added subscripts qk select one output element. The identities use exact partials; algorithms use their stored BF16 values.
$\widehat{d\,\mathrm{bias}_{\mathrm{part}}^{(\ell)}}$, $d\,\mathrm{bias}_{\mathrm{acc}}^{(\ell)}$	The hat marks a stored BF16 bias partial. The label acc marks the FP32 sum through group ℓ; state 0 already contains group 0's partial.
μ, λ	Running row maximum and softmax normalizer. Subscripts oq identify the row; a prime marks the value after the current key tile.
ρ, acc	Row rescaling factor and unnormalized FP32 output sum. Subscripts oq or oq : select a row scalar or vector; a prime marks the updated value.
s, row	Shared-memory 16-byte chunk and row indices used for address swizzling. This s is distinct from the query-chunk label in Q_s.
O_{32}, O_{16}, E_{16}	FP32 output before conversion, public BF16 output, and saved BF16 approximation of the output-rounding residual. Subscripts 32 and 16 label precision, not tensor coordinates; 16 denotes BF16.
$\widetilde{O}_{32}$, O^*	Output reconstructed from O_{16} and E_{16}, and exact attention output. The tilde marks reconstruction; the star marks the exact result.
x, y, e; $\widehat{e}, \widetilde{x}$	One FP32 output entry, its BF16 rounding, and the discarded residual. The hat marks the rounded BF16 residual; the tilde marks the reconstructed entry.
u_{16}, u_{32}	BF16 and FP32 unit roundoffs under the stated normal-range assumptions; the subscripts label precision.
RN_{16}, RN_{32}, $\mathrm{FP32}(\cdot)$, $\sum^{\mathrm{FP32}}$	Round-to-nearest conversion to BF16 and to FP32, exact conversion of a BF16 value to FP32, and a sum evaluated in FP32.
$\circ$, $\mathrm{softmax}_k$, diag, rowsum, rowmax, $X^\top$	Elementwise product, softmax over keys, the diagonal matrix of a vector, the sum and the maximum over each row, and the transpose. S_c, $V_{o,c}$, and P_c restrict a tile to the keys of forward chunk c.
$\mathbb{R}$	Real numbers; for example, one score tile lies in $\mathbb{R}^{\tau_q \times \tau_k}$.
$\Theta(\cdot)$, $\|\cdot\|_1$, $\|\cdot\|_2$	Asymptotic order, and the sum of absolute values and the Euclidean norm over all entries. The relative L_2 error of a tensor X against a reference X' is $\|X - X'\|_2 / \|X'\|_2$.
$\widehat{\delta}$, η	Approximate row correction (marked by the hat) and its difference from the reference correction. η_{oq} is one row's difference; η_o collects the differences for slice o.

Notation	Meaning
ΔdX	Change in a gradient when only the row correction changes; Δ marks the difference.
W_Q	Bytes for all BF16 query-gradient partials; Q labels the query gradient.
$W_{\mathrm{bias,all}}$, $W_{\mathrm{bias,res}}$	Bytes for all BF16 bias partials or only the resident subset. The labels all and res mean all and resident.
W_{KV}	Bytes for FP32 key/value gradient continuation buffers; KV labels the two gradients. Zero when these buffers are unnecessary.
W_E	Bytes for one attention call's BF16 output residual; E labels the residual tensor.
W_{dS}	Bytes for a complete FP32 score gradient; dS labels that tensor.
$A(t)$, $W(t)$	Bytes occupied by other live tensors and by temporary operator workspace at execution point t.
P_0, P_1	Peak memory without and with the added workspace. Subscripts $0, 1$ label these cases; these are not attention probabilities.
p	Two-sided sign-test p-value.

Table 2 gives the CUDA launch and tile settings for triangular attention and the 128-channel triangle multiplication implementation. A block is a cooperative thread array (CTA); each warp contains 32 threads. An attention row is one query in a fixed-token slice, whereas a pair row in triangle multiplication is one token pair from the BN^2 pairs.

Table 2: Kernel settings. Tiles describe the work per block iteration, and thread, warp, and warpgroup counts are per block. WGMMA groups count explicit groups of four cooperating warps; a dash means that the kernel does not use WGMMA. TriMul entries use 128 input and hidden channels.

Kernel	Tile or work per block	Threads	Warps	WGMMA groups
Attention forward	64 queries $\times$64 keys; one fixed token and head	128	4	1
Softmax correction	8 attention rows; one row per warp	256	8	—
Attention backward main kernel	32 queries $\times$64 keys $\times$4 fixed tokens; two tokens per warpgroup	256	8	2
Query-gradient reduction kernel	Up to 2,048 output elements; 8 per thread	256	8	—
Pair-bias reduction kernel	Up to 2,048 output elements; up to 8 adjacent keys per thread	256	8	—
TriMul input fusion	128 pair rows $\times$160 projection columns (5×32 channels)	256	8	—
TriMul output fusion	128 pair rows $\times$128 output channels	256	8	—
TriMul projection recomputation and gate backward	128 pair rows $\times$128 projection columns (4×32 channels)	256	8	—
TriMul output-gate backward	Up to 256 tensor elements	256	8	—
TriMul output normalization backward and transpose	32 pair rows $\times$128 channels	256	8	—
TriMul input normalization backward	8 pair rows $\times$128 channels	256	8	—
TriMul normalization parameter reductions	128 channels; one channel per thread	128	4	—
TriMul weight-gradient reduction	Up to 256 weight elements; one element per thread	256	8	—
cuBLAS matrix products	Tiles and launch settings selected by cuBLAS			

The attention forward score instruction is WGMMA $64 \times 64 \times 16$; backward uses WGMMA $64 \times 32 \times 16$ for the transposed score and probability-gradient products. Value aggregation and gradient contractions use warp-level MMA $16 \times 8 \times 16$. TriMul projection kernels use the same

warp-level instruction, with warp tiles of 32×80 for input fusion and 32×64 for output fusion and backward recomputation. All these matrix instructions use BF16 operands and FP32 accumulation.

The attention forward launch contains $BN_oH\lceil N/64 \rceil$ blocks. Each launch of the main backward kernel contains $B_cHR\lceil N/64 \rceil$ blocks for a full batch chunk and range of outer groups; boundary launches contain only their active entries. The thread block traverses the query rows assigned to that launch in 32-row tiles. TriMul input fusion and projection recomputation use four channel slices and a persistent row grid: each slice launches at most one quarter of twice the device's SM count, rounded down, and no more blocks than there are 128-row tiles. Output fusion launches at most twice the SM count, also limited by the number of row tiles. Blocks loop over remaining tiles. These launch limits specify the grid, not the number of blocks simultaneously resident on an SM.

For TriMul, input-normalization parameter partials cover 64–2,048 pair rows in multiples of 64; output-normalization parameter partials are reduced through 64 groups before the final reduction. Five-projection weight gradients use 16,384 pair rows per chunk and a 48 MiB limit for FP32 partial matrices. Their 640×128 output is reduced by 320 blocks. Input fusion uses 103.5 KiB of dynamic shared memory per block; output fusion and projection recomputation each use 103 KiB. The corresponding attention buffers and workspace scheduling are detailed below.

B Biomolecular Inputs and Tensor Organization

B.1 Tokenization and Chain Identity

A training example represents a biomolecular complex containing one or more chains. AlphaFold3 assigns one token to each standard polymer residue and one to each atom of other components (Abramson et al., 2024). A standard amino acid therefore contributes one token, while token count, chain count, and atom count describe different aspects of the same example.

For example, consider three standard residues in chain A and two in chain B:

$$\underbrace{A_1, A_2, A_3}_{\text{chain A}}, \qquad \underbrace{B_1, B_2}_{\text{chain B}}.$$

Indexing the selected tokens as $(A_1, A_2, A_3, B_1, B_2)$ gives $N = 5$ positions across two chains. Chain identity and residue position are retained as separate features, so adjacency between A_3 and B_1 in the array does not introduce a peptide bond. Cropping selects tokens from the complex; a spatial crop can contain noncontiguous residues from multiple chains.

The data pipeline supplies token, chain, and atom features together with masks. Model embedding and subsequent updates produce the pair representation on which triangular attention operates. The outer slices described below are views of this pair tensor.

B.2 Pair Representations and the Outer Index

For one example, the pair representation is $Z \in \mathbb{R}^{N \times N \times C_z}$, where $Z_{ij:}$ is a learned feature vector for ordered token pair (i, j). These features encode pairwise information rather than explicit coordinates or chemical bonds. In the two-chain example, the array partitions into

$$Z = \begin{pmatrix} Z_{AA} & Z_{AB} \\ Z_{BA} & Z_{BB} \end{pmatrix}, \qquad \begin{array}{ll} Z_{AA} : 3 \times 3 \times C_z, & Z_{AB} : 3 \times 2 \times C_z, \\ Z_{BA} : 2 \times 3 \times C_z, & Z_{BB} : 2 \times 2 \times C_z. \end{array}$$

The diagonal blocks contain pairs within each chain, and the off-diagonal blocks contain pairs across chains. Attention can use both kinds of pair; eligibility is determined by the masks.

Starting-node triangular attention fixes the first pair index. Setting $o = A_2$, for example, selects the feature sequence

$$Z_{A_2,:} = \big(Z_{A_2,A_1}, Z_{A_2,A_2}, Z_{A_2,A_3}, Z_{A_2,B_1}, Z_{A_2,B_2}\big).$$

Fixing the first index defines the *outer index* o. The selected slice contains five pair-feature vectors of dimension C_z, and attention operates along its second index. Thus the example has five outer slices, one per token. Each slice defines a logical 5×5 score matrix per head, evaluated in tiles.

Ending-node attention fixes the second pair index and attends along the first. The same attention core applies after permuting the pair axes and arranging the bias accordingly; the output permutation

then restores the original layout (The OpenFold3 Team, 2025). In both orientations, an outer slice is defined by a fixed pair-tensor endpoint, independently of chain boundaries or three-dimensional geometric planes.

B.3 Shared Pair Bias and Its Gradient

For starting-node attention, suppress batch and head indices and let the projections act on normalized pair features. A score for updating pair (i, j) using pair (i, k) has the form

$$S_{ijk} = \frac{1}{\sqrt{D}} \langle Q_{ij:}, K_{ik:} \rangle + \mathrm{bias}_{jk} + M_{ik}. \tag{B.1}$$

The score connects the pair relations (i, j), (i, k), and (j, k), which form the triangle. The tokens can belong to the same chain or to different chains; the construction uses learned pair relations rather than requiring covalent connectivity.

Here i is the outer index and j, k are the query and key indices. The same bias_{jk} is used for every i. Consequently, by the chain rule,

$$d\,\mathrm{bias}_{jk} = \sum_i dS_{ijk}. \tag{B.2}$$

The forward pass evaluates outer slices independently, whereas the shared bias gradient couples them through this sum. The attention core returns gradients for Q, K, V, and bias, which then propagate through the projections and normalization to the pair representation and model parameters.

C Forward Pipeline and Hardware Mapping

Figure 6 follows one query tile from global memory through shared-memory staging and computation to the saved output, and Algorithm 2 lists the steps of one thread block. The buffer sets exchange roles as the block traverses keys.

C.1 Online Softmax and Chunked Value Aggregation

The forward kernel evaluates Equation (B.1) with tiled online softmax (Milakov & Gimelshein, 2018). Like earlier memory-efficient attention methods (Rabe & Staats, 2021; Dao et al., 2022), it avoids storing the full score and probability matrices. One thread block contains a single warpgroup of four warps, each covering 16 query rows. It owns a query block of $\tau_q^{\mathrm{f}} = 64$ rows for one outer index and head, and traverses key blocks of $\tau_k^{\mathrm{f}} = 64$ keys in ascending order. The launch grid is (query blocks, heads, outer tasks). Each score tile is computed by one matrix-multiply chain, then consumed in ascending 16-key chunks indexed by c.

Suppose the block has consumed key blocks $0, \ldots, t-1$ and carries, for each query row, a running maximum μ_{oq}, a running normalizer λ_{oq}, and an unnormalized FP32 output accumulator $\mathrm{acc}_{oq:}$. Writing $\mathcal{K}_t^{\mathrm{f}}$ for the keys of forward block t, the update is

$$\mu'_{oq} = \max\Big(\mu_{oq}, \max_{k \in \mathcal{K}_t^{\mathrm{f}}} S_{oqk}\Big), \qquad \rho_{oq} = \exp\big(\mu_{oq} - \mu'_{oq}\big), \tag{C.1}$$

$$\lambda'_{oq} = \rho_{oq} \lambda_{oq} + \sum_{k \in \mathcal{K}_t^{\mathrm{f}}} \exp\big(S_{oqk} - \mu'_{oq}\big), \tag{C.2}$$

$$\mathrm{acc}'_{oq:} = \rho_{oq}\, \mathrm{acc}_{oq:} + \sum_{k \in \mathcal{K}_t^{\mathrm{f}}} \exp\big(S_{oqk} - \mu'_{oq}\big) V_{ok:}, \tag{C.3}$$

and after the last key block the block emits

$$O_{32,oq:} = \frac{\mathrm{acc}_{oq:}}{\lambda_{oq}}, \qquad \mathrm{LSE}_{oq} = \mu_{oq} + \log \lambda_{oq}. \tag{C.4}$$

These recurrences use per-row and per-score guards: if the previous maximum is $-\infty$, set $\rho = 0$; a score of $-\infty$ contributes zero probability; and if the final normalizer is zero, set $O_{32} = 0$ and $\mathrm{LSE} = -\infty$. These conventions handle initial masked key tiles and fully masked rows without

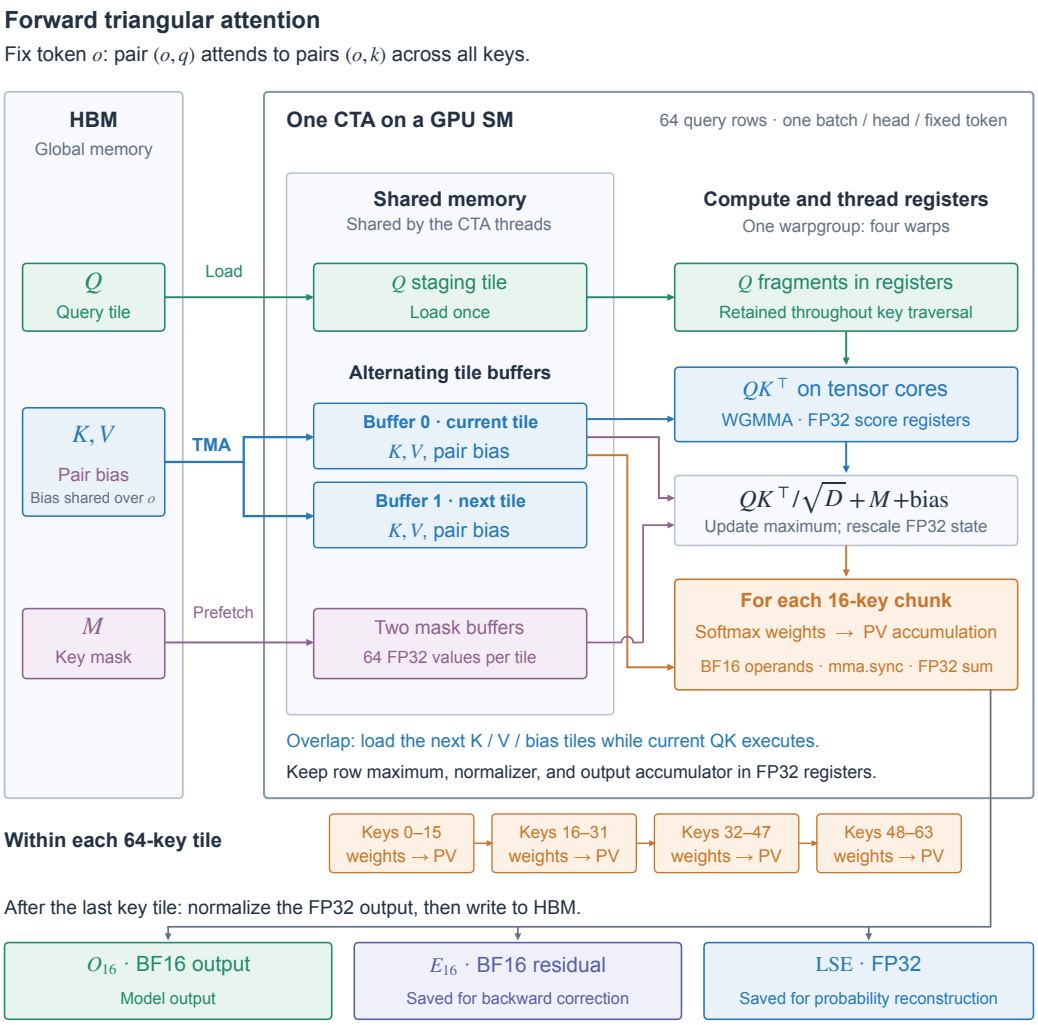


Figure 6: TurboPairFormer triangular attention forward pipeline. One CTA processes 64 query rows for a fixed token (outer index), batch element, and head. The illustrated TMA path stages keys, values, and pair bias in alternating shared memory buffers; query fragments remain in registers, and ordinary global loads prefetch the key mask. Transfers for the next key tile overlap the current WGMMA score computation. Within each tile, softmax weight evaluation and value aggregation alternate over four consecutive 16-key chunks. FP32 running state is retained in registers until final normalization. The epilogue writes the BF16 output, its BF16 rounding residual, and FP32 log-sum-exp. Buffer 0 and buffer 1 exchange roles on successive key tiles; residual storage is shown enabled.

evaluating $-\infty - (-\infty)$ or $0/0$. Unmasked scores are assumed finite. The forward pass saves LSE for backward recomputation, along with the output and optional residual described in Section C.4. Score and probability tiles remain on chip.

Probability evaluation is interleaved with value aggregation: exponentials are computed immediately before their 16-key chunk is consumed. Each thread reads the bias entries of its score fragment directly from the shared-memory bias tile, in the register layout of the score accumulator. Per-score $-\infty$ guards assign zero probability to masked entries. Ascending chunk and key-block traversal fixes the accumulation order in Equation (C.3).

C.2 TMA Staging and Shared Memory Layout

The kernel instantiates $D \in \{16, 32, 64\}$ and maps supported requests to the smallest instance that accommodates their head dimension. The reported OpenFold3 triangular-attention configuration uses $D = 32$. Each instance uses the same Tensor Memory Accelerator (TMA) pipeline.

TMA transfers three-dimensional K and V tiles from global to shared memory. Compatible pair-bias layouts use the same mechanism; the implementation retains ordinary loads for other bias layouts. Each TMA-staged tensor has two buffers and a separate array of asynchronous transaction barriers (`mbarrier`). Transfers for key block $t + 1$ are issued while the kernel consumes block t. Figure 6 shows the path with TMA bias staging enabled.

The shared-memory layout swizzles 16-byte chunks within 128-byte rows: chunk s is stored at position $s \oplus (\text{row} \bmod 8)$, with the swizzle span selected from D at compile time. The key mask contributes one FP32 value per key for the current outer index. Ordinary global loads prefetch its next segment into a second buffer while the current segment is consumed.

A reference path stages the same K, V tiles with two-stage `cp.async` transfers in 16-byte chunks, using commit-group and wait-group completion. It retains the tile extents, traversal order, and online-softmax recurrence, providing a direct comparison for the TMA path.

C.3 Tensor Core Execution

The score contraction $QK^\top$ uses Warpgroup Matrix Multiply-Accumulate (WGMMA) instructions of shape `m64n64k16`, with BF16 operands and FP32 accumulation. The first instruction initializes the accumulator and subsequent instructions accumulate along the head dimension, with warpgroup fence, commit, and wait operations controlling completion. Value aggregation PV uses warp-level `mma.sync` instructions of shape `m16n8k16`. Transposing shared-memory loads incorporate the swizzle into operand addressing; both contractions retain FP32 accumulators.

For fixed inputs, binary, and device, the TMA and reference paths produce bitwise-identical results in the reported equivalence checks. The WGMMA chain reproduces the reference path's chained matrix product, and the swizzled loads recover the same operand bytes. This path-level comparison is repeated after changes to either implementation.

C.4 Output Residual Storage

At the output conversion point the block holds $O_{32} = \text{acc}/\lambda$ in FP32 and writes

$$O_{16} = \text{RN}_{16}(O_{32}), \qquad E_{16} = \text{RN}_{16}\big(O_{32} - O_{16}\big), \tag{C.5}$$

The residual subtraction is evaluated in FP32 after widening O_{16}. The public output is always O_{16}, and the kernel writes E_{16} only when a residual buffer is supplied. Section D.5 analyzes this two-component representation through the scalar construction $y, e, \widehat{e}$, with $\widetilde{x} = O_{16} + E_{16}$ per output element.

C.5 Execution Conditions

The arithmetic configuration comprises τ_q^{f}, τ_k^{f}, the 16-key chunk width, traversal order, instantiated head dimension, and the conversion points for O_{16} and E_{16}. Workspace schedule invariance in Proposition D.2 holds these choices and the instruction semantics fixed. The TMA/reference substitution changes the instruction sequence and is therefore covered by the direct equivalence check above.

Algorithm 2 TURBOPAIRFORMER forward pass, thread-block view

Require: $Q, K, V \in \mathbb{R}^{B\times N_o\times H\times N\times D}$, $\mathrm{bias} \in \mathbb{R}^{B\times H\times N\times N}$, M in HBM; one query tile of τ_q^{f} rows in slice o; key tiles $\mathcal{K}_t^{\mathrm{f}}$ of τ_k^{f} keys.

Stage the query tile once

Load Q_o from HBM to shared memory, then to registers.
Prefetch K_o, V_o, M_o, and the bias tile of key tile 0 into shared-memory buffer 0.
On chip, set $\mu = -\infty$, $\lambda = 0$, and $\mathrm{acc} = 0 \in \mathbb{R}^{\tau_q^{\mathrm{f}}\times D}$ in FP32.

Visit the key tiles

for each key tile t, in ascending order **do**
  Wait for K_o of tile t; on chip, compute $T = Q_o K_o^\top$ (WGMMA).
  Prefetch K_o, V_o, M_o, and the bias tile of key tile $t+1$ into the other buffer.
  Wait for the bias tile of tile t; on chip, compute $S = T/\sqrt{D} + \mathrm{bias} + M_o$.
  On chip, compute $\mu' = \max(\mu, \mathrm{rowmax}(S))$.
  Per row, set $\rho = 0$ if $\mu = -\infty$, otherwise $\rho = \exp(\mu - \mu')$.
  On chip, set $\lambda \leftarrow \rho\lambda$, $\mathrm{acc} \leftarrow \rho\,\mathrm{acc}$, and $\mu \leftarrow \mu'$.
  Wait for V_o of tile t.
  for each 16-key chunk c, in ascending order **do**
    Set each P_c entry to zero if its score is $-\infty$, otherwise to $\exp(S_c - \mu)$.
    On chip, compute $\mathrm{acc} \leftarrow \mathrm{acc} + P_c V_{o,c}$ (mma.sync).
  end for
  On chip, compute $\lambda \leftarrow \lambda + \mathrm{rowsum}(P)$.
end for

Write the outputs

Per row with $\lambda > 0$, compute $O_{32} = \mathrm{acc}/\lambda$ and $\mathrm{LSE} = \mu + \log\lambda$.
For rows with $\lambda = 0$, set $O_{32} = 0$ and $\mathrm{LSE} = -\infty$.
On chip, round O_{32} to O_{16} in BF16, and round the residual $O_{32} - O_{16}$ to E_{16} in BF16.
Write O_{16}, E_{16}, and LSE to HBM.

D BACKWARD IDENTITIES AND ORDERED ACCUMULATION

D.1 GRADIENT IDENTITIES

Suppressing batch and head indices, the attention core is

$$S_{oqk} = \frac{1}{\sqrt{D}}\langle Q_{oq:}, K_{ok:}\rangle + \mathrm{bias}_{qk} + M_{oqk}, \tag{D.1}$$

$$P_{oq:} = \mathrm{softmax}_k(S_{oq:}), \qquad O_{oq:} = \sum_k P_{oqk} V_{ok:}. \tag{D.2}$$

For a scalar loss and incoming derivative dO, the backward identities are, in exact arithmetic,

$$dP_{oqk} = \langle dO_{oq:}, V_{ok:}\rangle, \tag{D.3}$$

$$\delta_{oq} = \sum_k P_{oqk} dP_{oqk} = \langle dO_{oq:}, O_{oq:}\rangle, \tag{D.4}$$

$$dS_{oqk} = P_{oqk}(dP_{oqk} - \delta_{oq}), \tag{D.5}$$

$$dQ_{oq:} = \frac{1}{\sqrt{D}}\sum_k dS_{oqk} K_{ok:}, \qquad dK_{ok:} = \frac{1}{\sqrt{D}}\sum_q dS_{oqk} Q_{oq:}, \tag{D.6}$$

$$dV_{ok:} = \sum_q P_{oqk} dO_{oq:}, \qquad d\,\mathrm{bias}_{qk} = \sum_o dS_{oqk}. \tag{D.7}$$

These identities apply to rows with at least one valid key. For fully masked rows, adopting the zero-output convention gives zero attention gradients.

Derivation. Differentiating $O = PV$ gives $dP = dO\,V^\top$ and $dV = P^\top dO$. The softmax Jacobian in each row is $\mathrm{diag}(P) - PP^\top$, yielding Equation (D.5). Differentiating the bilinear score gives Equation (D.6); differentiating the broadcast bias gives the sum over o in Equation (D.7). Finally, exchanging the finite sums over k and feature index d gives

$$\sum_k P_{oqk}\sum_d dO_{oqd} V_{okd} = \sum_d dO_{oqd}\sum_k P_{oqk} V_{okd} = \langle dO_{oq:}, O_{oq:}\rangle.$$

This last identity makes the row correction available before a key-tile traversal when the forward output is retained.

The three reduction axes determine the backward ownership problem: dQ sums over keys, dK, dV sum over queries, and $d\,\mathrm{bias}$ sums over outer indices. The next construction shares the recomputed intermediates while assigning each partial a unique writer.

D.2 Key Tile Ownership and Gradient Partitioning

Partition keys into disjoint tiles $\mathcal{K}_t$ and outer indices into disjoint outer groups $\mathcal{L}_\ell$. A thread block owns one key tile, one head, and one outer group, then traverses query tiles. Separate dK, dV accumulators retain the contribution for each owned outer index. The remaining reductions are partitioned into the exact partials

$$dQ^{(t)}_{\mathrm{part},oq:} = \sum_{k \in \mathcal{K}_t} dS_{oqk} K_{ok:}, \tag{D.8}$$

$$d\,\mathrm{bias}^{(\ell)}_{\mathrm{part},qk} = \sum_{o \in \mathcal{L}_\ell} dS_{oqk}. \tag{D.9}$$

Proposition D.1 (Partitioned backward)**.** *If every logical (o, q, k) contribution is computed exactly once, then*

$$dQ_{oq:} = \frac{1}{\sqrt{D}} \sum_t dQ^{(t)}_{\mathrm{part},oq:}, \qquad d\,\mathrm{bias}_{qk} = \sum_\ell d\,\mathrm{bias}^{(\ell)}_{\mathrm{part},qk}, \tag{D.10}$$

and the thread blocks that own the key tiles obtain dK, dV by completing their query traversals. The computation can reuse one recomputation of each probability tile for all four gradients without retaining the complete score gradient.

Proof. Substitute the disjoint key and outer partitions into the finite sums in Equations (D.6)–(D.7). Every term belongs to exactly one partial. Once δ is available, a recomputed tile P supplies dV; its associated tile dS supplies dQ, dK, and $d\,\mathrm{bias}$. A thread block that owns a key tile visits all query tiles required by its dK, dV entries. The remaining sums are exactly Equation (D.10). □

Key-tile ownership exposes parallelism while keeping dK, dV local. Thread blocks that own different key tiles contribute to dQ, and different outer groups contribute to the shared bias gradient. Each thread block writes its partials at unique addresses; separate reduction kernels then sum those partials in the prescribed order. This realizes Proposition D.1 with one probability recomputation per tile and no floating-point atomic accumulation.

The implementation stores both kinds of partial in BF16 and accumulates their final reductions in FP32. This defines the floating-point realization of the exact partition identities. Numerical accuracy is evaluated against a reference, while repeatability is determined by the operation order established below.

D.3 Backward Kernel Implementation

Algorithm 1 in the main text summarizes the backward dataflow, including preprocessing and the ordered reductions. Algorithms 3 and 4 expand its main-kernel and reduction-kernel phases into kernel-level steps, and Figure 7 shows where each tensor resides and how it moves between HBM, shared memory, and registers.

Algorithm 3 assigns a 256-thread block with two warpgroups to one key tile, head, and outer group. The grid is (key tiles, heads, outer groups of the launch), and the declared occupancy target is two blocks per multiprocessor. For each outer index and query tile, the block recomputes $\{S^\top, P^\top, dP^\top, dS^\top\}$ and consumes the shared intermediates for all four gradients.

The score and probability-gradient products use WGMMA instructions of shape `m64n32k16`; the remaining contractions use warp-level `mma.sync` instructions of shape `m16n8k16`, with BF16 operands and FP32 accumulation. Shared-memory rows for the bias tree, bias tile, and dS tile are padded from 64 to 65, 72, and 72 elements to avoid bank conflicts. Warpgroup w owns outer

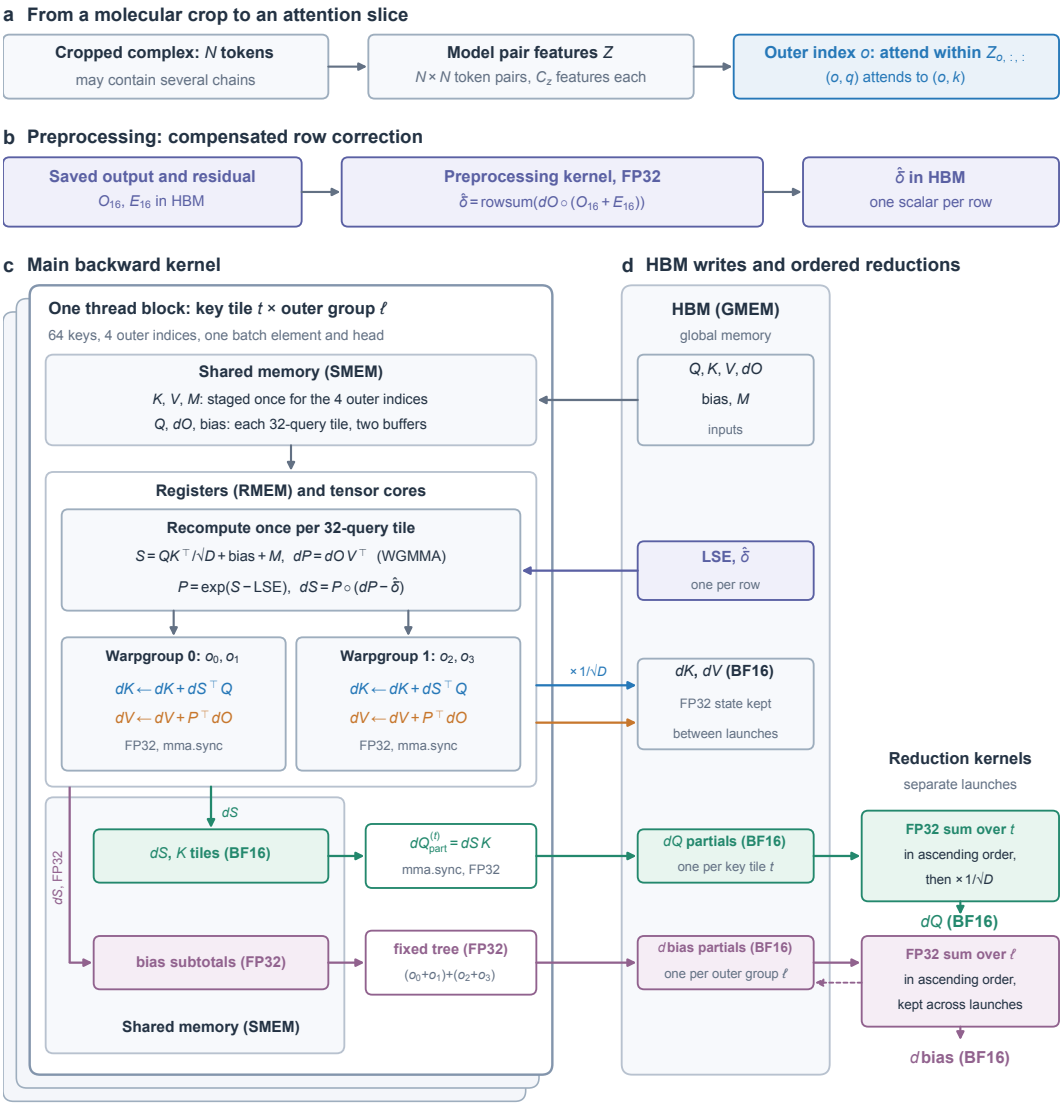


Figure 7: **Backward data movement.** (a) Fixing the outer index o selects the slice $Z_{o,:,:}$, in which the query pair (o, q) attends to the pairs (o, k). (b) A preprocessing kernel computes the compensated row correction $\hat{\delta}$ in FP32 and writes it to HBM. (c) Each thread block of the main backward kernel owns one key tile and one outer group, and further thread blocks run concurrently on other SMs. It stages K, V, and M once and double-buffers Q, dO, and the bias tile in shared memory (SMEM), recomputes S, P, dP, and dS once per query tile, and accumulates dK and dV in FP32 registers (RMEM). The score gradient passes through SMEM to form one query-gradient partial per key tile and, through a fixed FP32 tree, one pair-bias partial per outer group. (d) The thread block writes dK, dV, and its partials to HBM; separate reduction kernels add the partials in ascending index order in FP32, and the pair-bias partial buffers are reused once consumed (dashed arrow).

positions $\{2w, 2w+1\}$, initializing its bias subtotal with the first and adding the second. The dK, dV contractions consume query tiles in a fixed two-chunk order; each warp computes dQ for one outer position and half a query tile.

The score and probability-gradient products use the same operand encoding as the two-pass dK, dV kernel, preserving its accumulation chain. The two warpgroup bias subtotals combine into one uniquely written outer-group partial, and feature tails are zero-padded when K, V, Q, and dO are staged. Probabilities are zero in fully masked rows and at invalid outer indices of a partial outer group,

Algorithm 3 TURBOPAIRFORMER main backward kernel, thread-block view

Require: $Q, K, V, dO, \text{bias}, M, \text{LSE}, \widehat{\delta}$ in HBM; key tile t; outer group ℓ with outer indices $o_0, \ldots, o_3$; the query rows of this launch.

Stage the key tile once

Load K_o, V_o, M_o of key tile t for $o \in \mathcal{L}_\ell$ from HBM to shared memory.
On chip, set $dK_o = dV_o = 0$ in FP32, or reload the FP32 values saved by an earlier launch.
Prefetch Q_o, dO_o and the bias tile of the first query tile into shared-memory buffer 0.

Visit the query tiles

for each query tile, in ascending order **do**
  Wait for this tile's buffer; prefetch the next query tile into the other buffer.
  for each warpgroup w in parallel, and each of its two outer indices o in order **do**
    On chip, compute $S = Q_o K_o^\top / \sqrt{D} + \text{bias} + M_o$ and $dP = dO_o V_o^\top$ (WGMMA).
    On chip, compute $P = \exp(S - \text{LSE})$ and $dS_o = P \circ (dP - \widehat{\delta})$.
    On chip, compute $dV_o \leftarrow dV_o + P^\top dO_o$ and $dK_o \leftarrow dK_o + dS_o^\top Q_o$ (mma.sync).
    Add dS_o to the FP32 bias subtotal of warpgroup w; copy dS_o to shared memory in BF16.
  end for
  Synchronize the thread block.
  Write $d\,\text{bias}^{(\ell)}_{\text{part}} = (dS_{o_0} + dS_{o_1}) + (dS_{o_2} + dS_{o_3})$, the sum of the two subtotals, to HBM in BF16.
  On chip, compute $dQ^{(t)}_{\text{part},o} = dS_o K_o$ for each o (mma.sync); write them to HBM in BF16.
end for

Store the key-tile gradients

if this launch covers the last query rows **then**
  Write $dK_o/\sqrt{D}$ and dV_o to HBM in BF16.
else
  Write dK_o and dV_o to HBM in FP32, to be reloaded by the next launch.
end if

so these entries add nothing to the four gradients. The row scalar $\widehat{\delta}$ is supplied by the preprocessing phase of Algorithm 1; enabling residual compensation changes this scalar while retaining the kernel structure.

Algorithm 4 sums query partials in key-tile order and bias partials in outer-group order. Both reductions use unique output ownership: a thread owns eight consecutive elements of dQ, one aligned 16-byte vector, or up to eight adjacent keys within one row of $d\,\text{bias}$, and row tails and unaligned rows use scalar accesses with the same summation order. Their per-element addition sequences depend on J_K and L, while workspace scheduling determines which partials are resident.

D.4 WORKSPACE SCHEDULE INVARIANCE

Meaning of the scheduling units. Cropping selects the molecular tokens before attention begins. The kernel subsequently partitions computation on this fixed input. A query chunk is a group of Q_s consecutive query rows processed in one launch, with each row still attending to all valid keys through the key tiles. An outer group contains $G = 4$ fixed-token slices. One launch makes the partials from R consecutive outer groups resident together, then permits their buffers to be reused after reduction. A batch chunk selects B_c independent local examples. These units describe GPU work and memory residency; they do not denote protein chains, molecular fragments, or additional data crops.

Let $\widehat{d\,\text{bias}}^{(\ell)}_{\text{part}}$ denote the stored BF16 bias partial for one output element of outer group ℓ. At a common query and key coordinate, its local reduction is

$$\widehat{d\,\text{bias}}^{(\ell)}_{\text{part}} = \text{RN}_{16}(\text{RN}_{32}[\text{RN}_{32}(dS_{o_0} + dS_{o_1}) + \text{RN}_{32}(dS_{o_2} + dS_{o_3})]) \,. \tag{D.11}$$

The two inner additions are the warpgroup subtotals; the remaining FP32 addition combines them before BF16 storage. Invalid outer positions contribute zero. Here RN_{16} and RN_{32} denote round-to-nearest conversion to BF16 and FP32. Fix the input bits, binary, device, arithmetic tiles, and instruction semantics, including this grouping, precision, and parenthesization. Exact BF16-to-FP32

Algorithm 4 TURBOPAIRFORMER reduction kernels

Require: Partials $dQ^{(t)}_{\text{part}}$ for $0 \le t < J_K$, and $d\,\text{bias}^{(\ell)}_{\text{part}}$ for the outer groups of this launch, in HBM.

Query gradient, once after all launches

for each element of dQ, in parallel **do**
  Add $dQ^{(0)}_{\text{part}}, \dots, dQ^{(J_K-1)}_{\text{part}}$ in this order in FP32.
  Scale the sum by $1/\sqrt{D}$ and write it to HBM in BF16.
end for

Pair-bias gradient, after every launch

for each element of $d\,\text{bias}$, in parallel **do**
  Start from the FP32 running sum left by the previous launch, if any.
  for each outer group ℓ of this launch, in ascending order **do**
    Add $d\,\text{bias}^{(\ell)}_{\text{part}}$ to the running sum in FP32.
  end for
  Keep the running sum in HBM for the next launch; after the last launch, write it in BF16.
end for

conversion gives the ordered sum

$$d\,\text{bias}^{(0)}_{\text{acc}} = \text{FP32}(\widehat{d\,\text{bias}}^{(0)}_{\text{part}}), \tag{D.12}$$

$$d\,\text{bias}^{(\ell)}_{\text{acc}} = \text{RN}_{32}\big(d\,\text{bias}^{(\ell-1)}_{\text{acc}} + \text{FP32}(\widehat{d\,\text{bias}}^{(\ell)}_{\text{part}})\big), \quad 1 \le \ell < L. \tag{D.13}$$

The final cast occurs after the last outer group. Across launches, the reduction kernel retains the current FP32 state and resumes at the next outer group. Query partials follow the corresponding fixed key-tile order.

Proposition D.2 (Workspace schedule invariance)**.** *Suppose a physical schedule changes only residency and execution timing, and satisfies all of the following conditions:*

1. *Every logical partial is produced once by the same deterministic floating-point operations and written by one thread block to its own workspace location.*
2. *Each reduction kernel consumes the same ordered partials. Across a launch boundary it stores and reloads the current FP32 running state without conversion or reassociation.*
3. *If query traversal spans several launches, dK, dV retain their FP32 accumulator states and continue the same ordered query-tile operations.*
4. *Stream ordering between the main kernel and the reduction kernels prevents stale reads and buffer reuse before consumption, and the final output casts remain unchanged.*

Then all four output gradients are bitwise invariant across such schedules.

Proof. Each schedule evaluates the same directed acyclic graph of deterministic floating-point operations. Induct over its dependencies: input nodes have the same bits; a node with identical operand bits and the same operation has identical result bits. Unique writers and synchronization preserve those operands. In particular, dividing Equation (D.13) into launches only inserts bit-preserving FP32 stores and loads. Query chunks partition dQ and $d\,\text{bias}$ output coordinates, while their contributions to the same dK, dV entries continue the original FP32 recurrences. Batch chunks act on independent batch coordinates. Thus every output node has the same bits. □

The distinction between rescheduling and regrouping is essential. Under FP32 round-to-nearest with ties to even,

$$\text{RN}_{32}\big(\text{RN}_{32}(2^{24}+1)+1\big) = 2^{24},$$
$$\text{RN}_{32}\big(2^{24} + \text{RN}_{32}(1+1)\big) = 2^{24}+2.$$

The example shows why a launch boundary preserves the current FP32 state instead of introducing a new partial sum: regrouping changes the arithmetic even when both schedules are deterministic. Proposition D.2 permits residency changes with fixed tiles, outer-group membership, partial precision, and instructions. The proposition applies to fixed-input replay of this attention operator with those arithmetic choices held fixed.

D.5 Output Residual Compensation

The row correction enters every score gradient in a query row. Retaining the forward output's rounding residual reduces the output-reconstruction error bound and can improve this shared quantity before probability recomputation. The forward epilogue computes

$$O_{16} = \mathrm{RN}_{16}(O_{32}), \qquad E_{16} = \mathrm{RN}_{16}(\mathrm{RN}_{32}(O_{32} - O_{16})) \,. \tag{D.14}$$

The subtraction widens O_{16} to FP32. Once dO is available, a separate row reduction computes

$$\widehat{\delta}_{oq} = \sum_{d}^{\mathrm{FP32}} dO_{oqd}\ \mathrm{RN}_{32}(O_{16,oqd} + E_{16,oqd}) \,. \tag{D.15}$$

All BF16 operands are widened to FP32; both the reconstruction and the dot product execute in FP32. This scalar is available before the key-tile traversal and requires no probability recomputation. The following proposition isolates its effect on the four gradients.

Proposition D.3 (Propagation of the row correction)**.** *For rows with at least one valid key, hold P, dP, Q, K, V, dO fixed in exact arithmetic and replace δ by $\widehat{\delta} = \delta + \eta$. Then*

$$\Delta dS_{oqk} = -P_{oqk}\eta_{oq}, \qquad \sum_k \Delta dS_{oqk} = -\eta_{oq}, \tag{D.16}$$

$$\Delta dQ_{oq:} = -\frac{1}{\sqrt{D}}\eta_{oq} \sum_k P_{oqk} K_{ok:}, \tag{D.17}$$

$$\Delta dK_{ok:} = -\frac{1}{\sqrt{D}} \sum_q P_{oqk}\eta_{oq} Q_{oq:}, \tag{D.18}$$

$$\Delta d\,\mathrm{bias}_{qk} = -\sum_o P_{oqk}\eta_{oq}, \qquad \Delta dV = 0. \tag{D.19}$$

Proof. Subtract the two instances of Equation (D.5) and substitute the difference into Equations (D.6)–(D.7). The row-sum identity follows from $\sum_k P_{oqk} = 1$. □

The same η_{oq} acts on every key in the row, with $\|\Delta dS_{oq:}\|_1 = |\eta_{oq}|$. Before weighting by P, the broadcast difference $-\eta_o \mathbf{1}^\top$ has rank at most one; the resulting score-gradient difference is $-\operatorname{diag}(\eta_o)P_o$. Its propagation into dQ, dK, and $d\,\mathrm{bias}$ depends on the subsequent contractions and possible cancellation. The proposition therefore identifies the shared precision channel addressed by the residual.

To isolate output conversion, let x be one entry of the normalized FP32 forward result O_{32}, and define

$$y = \mathrm{RN}_{16}(x), \quad e = x - y, \quad \widehat{e} = \mathrm{RN}_{16}(e), \quad \widetilde{x} = y + \widehat{e}.$$

The corresponding elementwise output reconstruction is $\widetilde{O}_{32} = O_{16} + E_{16}$. The sums defining $\widetilde{x}$ and $\widetilde{O}_{32}$ are interpreted over the reals for the following bound. Assume round-to-nearest, no overflow or relevant underflow, and exact evaluation of the residual subtraction. With BF16 unit roundoff $u_{16} = 2^{-8}$,

$$|\widetilde{x} - x| = |\widehat{e} - e| \le u_{16}|e| \le u_{16}^2|x|, \qquad |y - x| \le u_{16}|x|. \tag{D.20}$$

The bound assumes exact residual subtraction and real-valued reconstruction. Rounded subtraction contributes its own term; explicit FP32 reconstruction contributes at most $u_{32}|\widetilde{x}|$, where $u_{32} = 2^{-24}$, under the same normal-range assumptions. For an exactly evaluated row dot product, Equation (D.20) gives

$$\left|\langle dO, \widetilde{O}_{32} - O_{32}\rangle\right| \le u_{16}^2 \sum_d |dO_d (O_{32})_d|.$$

To relate this component to the computed row correction, let $O^* = PV$ be the exact mathematical output and decompose

$$\widehat{\delta} - \langle dO, O^* \rangle = \underbrace{\widehat{\delta} - \langle dO, \widetilde{O}_{32} \rangle}_{\text{evaluation rounding}} + \underbrace{\langle dO, \widetilde{O}_{32} - O_{32} \rangle}_{\text{output conversion}} + \underbrace{\langle dO, O_{32} - O^* \rangle}_{\text{forward arithmetic}}. \tag{D.21}$$

Residual retention reduces the bound on the output-conversion term from first to second order in BF16 unit roundoff under the stated assumptions. This does not guarantee lower actual error in every row correction, because reconstruction errors can cancel in the dot product. Forward arithmetic and evaluation rounding remain separate contributions. The design preserves the public BF16 output and supplies the correction without another probability recomputation, at the cost of one saved BF16 tensor and its associated stores, reads, and correction arithmetic.

E Workspace Complexity and Training Memory

E.1 Workspace Allocation

TurboPairFormer stores BF16 gradient partials to reuse each recomputed tile and to order accumulation. The query partials occupy a full buffer, while the bias partials are produced and consumed in successive launches. Each stored element requires two bytes. The query-partial buffer has logical shape $[J_K, BN_o, H, N, D]$ and size

$$W_Q = 2J_K BN_o HND. \tag{E.1}$$

Storing the bias partials of all outer groups for all query rows would require

$$W_{\text{bias,all}} = 2BLHN^2. \tag{E.2}$$

Instead, the planner keeps B_c batch elements, R outer groups, and Q_s query rows resident at a time, using

$$W_{\text{bias,res}} = 2B_c RHQ_s N. \tag{E.3}$$

Successive launches reuse these buffers after their partials have been consumed. This scheduling controls bias-buffer residency independently of the full query-partial allocation.

Splitting query traversal across launches additionally requires FP32 continuation state for dK and dV. Two full continuation tensors occupy

$$W_{KV} = 8BN_o HND \text{ bytes}. \tag{E.4}$$

The term is zero when no continuation tensors are allocated, as in the selected full-query plans below. If a schedule retains only a subset of outer indices, its continuation cost follows that resident shape.

The saved BF16 output residual occupies $W_E = 2BN_o HND$ bytes per attention call. Each FP32 row tensor, including LSE or δ, occupies $4BN_o HN$ bytes. A full FP32 bias running state occupies $4BHN^2$ bytes, or $4B_c HQ_s N$ for a query-chunk-local state. Total memory accounting also includes alignment, allocator behavior, and tensor lifetimes. Activation checkpointing trades saved activations for forward recomputation (Chen et al., 2016); residuals saved by several attention calls can coexist according to the checkpointing schedule.

E.2 Asymptotic Complexity

The full query-partial buffer remains cubic in crop size. Substituting $N_o = N$ gives

$$W_Q = 2BHDN^2 \left\lceil \frac{N}{\tau_k} \right\rceil = \Theta\left(\frac{BHDN^3}{\tau_k} \right) \quad \text{for fixed arithmetic tile width.} \tag{E.5}$$

For comparison, materializing the FP32 score gradient requires $W_{dS} = 4BHN^3$ bytes. When N is divisible by τ_k,

$$\frac{W_Q}{W_{dS}} = \frac{D}{2\tau_k}. \tag{E.6}$$

Thus $D = 32$ and $\tau_k = 64$ make the query-partial buffer one quarter the size of a full FP32 score gradient. This comparison concerns those two allocations; the operator also retains bias partials and the states described above. Reusing bias buffers across launches reduces resident bias storage, while the output residual and full continuation state scale quadratically with N for fixed B, H, D.

Using the candidate full continuation allocation for W_{KV}, expressing tensor sizes in units of the saved residual, $W_E = 2BHN^2D$, yields

$$\begin{aligned} W_Q &= J_K W_E, \quad W_{KV} = 4W_E, \\ W_{\text{bias,all}} &= \frac{L}{D} W_E, \qquad W_{dS} = \frac{2N}{D} W_E. \end{aligned} \tag{E.7}$$

Table 3 gives the resulting tensor sizes and the resident bias allocations selected by the planner.

Table 3: Analytical allocation sizes in MiB for $B = 1$, $N_o = N$, $H = 4$, $D = 32$, $\tau_k = 64$, $G = 4$, and BF16 partials. The W_{KV} column gives the candidate continuation cost; the selected plans use full query traversals, so their actual continuation allocation is zero. These are tensor sizes rather than measured training peaks. Crop 1024 is an analytical extrapolation.

N	W_Q	$W_{\text{bias,all}}$	$W_{\text{bias,res}}$	W_E	W_{KV}	W_{dS}
384	216	108	108	36	144	864
640	1000	500	500	100	400	4000
768	1728	864	252	144	576	6912
1024	4096	2048	256	256	1024	16384

At crop 768, the selected full-query plan uses $Q_s = 768$ and $R = 56$, giving 252 MiB of resident bias partials alongside 1728 MiB of query partials and a 144 MiB residual per call. An alternative query-chunk plan uses a different, 216 MiB bias allocation. The planning target guides this choice of schedule: it is not a strict allocation cap. For example, the reported 256 MiB target selects a 500 MiB bias buffer at crop 640, and the separately allocated query buffer remains outside that target.

E.3 Peak Training Memory

Whether a transient buffer increases the training peak depends on its lifetime. For a fixed execution schedule, let $A(t)$ denote the bytes occupied by all other live tensors and $W(t)$ the additional transient operator workspace. Define

$$P_0 = \max_t A(t), \qquad P_1 = \max_t \{A(t) + W(t)\}. \tag{E.8}$$

The peak is unchanged exactly when the workspace fits within the available headroom throughout its lifetime:

$$P_1 = P_0 \quad \Longleftrightarrow \quad W(t) \le P_0 - A(t) \text{ for every } t. \tag{E.9}$$

To see this, the right-hand condition is equivalent to $A(t)+W(t) \le P_0$ at every time, while $W(t) \ge 0$ gives $P_1 \ge P_0$. Thus transient backward workspace can fit below a peak established elsewhere in the step by saved activations, pair states, gradients, or optimizer states. Coexisting allocations add, so the relevant quantity is the headroom at each time rather than the size of any one allocation.

E.4 Measured Memory and Limitations

At crop 768, sweeping the bias-workspace planning target from 64 to 512 MiB left the measured peak allocated, reserved, and sampled device memory unchanged. The tested training configurations accommodated the selected schedules, and their observed peak was insensitive to this change in bias-buffer scheduling. Allocated memory, allocator-reserved capacity, and sampled device memory are reported separately because they measure different quantities.

The main memory limitation is the cubic query-partial buffer, which grows with both crop size and local batch size. The sweep holds that buffer and the saved residual fixed, so it measures the effect of bias scheduling rather than the total cost of auxiliary storage. In the headroom argument, persistent residuals belong to $A(t)$; their added cost and the source of the training peak require matched memory measurements and tensor-lifetime traces. The same accounting is needed across backends, whose live allocations can differ. Reducing the cubic query-partial allocation while preserving the reuse of recomputed tiles, concurrency, and fixed arithmetic remains a direction for future work.

F Comparison with Existing Kernels

Triangular attention backward implementations differ in how they share recomputed probabilities and combine gradient contributions across thread blocks. We organize the inspected kernels by three strategies: separate recomputation passes, dense score-gradient storage, and atomic accumulation. Table 4 records these choices for the inspected source versions.[3] The two Triton rows distinguish implementations with different gradient ownership; the versions used for H100 performance and synthetic accuracy/replay are specified in Sections G.1 and G.7, respectively. Section F.2 discusses how TurboPairFormer differs from other kernels that also use one recomputation.

Table 4: Structure of the inspected backward kernels. The three "No" columns indicate the absence of dense FP32 score-gradient storage or floating point atomic accumulation. "Fixed" denotes an accumulation order independent of block retirement under fixed execution settings. Partial storage in TurboPairFormer remains cubic in N. The Triton suffixes are SHA-256 prefixes of `evoformer.py`.

Backend	Auto-tune	P rebuilds	No dense FP32 dS	No dQ atomics	No bias atomics	Fixed order
TurboPairFormer	No	1	Yes	Yes	Yes	Yes
TriFast	Yes	3	Yes	Yes	Yes	Yes
Protenix	Yes	3	Yes	Yes	Yes	Yes
MegaFold	No	2	Yes	Yes	No	No
Triton (`8e0dd74b`)	No	2	Yes	No	Yes	No
Triton (`d704955e`)	No	2	Yes	Yes	No	No
ScaleFold	No	1	No	Yes	Yes	Yes
DeepSpeed	No	1	Yes	Yes	No	No

F.1 Recomputation and Accumulation Strategies

Separate recomputation passes. TriFast computes dQ, (dK, dV), and the pair-bias gradient in three successive kernels. Each recomputes the probabilities from the saved softmax normalization statistics; the first also computes and stores δ for the remaining two. Masked stores give every gradient element one writer, fixing its accumulation order independently of block retirement. Protenix uses the same three-pass structure, with its bias kernel traversing the outer index inside one program before a single store. Neither module uses atomics. Its autotuning candidates match TriFast's base lists. These implementations obtain fixed accumulation orders by assigning each gradient its own traversal, at the cost of recomputing the probabilities three times.

Dense score-gradient storage. The ScaleFold path in Apex OpenFold Triton recomputes probabilities once and writes the complete FP32 score gradient to a tensor of logical shape $[BN_o, H, N, N]$. A subsequent reduction computes the pair-bias gradient from this tensor. The query gradient uses a zero-initialized FP32 buffer, updated by non-atomic loads, additions, and stores in traversal order. This design avoids both repeated recomputation and atomic accumulation, while requiring the $4BHN^3$ bytes of score-gradient storage analyzed as W_{dS} in Section E. Tile sizes, warp counts, and stages are selected from a shape-indexed table with autotuning disabled; the evaluated crops use its default entry.

[3] Fixed source anchors: TriFast `2d525113`, Protenix `4c355be4`, MegaFold `845905e7`, and the newer and earlier OpenFold3 kernels. These identify inspected upstream files; experiment-specific versions are recorded separately.

Atomic accumulation. MegaFold's EvoFlash-3D first preprocesses the row correction, then recomputes probabilities in separate (dK, dV) and fused $(dQ, d\text{bias})$ kernels. Each query, key, and value gradient element has one writer. The bias gradient is accumulated across outer positions within the dQ kernel through the module's only atomic addition, issued with relaxed semantics. Replay measurements reflect this ownership: the output and dQ, dK, dV repeat bitwise, whereas the bias gradient can vary across replays. Its launch block sizes were tuned offline, and `triton.autotune` is commented out. The newer OpenFold3 Triton source (`8e0dd74b`) also recomputes twice, but accumulates bias contributions in registers and stores them once, while dQ uses relaxed atomic additions. The earlier AQLab90e source (`d704955e`), used for the synthetic accuracy and replay results, instead assigns each dQ, dK, dV element one writer and accumulates the shared bias gradient through FP32 atomics. Both paths use two probability recomputations; their atomic accumulation applies to different gradients.

DeepSpeed's DS4Sci Evoformer[4] instead recomputes once in a fused CUTLASS backward kernel. A block owns an outer slice and head, traverses key blocks in order, and accumulates dQ in an FP32 workspace without atomics. Its bias epilogue uses atomic output tile iterators to combine contributions across outer positions. The atomic operation is supplied by the instantiated iterator rather than an explicit atomic call in the surrounding DeepSpeed sources. In these kernels, the gradient accumulated through atomics has an addition order that depends on block execution.

Shared recomputation with ordered partials. TurboPairFormer uses one recomputation for all four gradients (Algorithm 3). Each dK and dV element has one writer; query and bias contributions are written as partials with unique writers and are summed in the prescribed orders of Algorithm 4. Among the inspected paths, it combines one recomputation, no floating point atomic accumulation, and no dense score-gradient tensor. This operating point uses partial-gradient storage: its bias buffers are reused across planner-selected launches, and its query-partial buffer occupies $D/(2\tau_k)$ times the storage of a full FP32 score gradient, or one quarter at the reported configuration (Equation (E.7)). The query-partial buffer remains cubic in N; this ratio compares that buffer alone with dense dS, rather than total auxiliary memory.

F.2 Comparison with Other Single Recomputation Methods

The inspected DeepSpeed DS4Sci Evoformer and ScaleFold/Apex backward kernels already recompute each probability tile once and reuse its intermediates across the four input gradients.[5] Single recomputation is therefore a shared computational strategy. TurboPairFormer differs in the assignment of work across key tiles, the accumulation of the shared pair bias gradient, and the precision of the backward softmax correction.

Work partitioning and parallelism. In both inspected implementations, a thread block or Triton program owns an outer index and head and traverses key tiles sequentially. Query gradient contributions accumulate through an FP32 workspace within that traversal. TurboPairFormer assigns different key tiles to independent thread blocks. Each thread block traverses query tiles, retains its key and value gradient accumulators, and writes query partials with unique writers. This key parallelism requires an explicit reduction of query partials, while all four gradients continue to share the recomputed intermediates.

Shared pair bias accumulation. In these outer-axis partitionings, the shared bias gradient combines contributions from multiple thread blocks. DeepSpeed performs this accumulation through FP32 global atomic additions in its bias epilogue. The inter-block addition order is not prescribed and does not guarantee bitwise replay. The inspected Apex path instead writes the full FP32 score gradient and applies a separate reduction across outer indices. TurboPairFormer combines four outer contributions through a fixed local tree, stores a BF16 bias partial for each outer group, and sums these partials in a prescribed order using FP32 running sums. Query partials are likewise summed in key tile order.

[4] Official DS4Sci documentation: `https://www.deepspeed.ai/tutorials/ds4sci_evoformerattention/`.

[5] The comparison refers to DeepSpeed commit `4a5856c2` and Apex commit `8a6508aa`, including their backward kernels, bias epilogues, and Python wrappers.

These reductions avoid floating point atomics and full score gradient storage. FP32 continuation states preserve this arithmetic across the workspace schedules in Proposition D.2.

Precision of the row correction. The inspected DeepSpeed and Apex paths form their softmax correction from the saved output and upstream gradient, without retaining an auxiliary output rounding residual. TurboPairFormer saves a BF16 approximation of the residual between its FP32 forward result and public BF16 output. Backward uses the output and residual together when evaluating the row correction, reducing the output-reconstruction error bound without another probability recomputation. The controlled residual comparison preserves the public output and dV and lowers the measured dQ, dK, and $d\,\mathrm{bias}$ errors. Section D.5 bounds the output conversion contribution and separates the remaining rounding contributions.

Storage cost and scope of the distinction. Ordered partials require additional memory. As detailed in Section E, the current query partial buffer remains cubic in crop size; for the evaluated crops with $D = 32$ and $\tau_k = 64$, it occupies one quarter of the storage of a full FP32 score gradient. This ratio concerns the query buffer alone, excluding resident bias partials, continuation state, and saved residuals. The distinction is thus a joint design for parallelism, reproducible reduction, and backward precision, with the workspace costs quantified in Section E.

G Experimental Setup and Additional Results

This appendix reports the supplemental eight-H200 training study accompanying the main sixteen-H100 performance experiment, together with additional accuracy results, gradient replay, and fine-tuning validation details. The synthetic tests use absolute RMSE, whereas the captured-call studies use relative L_2 error; their values are reported separately throughout. Triangle multiplication and its component ablation are detailed in Appendix J.

G.1 Sixteen-H100 Performance Protocol

The main performance comparison uses two nodes with eight H100 GPUs each, DDP, and BF16 mixed precision. Each rank processes one input per optimizer step, with no gradient accumulation, giving a global batch size of 16. Each of the eight configurations runs 200 steps per crop, of which steps 21–200 are measured. The host checkout is pinned to `68d1082d8`, with PyTorch 2.10.0, CUDA 12.9, Triton 3.6.0, and cuEquivariance 0.10.0. For this H100 study, PyTorch's FP32 matrix-multiplication precision is set to `high`, enabling TF32; the tensor-core operands of TurboPairFormer's attention kernels remain BF16.

Inputs. The native `WeightedPDBDataset` pipeline uses a common pool of 1,311 entries filtered from the 9,297-entry kinase cache. The static filter requires at least 768 tokens and 768 resolved token-center atoms, with at most 20 chains. All three crop sizes use this pool. Each run initializes the native distributed sampler with seed 42 and uses ten data-loader workers per rank. Cropping and collation occur during the run, and a runtime check rejects any input containing token padding. This protocol is separate from the pre-serialized input replay used in the H200 supplement below; it does not establish bitwise-paired input tensors across H100 runs.

Timing. A callback records CUDA events in `on_train_batch_start` and `on_train_batch_end`. The event interval covers forward and backward GPU computation and the optimizer update. The end event is recorded before the module's batch-end hook, which updates EMA; consequently, EMA updates, pre-batch data loading, and host-to-device transfer are outside the measured interval. The harness synchronizes at the start and end of the measured window, then takes the maximum duration across the 16 ranks for each step and averages the 180 durations. The figure reports these GPU event times, rather than the separately recorded wall-clock times. This timing boundary differs from that of the H200 supplement, whose interval includes EMA.

Backend scope. TurboPairFormer enables both its attention and triangle multiplication kernels. The OpenFold3 Triton configuration enables both Triton operators, whereas Protenix, TriFast, MegaFold, DeepSpeed-Evo, and ScaleFold use cuEquivariance triangle multiplication. The Protenix source is

pinned to `4c355be455`; TriFast 0.1.13 is based on `2d525113` with additive-mask compatibility changes, MegaFold uses `845905e7`, and ScaleFold uses the direct-call Apex path at `8a6508aa`. DeepSpeed-Evo uses the host DS4Sci integration from DeepSpeed 0.19.6. These performance configurations are distinct from the numerical-fix variants identified in the synthetic accuracy and replay studies. The separate stability study also differs: its OpenFold3 Triton attention runs with PyTorch triangle multiplication.

G.2 Hardware and Backend Configuration

For the supplemental performance study, we evaluate OpenFold3 training on eight NVIDIA H200 GPUs using distributed data parallelism (DDP) and BF16 mixed precision at crops 384, 640, and 768. Each configuration runs 400 global optimizer steps: all eight ranks participate in each synchronized parameter update. With one input per rank and no gradient accumulation, the effective global batch size is eight and each run consumes 3,200 rank-local inputs. Steps 21–400 are used for measurement after 20 timing warmup steps. All configurations consume the same pre-serialized batches through the same sequential sampler. CUDA events bracket each training batch on every rank, including forward and backward computation, per-sample gradient clipping, the optimizer, the scheduler, and the exponential moving average. Data loading, host-to-device transfer, and logging lie outside the timed interval. We report the maximum elapsed time across ranks for each step.

The eight configurations use TurboPairFormer, Protenix, cuEquivariance, TriFast, MegaFold, DeepSpeed-Evo, ScaleFold, and AQLab Triton. DeepSpeed-Evo denotes the DS4Sci Evoformer attention integration; AQLab Triton denotes the flag-enabled OpenFold3 Triton backend discussed in Section F. TurboPairFormer and OpenFold3 Triton replace both triangular attention and triangle multiplication. Protenix, TriFast, MegaFold, and ScaleFold replace attention while retaining cuEquivariance triangle multiplication; these four therefore differ from the cuEquivariance configuration in attention alone.

The software stack comprises driver 590.48.01, CUDA 12.9, PyTorch 2.10.0, PyTorch Lightning 2.6.1, NCCL 2.30.4, cuDNN 9.10.2, Triton 3.6.0, and CPython 3.14.6. TriFast and Protenix autotune their Triton tiles on the first call for each shape, using caches generated on the H200 nodes. The remaining inspected kernels use fixed tile configurations (Table 4). Timing uses the cached configurations after warmup and does not measure cold-start time.

Numerical precision. The supplemental H200 training study runs without TF32. TurboPairFormer uses BF16 operands for tensor core contractions with FP32 accumulation. Softmax statistics, the row correction, and the ordered reduction sums use FP32, while the stored gradient partials and output residual use BF16. The FP64 computations used for numerical reference comparisons are separate from the timed BF16 training runs.

G.3 PDB Data, Cropping, and Frozen Inputs

The eight-GPU H200 comparison uses OpenFold3's `WeightedPDBDataset` with the `PDB-weighted-subset-4096` cache. A PDB entry describes a biomolecular structure or complex, potentially containing multiple chains. A training input is a token crop derived from an entry, so repeated samples from the same entry need not represent different structures. The crop sizes 384, 640, and 768 refer to the number of selected tokens supplied to the model, not to chain count or the length of a single protein sequence.

We first filter the common PDB cache by the uncropped token count to form a candidate pool for each target size. OpenFold3's DataModule, distributed sampler, and collator then construct inputs with data seed 42, four templates, chain cropping enabled, and token-crop sampling weights of 0.2 for contiguous cropping, 0.4 for spatial cropping, and 0.4 for spatial-interface cropping. Passing the initial length threshold does not guarantee a full-sized crop after preprocessing. We therefore reject generated samples with fewer valid tokens than the target, rather than filling the remainder with padding. Every accepted input has exactly the requested number of valid tokens and zero token padding. Validation is disabled during these timing runs.

Table 5 distinguishes the candidate structures from the accepted frozen input caches. For each crop size, every backend consumes the same first 3,200 cached rank-local inputs in the same order: one per

rank per optimizer step for 400 steps on eight GPUs. Warmup and measured steps use this common sequence. The cache counts and distinct-PDB counts describe the complete accepted cache, not the consumed prefix or independent experimental replicates.

Table 5: Data composition for the eight-GPU H200 comparison. Candidate entries meet the uncropped token-count threshold. Accepted cached inputs have exactly the target number of valid tokens; the distinct-PDB column counts entries represented in the complete accepted cache. All backend configurations consume 3,200 rank-local inputs per crop.

Crop tokens	Candidate entries	Accepted inputs	Rejected inputs	Distinct PDBs
384	2,279	3,421	79	1,155
640	1,371	3,659	141	812
768	1,109	3,651	149	711

Once a frozen input reaches the model, embeddings and pair updates produce Z. The fixed-token slices, query chunks, and outer groups in Sections B and D.4 partition attention on that input; they do not alter the crop or its chain identities. This data construction is specific to the H200 comparison and does not describe the separate H100 accuracy captures or multi-node fine-tuning runs.

G.4 Training Step Latency

Figure 8 and Table 6 report the supplemental eight-H200 comparison across eight backend configurations at crops 384, 640, and 768. Each value is the mean GPU computation time over steps 21–400, reported to two decimal places.

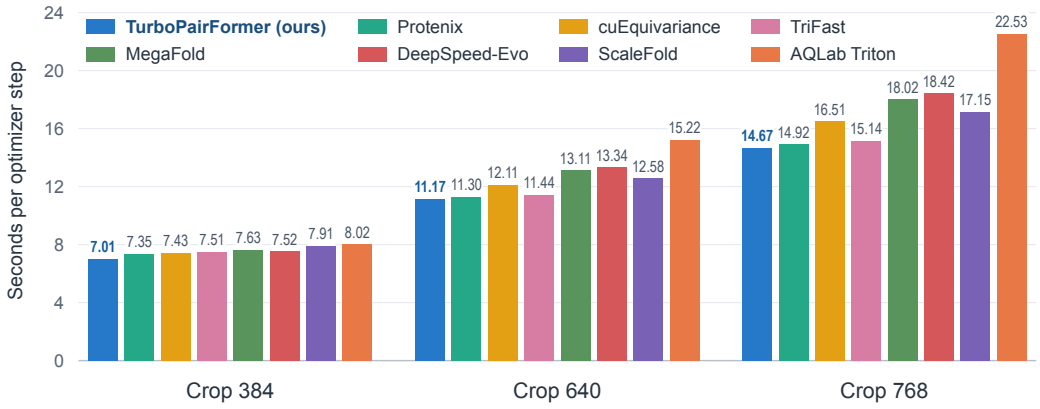


Figure 8: **Supplemental OpenFold3 training performance on 8xH200 GPUs.** Lower is better. Each configuration runs 400 optimizer steps with eight samples per step; means cover steps 21–400 after 20 warmup steps. CUDA events measure GPU training computation and parameter updates on common frozen inputs, taking the maximum duration across ranks; input loading, host-to-device transfers, and logging are excluded. TurboPairFormer has the lowest observed mean at all three crop sizes.

TurboPairFormer records mean step times of 7.01, 11.17, and 14.67 seconds at crops 384, 640, and 768, respectively, the lowest observed means among the eight configurations. Its step time is approximately 12.6%, 26.6%, and 34.9% lower than AQLab Triton at crops 384, 640, and 768, respectively. Relative to DeepSpeed-Evo, the corresponding reductions are approximately 6.8%, 16.3%, and 20.4%.

At crop 768, cuEquivariance and ScaleFold take 16.51 and 17.15 seconds per step, respectively; TurboPairFormer reduces these times by approximately 11.1% and 14.5%. Protenix and TriFast take 14.92 and 15.14 seconds, respectively, leaving smaller margins of approximately 1.7% and 3.1%. These are comparisons of integrated training configurations. The timing differences alone do not isolate the effects of attention recomputation, ordered reduction, or residual compensation.

Table 6: Mean seconds per optimizer step on eight NVIDIA H200 GPUs, excluding 20 warmup steps from each 400-step run. All eight configurations are complete at all three crop sizes. Protenix, TriFast, MegaFold, and ScaleFold use cuEquivariance triangle multiplication. Bold denotes the lowest mean in each column.

Backend	Crop 384	Crop 640	Crop 768
TurboPairFormer	**7.01**	**11.17**	**14.67**
Protenix	7.35	11.30	14.92
cuEquivariance	7.43	12.11	16.51
TriFast	7.51	11.44	15.14
MegaFold	7.63	13.11	18.02
DeepSpeed-Evo	7.52	13.34	18.42
ScaleFold	7.91	12.58	17.15
AQLab Triton	8.02	15.22	22.53

Across crops 384 to 768, TurboPairFormer's mean increases by a factor of 2.09, compared with 2.81 for AQLab Triton. The widening margin motivates optimizing pair updates as crop sizes increase; the integrated measurements do not attribute the entire improvement to attention alone.

Separate H100 operator measurements. Appendix H times the attention and triangle multiplication operators on a single NVIDIA H100. These operator timings are separate from the integrated H200 results, which enable both TurboPairFormer attention and TriMul.

Component attribution. The separate attention-only, TriMul-only, and joint-replacement study in Appendix J quantifies both operator contributions (Table 22). Its 100-step protocol uses gradient accumulation of 16 and five timing warmup steps, so its reductions are reported separately from the H200 comparison above. The combined gains are measured directly, rather than obtained by adding the individual percentages.

G.5 Paired Statistical Analysis

The paired analysis covers the original seven configurations at crops 384 and 640, using recorded per-step timings. Its confidence intervals and loss observations do not extend to DeepSpeed-Evo or crop 768; Table 6 reports the completed means for those additional results.

Matched inputs. The same step number identifies the same serialized batch, token crop, and padding in every configuration. The loss traces are consistent with this alignment: their mean correlation with TurboPairFormer's trace is 0.9987 at crop 640 and 0.9988 at crop 384. The standard deviation across configurations at a fixed step is 2.3% of the standard deviation of the step-mean loss across steps. Pairing by input removes much of the shared input-dependent timing variation. For example, at crop 640, TurboPairFormer's mean step time is 11171.8 ms with standard deviation 1024.8 ms, whereas its mean difference from Protenix is 133.2 ms.

Estimator. For each of the 380 measured steps, we subtract TurboPairFormer's computation time from the comparator's time on the same input. We average these paired differences and express the result as a percentage of the comparator's mean time. We compute 95% intervals using a moving-block bootstrap with block length 20 and 20 000 resamples, and also report a two-sided sign test. The lag-one autocorrelation of the paired differences lies between -0.12 and $+0.05$ across comparisons. Bootstrap intervals and ordinary paired intervals agree within a few hundredths of a percentage point.

Paired results. Table 7 reports these six comparisons at crops 384 and 640. At crop 640, the Protenix and TriFast reductions are 1.18% $[0.89, 1.53]$ and 2.31% $[1.98, 2.66]$, respectively. TurboPairFormer is faster on 273 and 288 of the 380 paired steps, with reported two-sided sign-test $p < 10^{-17}$ in both comparisons, and on all 380 steps relative to OpenFold3 Triton. Every crop-384 comparison also favors TurboPairFormer, with wider intervals. The crop-384 configurations span two consecutive job executions and are paired by batch identity; all crop-640 configurations ran within one job.

Table 7: Paired step-time reduction of TurboPairFormer relative to each comparator, as a percentage of the comparator's mean, with 95% moving-block bootstrap intervals for the original seven-configuration study at crops 384 and 640.

Comparator	Crop 384	Crop 640
Protenix	4.56 [4.06, 4.96]	1.18 [0.89, 1.53]
TriFast	6.61 [6.14, 7.10]	2.31 [1.98, 2.66]
cuEquivariance	5.59 [5.08, 5.96]	7.73 [7.47, 8.04]
ScaleFold	11.32 [10.79, 11.85]	11.17 [10.90, 11.51]
MegaFold	8.06 [7.68, 8.58]	14.81 [14.55, 15.08]
AQLab Triton	12.59 [12.23, 13.06]	26.60 [26.37, 26.87]

Interpretation and scope. These measurements compare integrated backend configurations. Because TurboPairFormer also replaces triangle multiplication, its margins against the four attention comparators include both operator choices; isolating attention requires a TurboPairFormer attention configuration paired with cuEquivariance triangle multiplication. The confidence intervals quantify within-execution uncertainty over matched inputs. They do not sample variation from recompilation, autotuning, clocks, or thermal state across repeated executions, so the approximately one-percent margin against Protenix remains conditional on these runs. For the original seven configurations at crops 384 and 640, the finite, similar loss traces provide evidence over the measured 400-step interval; convergence and final model quality require separate evaluation.

G.6 Training Loss Comparisons

The H200 loss summary in Table 8 covers the original seven configurations at crops 384 and 640, each run for 400 global optimizer steps on the common frozen input sequence. It excludes DeepSpeed-Evo and crop 768, for which the completed timing means are reported separately. Every loss in these seven-configuration runs remains finite. The final-50-step means are computed separately for each configuration, then summarized by their minimum and maximum. They describe observed agreement over this interval, without implying identical losses or equivalent convergence.

Table 8: Loss summary for the seven-configuration H200 comparison. Correlation is the reported mean correlation of the comparator traces with TurboPairFormer. Final-50 ranges are across configurations.

Crop	Mean correlation	Final-50 mean loss range	Absolute range
384	0.9988	1.31650–1.31981	0.00331
640	0.9987	1.35317–1.35514	0.00197

Resuming a shared step-1000 checkpoint. For each crop in Figure 4, the five configurations resume the same optimizer-bearing step-1000 checkpoint in new processes with the same seed, so their data order matches update by update, and train updates 1001–1400 on one H100 with validation disabled. TurboPairFormer runs with one recomputation and the output residual. Route receipts confirm that the AQLab arm executes OpenFold3's Triton triangle attention. After 400 updates, the cumulative mean loss of TriFast is 1.648, 1.625, and 1.609 at crops 384, 640, and 768; with the TurboPairFormer and cuEquivariance means of Section 3.3, the three configurations differ by at most 0.022 at every crop. MegaFold ends at 2.962, 1.746, and 2.680.

Separate 1,000-step H100 runs. We additionally compare the mean training loss of four configurations, PyTorch (stock attention), Triton, cuEquivariance, and TurboPairFormer, over separate 1,000-step runs at crop 384 on two nodes with eight H100 GPUs each. The runs use the dataset and hyperparameters of Appendix G.9 and start from two initial models, OpenBind0 and OpenFold3 Preview2. The four means span 1.1797–1.1802 for OpenBind0 and 1.1847–1.1851 for Preview2, and TurboPairFormer records 1.1802 and 1.1847, respectively. This dataset and protocol are separate from the H200 Weighted PDB benchmark; held-out structure quality is not inferred from the training loss means.

G.7 Additional Accuracy Results

Synthetic inputs. On one H100, we evaluate six shapes, $N \in \{384, 640, 768\}$ and $D \in \{16, 32\}$, with ten seeds per shape, batch size one, four heads, and N outer planes. For each of Q, K, V, we generate $X = Z + IU$ elementwise with independent $Z \sim \mathcal{N}(0, 1)$, $U \sim \mathcal{N}(0, 10^2)$, and $I \sim \mathrm{Bernoulli}(0.001)$, then round once to BF16. A CPU PyTorch generator with seed $s \in \{0, \ldots, 9\}$ draws Z, U, I in that order for Q, then K, then V, in FP64. Independently, seed $10^6 + s$ generates standard-normal FP64 dO, also rounded once to BF16. All implementations receive the same resulting tensors, zero shared BF16 bias, and zero FP32 mask; attention is noncausal with no dropout. We measure the absolute RMSE of public BF16 gradients against an FP64 reference evaluated on the same quantized inputs, reporting the mean and sample standard deviation over seeds. The implementation variants are specified below. Across all six shapes, TurboPairFormer has the lowest mean RMSE for dQ, dK, and $d\mathrm{bias}$ among the eight implementations, reducing error relative to cuEquivariance by 31.4–44.4%, 28.7–42.4%, and 16.9–37.9%, respectively. Its dV error is essentially tied with cuEquivariance.

Implementation variants. The accuracy and fixed-input replay studies use the same eight implementations in Table 9. Asterisks identify our numerical modifications to the named upstream kernels. These identities apply to these two studies; the separate H100 training-performance versions are given in Section G.1.

Table 9: Implementation identities for synthetic accuracy and replay. Hashes are SHA-256 prefixes of the attention source: forward/backward for TurboPairFormer, backward for Protenix and DeepSpeed-Evo, and the attention kernel for the other source-based entries.

Implementation	Base version	Kernel source SHA-256
TurboPairFormer	SR + residual	`03d2fb96e9/2cbb3f3c1e`
Protenix	`4c355be455`	`fac61c5601`
cuEquivariance	0.10.0	Native binary
AQLab Triton	Host `90e2008127`	`d704955ea4`
TriFast*	0.1.13, `2d5251135c`	`55e79b3633`
MegaFold*	`845905e7c7`	`b3cff75ce9`
ScaleFold (compat.)	Apex `8a6508aaad`	`4054c6672a`
DeepSpeed-Evo*	0.19.8, `bb5c23e025`	`090d365da8`

TriFast and MegaFold save the FP32 row maximum and exponential sum separately for backward probability reconstruction. TriFast uses base-2 maxima and an explicit rounded score conversion; its additive-mask adapter preserves the supplied mask values. The numerical-patch SHA-256 prefixes are `e7e553a4ef` and `43b87b4dbe`, respectively. DeepSpeed-Evo additionally centers scores in FP32 before exponentiation, accepts the FP32 additive mask, and corrects stream and boundary handling (patch `22f6f00c88`). ScaleFold uses Apex's direct-call path with an adapter for these shapes. The starred and compatibility entries therefore describe modified or adapted implementations.

Residual ablation. Enabling the saved output residual reduces the error of each of dQ, dK, and $d\mathrm{bias}$ in all 60 paired synthetic cases (Figure 9). Across the six shapes, mean RMSE decreases by 36.3–47.1%, 28.7–42.4%, and 28.3–43.1%, respectively. The dashed lines in the figure give the RMSE of the FP64 reference rounded once to BF16, the smallest RMSE any BF16 tensor can attain; relative to this floor, the mean dQ error falls from 2.76–3.37 to 1.76–1.78 times, and the mean dK and $d\mathrm{bias}$ errors from 1.98–2.50 to 1.42–1.43 times. The public output and dV remain byte-identical in all 60 on/off pairs, consistent with compensation entering the softmax backward correction. On captured real inputs, the corresponding OpenFold3 median errors decrease by 35.9–55.5%; the additional forward-plus-backward cost is 1.06–2.33% across five H100 operator cases.

Captured training inputs. Table 10 summarizes relative L_2 errors over 32 captured OpenFold3 attention calls using common inputs and an FP64 reference. TurboPairFormer's median dQ error is 0.3769%, approximately 54% lower than either comparator; dK and shared-bias gradients also improve, while output and dV remain close. This complements the synthetic absolute-RMSE comparison with inputs captured during training.

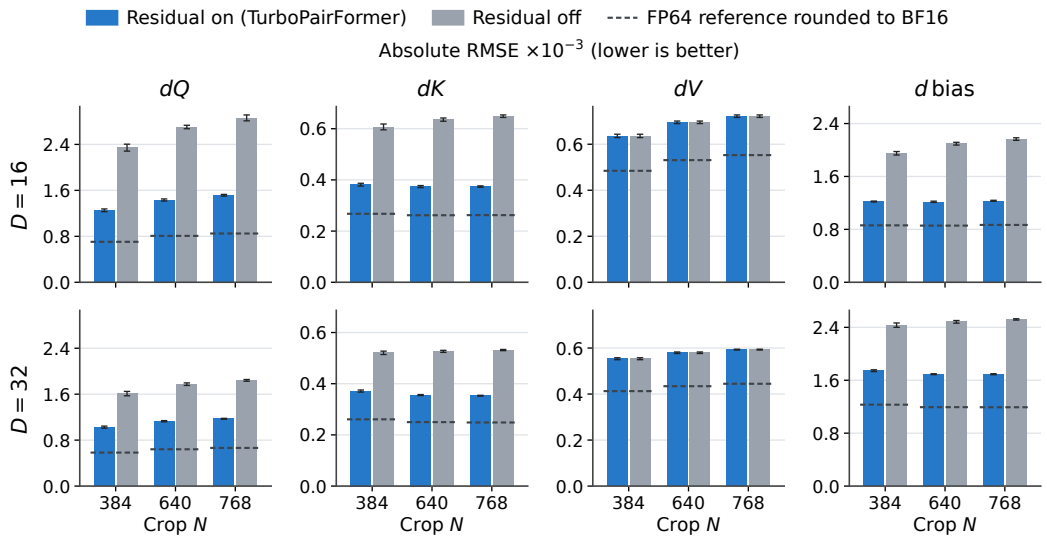


Figure 9: **Residual compensation on and off.** TurboPairFormer runs with and without the saved output residual on one H100, with identical inputs and incoming gradients. Bars show the mean absolute RMSE of the public BF16 gradients against an FP64 reference over ten seeds, and error bars the sample standard deviation. Dashed lines mark the RMSE of the FP64 reference rounded once to BF16. The residual enters only the softmax correction, so dV is bitwise identical in both settings.

Table 10: Median relative L_2 error (%) against an FP64 reference over 32 captured OpenFold3 attention calls. Output error is evaluated on valid query rows. Bold marks the lowest reported value in each row.

Quantity	Protenix	cuEquivariance	TurboPairFormer
O	0.1695	**0.1686**	**0.1686**
dQ	0.8202	0.8222	**0.3769**
dK	0.4347	0.4358	**0.2437**
dV	0.2201	**0.2164**	**0.2164**
d bias	0.3534	0.3506	**0.2449**

G.8 Deterministic Fixed-Input Replay

On one H100, we test seeds $s = 0, \ldots, 99$ for each of the six crop–width combinations, totaling 600 cases per method, using the input and dO generation recipe of Section G.7. Each case executes five forward and backward calls in the same process, building a fresh autograd graph each time with fixed BF16 inputs and dO, zero shared bias, and zero mask. A case passes when all four public gradients remain finite and bitwise identical across all five calls. Figure 10 summarizes the results: TurboPairFormer, Protenix, ScaleFold, and TriFast pass all 600 cases; MegaFold passes 165, while cuEquivariance, AQLab Triton, and DeepSpeed-Evo pass none. These measurements assess operator repeatability in the tested environment, not end-to-end training determinism: optimization can amplify small numerical perturbations (Summers & Dinneen, 2021). Implementation qualifiers follow the accuracy study above.

G.9 Sixteen-GPU Fine-Tuning Validation

Evaluation protocol. Fine-tuning uses crop size 384, seed 42, learning rate 3×10^{-4}, 50 warmup steps, EMA decay 0.98, and 32 diffusion samples on two nodes with eight H100 GPUs each. Each evaluation pass runs the native validation of the fine-tuning configuration on a final EMA checkpoint, using five rollout samples per target and averaging top-1 and best-of-five values for each metric before target aggregation. All checkpoints are evaluated with the same Triton validation kernels,

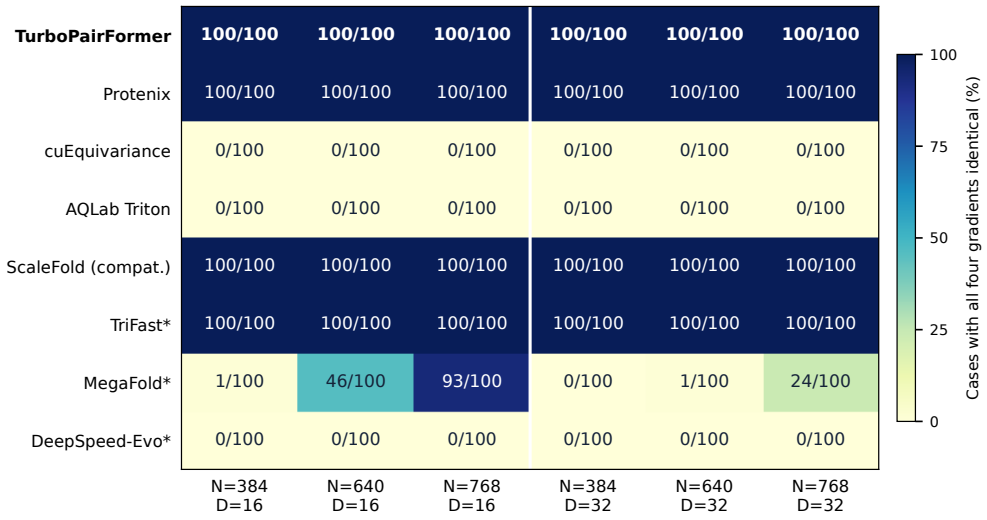


Figure 10: **Bitwise gradient replay across eight attention implementations.** Each shape contains 100 fixed synthetic inputs, with five calls per input on one H100. A case passes only when all four public BF16 gradients are finite and bitwise identical across all five calls; forward output is excluded from this joint criterion. Asterisks identify numerical-fix versions, and ScaleFold uses the compatibility version.

irrespective of the training backend; the starting Preview2 checkpoint (recorded as `of3-p2-155k`) is evaluated as the step-0 reference (The OpenFold3 Team, 2026). Before each target, the random number generators are seeded from the evaluation seed and the target identifier, so all checkpoints receive the same diffusion noise in a pass. Seeds 0–4 measure evaluation sampling variability for each checkpoint, rather than variation across independent training runs. The model-selection score weights the intra-protein, intra-ligand, and protein–protein lDDT by 20 and the protein–ligand lDDT and the relative accessible surface area (RASA) term by 10, applied to their eligible-target means and normalized by the total weight of 80.

Top-1 selects the rollout with the lowest global predicted distance error (gPDE). Best-of-five maximizes each lDDT, DockQ, GDT-TS, and RASA metric, or minimizes RMSD, independently. Figure 5 and Table 11 report the five-seed mean of the target aggregates; the figure also shows their sample SD.

The logged keys, all prefixed by `val/`, are `model_selection`; `lddt_intra_protein`, `lddt_inter_protein_ligand`, `lddt_intra_ligand`, and `lddt_inter_protein_protein` for the four lDDT columns; `dockq_protein_protein_w`; `gdt_ts`; and `rmsd`. Model selection additionally uses `rasa`.

The evaluation cohort contains 85 complexes, with metric availability depending on each complex's composition. The intra-protein, protein–ligand, intra-ligand, and protein–protein lDDT (Mariani et al., 2013) metrics have 11, 77, 66, and 12 eligible targets, respectively; weighted DockQ (Basu & Wallner, 2016) has 12, GDT-TS (Zemla, 2003) and RMSD have 85, and RASA has 83. Complete training-split and pretraining exclusion manifests were not verified for this comparison, so it does not establish temporal holdout or exclusion from Preview2 pretraining.

Results. Table 11 lists the means over the five seeds. The pooled standard deviation over seeds is 0.0045 for model selection, larger than the 0.0029 standard deviation of the nine fine-tuned means. For the paired comparison in Table 12, each target's metrics are averaged over the five seeds, and the model-selection difference is recomputed on 2,000 bootstrap resamples of the 85 targets. A second TurboPairFormer run with identical code, configuration, and seed, evaluated with seeds 0–2, scores 0.7185 in model selection, 0.0057 above the first run under the same seeds; this observed

run-to-run difference exceeds the magnitudes of seven of the eight TurboPairFormer–comparator mean differences. This pair of runs is insufficient to characterize training variability across backends.

Table 11: Validation after 1,000 global steps on 16 H100 GPUs, with the starting checkpoint (step 0) as a reference. Entries are means over five evaluation seeds on 85 kinase validation complexes, with metric-specific target counts given above. Structure metrics average top-1 and best-of-five values per target before aggregation. MS is the model-selection score; the four lDDT columns are intra-protein, protein–ligand, intra-ligand, and protein–protein. Higher is better except for RMSD (Å).

		lDDT						
Checkpoint	MS	Intra-prot.	Prot.–lig.	Intra-lig.	Prot.–prot.	DockQ	GDT-TS	RMSD
Preview2 (step 0)	0.722	0.800	0.738	0.943	0.504	0.424	0.784	3.81
PyTorch	0.717	0.809	0.760	0.947	0.488	0.412	0.792	4.10
Triton	0.713	0.806	0.741	0.943	0.490	0.415	0.786	4.35
cuEquivariance	0.718	0.810	0.749	0.947	0.495	0.421	0.789	4.17
Protenix	0.715	0.810	0.730	0.937	0.505	0.432	0.789	4.22
TriFast	0.717	0.812	0.741	0.944	0.498	0.417	0.794	4.27
MegaFold	0.723	0.811	0.756	0.944	0.512	0.432	0.796	4.13
DeepSpeed-Evo	0.715	0.808	0.739	0.941	0.499	0.427	0.794	3.96
ScaleFold	0.719	0.810	0.754	0.945	0.500	0.423	0.791	4.32
TurboPairFormer	0.714	0.808	0.747	0.943	0.489	0.413	0.790	4.31

Table 12: Paired model-selection difference, TurboPairFormer minus each checkpoint, with targets averaged over the five seeds and 95% bootstrap intervals over the 85 targets.

Checkpoint	Difference	95% interval
Preview2 (step 0)	-0.0076	$[-0.0236, +0.0052]$
PyTorch	-0.0030	$[-0.0092, +0.0022]$
Triton	$+0.0012$	$[-0.0036, +0.0057]$
cuEquivariance	-0.0035	$[-0.0092, +0.0024]$
Protenix	-0.0009	$[-0.0083, +0.0061]$
TriFast	-0.0030	$[-0.0094, +0.0032]$
MegaFold	-0.0082	$[-0.0169, -0.0002]$
DeepSpeed-Evo	-0.0006	$[-0.0077, +0.0055]$
ScaleFold	-0.0050	$[-0.0131, +0.0019]$

H Single-GPU Operator Measurements

This appendix times the triangular attention and triangle multiplication operators on one NVIDIA H100 80GB HBM3 GPU. Each measurement covers one operator call, so its reductions do not carry over directly to training step time: both operators are only part of a step, and an OpenFold3 update runs several attention forwards per backward pass (Section H.2). Table 22 reports the step-level effect of replacing each operator.

H.1 Triangular Attention

Setup. We time one attention call at crops $N \in \{384, 640, 768\}$ and head dimensions $D \in \{16, 32\}$, with $B = 1$, $N_o = N$, and $H = 4$, for TurboPairFormer, cuEquivariance 0.10.0, and Protenix. Every OpenFold3 triangular-attention layer uses $D = 32$, except those of the two-block template stack, which use $D = 16$. All three backends are called as the OpenFold3 attention layer calls them: Q, K, and V are head-major views of the BF16 projection outputs, the pair bias is a permuted view of its projection, and the FP32 mask is zero, which OpenFold3 converts to a Boolean key mask for cuEquivariance. Gradients are taken with respect to the projection outputs, so each backend's layout handling is part of the measured time. TurboPairFormer runs its training configuration, with one recomputation and the output residual, and uses one fixed tile configuration at every shape. Protenix requires parameter tuning: in each process, it first autotunes its tile configuration for each shape on

this GPU, and the timed calls use the selected configurations. CUDA events time forward calls with and without gradient recording, backward calls after an untimed forward, and complete forward-plus-backward calls. Each measurement pools five rounds of 15 calls, each round preceded by three warmup calls, with the backend order rotated between rounds. Every entry averages two independent processes, whose relative differences agree within 1.1 percentage points for cuEquivariance and 3.0 for Protenix. The software stack is PyTorch 2.10.0 with CUDA 12.8, Triton 3.6.0, and cuEquivariance 0.10.0.

Comparison with cuEquivariance. Table 13 reports the results. TurboPairFormer takes less time than cuEquivariance at every shape and in every mode, in all ten rounds of the two processes. At $D = 32$, its forward pass takes 29.8–33.1% less time, its backward pass 35.7–37.6% less, and forward plus backward 35.6–37.1% less; at $D = 16$, these reductions are 39.9–44.0%, 44.2–46.6%, and 43.1–45.6%. Without gradient recording, the mode of recycling iterations, the forward pass takes 29.6–43.7% less time. The gradients of the two backends differ by a relative L_2 distance of at most 3.8×10^{-3}.

Comparison with Protenix. TurboPairFormer's forward pass takes 3.9–12.5% less time than Protenix's at every shape, and 3.8–12.0% less without gradient recording. Its backward pass is 2.3–9.2% faster at $D = 16$ and 3.3% faster at $D = 32$ and $N = 640$, but takes 2.5% and 6.7% longer than Protenix's tuned configurations at $D = 32$ with $N = 384$ and 768. Forward plus backward ranges from 7.4% faster to 3.6% slower. The timed forward includes the output residual, which the current implementation also writes when gradients are not recorded; relative to the same kernel without the residual, it adds 3.8–9.4% to forward time without gradient recording and 1.0–2.6% to forward-plus-backward time. The gradients of TurboPairFormer and Protenix differ by a relative L_2 distance of at most 4.4×10^{-3}.

Table 13: Time in ms of one attention call on one H100 ($B = 1$, $N_o = N$, $H = 4$) for TurboPairFormer (Ours) in its training configuration, cuEquivariance 0.10.0 (cuEq), and Protenix after autotuning. The last three columns give TurboPairFormer's change in time relative to cuEquivariance; negative values mean TurboPairFormer is faster. Each entry averages two independent processes, each pooling five rounds of 15 calls.

		Forward			Backward			Forward + backward			Change vs. cuEq (%)		
D	N	Ours	cuEq	Protenix	Ours	cuEq	Protenix	Ours	cuEq	Protenix	Fwd	Bwd	F+B
16	384	0.384	0.640	0.417	1.167	2.097	1.285	1.493	2.623	1.587	−39.9	−44.3	−43.1
16	640	1.177	2.041	1.316	4.300	8.046	4.655	5.398	9.918	5.830	−42.3	−46.6	−45.6
16	768	1.870	3.341	2.137	7.392	13.244	7.563	9.025	16.478	9.499	−44.0	−44.2	−45.2
32	384	0.474	0.708	0.502	1.520	2.430	1.482	1.920	3.015	1.854	−33.1	−37.5	−36.3
32	640	1.538	2.192	1.602	5.468	8.764	5.654	6.810	10.823	7.129	−29.8	−37.6	−37.1
32	768	2.448	3.557	2.630	9.348	14.544	8.764	11.578	17.965	11.262	−31.2	−35.7	−35.6

One recomputation versus two. TurboPairFormer retains the two-pass backward of its earlier design, which recomputes the probabilities in one pass for the key and value gradients and again in a second pass for the query and pair-bias gradients. With the output residual disabled in both, the two-pass backward takes 28.3–64.0% longer than the single-recomputation backward (Table 14). Its passes use kernels different from the single-recomputation kernel, so the difference does not isolate the recomputation count.

Memory. The peak memory that one forward-plus-backward call allocates beyond its inputs is 0.36–3.09 GiB across the six shapes in the training configuration, compared with 0.19–1.46 GiB for cuEquivariance, 0.10–0.73 GiB for Protenix, and 0.24–1.23 GiB for the two-pass backward, which stores no query-gradient partials. At $D = 32$ and $N = 768$, the query-gradient partials occupy 1.69 GiB of the 3.09 GiB peak (Appendix E).

Table 14: Backward time in ms of one attention call on one H100 ($B = 1$, $N_o = N$, $H = 4$) for the single-recomputation backward and the earlier two-pass backward of TurboPairFormer, both without the output residual. Entries average the two processes of Table 13.

D	N	One recomputation	Two passes	Change
16	384	1.155	1.688	+46.1%
16	640	4.292	7.038	+64.0%
16	768	7.319	11.549	+57.8%
32	384	1.506	1.933	+28.3%
32	640	5.435	7.462	+37.3%
32	768	9.362	12.369	+32.1%

H.2 Forward Dataflow Ablation

Forward executions outnumber backward executions in OpenFold3 training. Up to three sampled recycling iterations run the trunk without gradients, and activation checkpointing recomputes the forward pass of the final iteration during backward. In a profiled crop-768 update with three recycling iterations, the triangular-attention calls comprised 600 forward executions, 480 in the model forward and 120 during recomputation, and 120 backward executions.

We isolate the forward dataflow of Sections C.1 and C.2 with a four-arm operator ablation on the same GPU. The baseline stages keys and values with the Tensor Memory Accelerator, but reads the pair bias and key mask directly from global memory and evaluates all probabilities of a key tile before value aggregation. The core arm adds bias staging with accumulator-layout loads, mask prefetching, and chunked probability evaluation. The pack arm changes only the vectorized kernel that repacks Q, K, and V into the attention layout. The combined arm contains both changes and matches the current forward kernel source. All arms share the same backward kernels. Inputs are synthetic BF16 projections with $B = 1$, $N_o = N$, $H = 4$, and $D = 32$. For each crop, eight blocks run the four arms in balanced order, with five warmup and 20 timed forward-plus-backward calls per arm and block, measured with CUDA events. In each block, the reduction is one minus the ratio of the arm's mean time to the baseline's mean time; the table reports its average over the eight blocks.

Table 15: Forward dataflow ablation on one H100 ($B = 1$, $N_o = N$, $H = 4$, $D = 32$). Times are means of 160 forward calls per arm; percentages are mean reductions over eight paired blocks relative to the baseline.

	Forward time (ms)			Forward reduction (%)		F+B reduction (%)	
N	Baseline	Core	Combined	Core	Combined	Core	Combined
384	0.547	0.408	0.388	25.5	29.1	8.1	9.2
640	2.037	1.478	1.426	27.4	30.0	8.3	9.0
768	3.336	2.419	2.341	27.5	29.8	7.8	8.5

Table 15 reports the results. Every arm reduced forward and forward-plus-backward time in all eight blocks at every crop; changing only the repacking reduced forward time by 2.4–3.5%. Backward time changed by at most 0.2% in any arm, consistent with the shared backward kernels. Across 13 inputs with three replays per arm, outputs, log-sum-exp values, residuals, and all four gradients were bitwise identical to the baseline. The core changes are measured together, so the ablation does not apportion their saving among bias staging, mask prefetching, and chunked evaluation. Exclusive use of the GPU was not confirmed during timing, and all samples are retained.

H.3 Triangle Multiplication

Setup. We time the triangle multiplicative update of Appendix J against cuEquivariance 0.10.0 at the twelve shapes of OpenFold3 training with one example: crops $N \in \{384, 640, 768\}$, 128 channels as in the Pairformer and MSA module or 64 channels as in the template stack, and the incoming and outgoing directions. Each timed call runs the forward and backward passes of one module under BF16 autocast, with FP32 parameters and a mask of zeros and ones, and returns

the gradients of the input and all ten parameters; TF32 is disabled. Each of three rounds times cuEquivariance, TurboPairFormer, and cuEquivariance again, with 20 warmup and 30 timed calls per segment. The software stack is PyTorch 2.10.0 with CUDA 12.9, and the TriMul kernel sources are identical to those of the evaluated build apart from a device check.

Table 16: Median time in ms of one triangle multiplicative update, forward plus backward, on one H100 with one example. Parentheses give TurboPairFormer's (Ours) change in time relative to cuEquivariance 0.10.0; negative values mean TurboPairFormer is faster. Medians pool all timed calls over three rounds: 180 for cuEquivariance and 90 for TurboPairFormer.

		Incoming		Outgoing	
N	Channels	cuEquivariance	Ours	cuEquivariance	Ours
384	128	2.769	1.326 (-52.1%)	2.777	1.323 (-52.4%)
640	128	3.615	3.334 (-7.8%)	3.678	3.336 (-9.3%)
768	128	5.142	4.788 (-6.9%)	5.210	4.782 (-8.2%)
384	64	2.734	1.111 (-59.4%)	2.735	1.116 (-59.2%)
640	64	2.777	1.921 (-30.8%)	2.784	1.922 (-31.0%)
768	64	2.842	2.692 (-5.3%)	2.879	2.691 (-6.5%)

Results. Table 16 reports the results. TurboPairFormer is faster at all twelve shapes, and in every round its median lies below the geometric mean of the two adjacent cuEquivariance medians. With 128 channels, it takes 52.1–52.4% less time at $N = 384$, 7.8–9.3% less at $N = 640$, and 6.9–8.2% less at $N = 768$. cuEquivariance takes 2.7–2.9 ms at every shape except the 128-channel shapes with $N \geq 640$, which indicates a per-call cost floor; the large reductions at $N = 384$ and with 64 channels reflect this floor. With 64 channels, the reduction falls from 59.2–59.4% at $N = 384$ to 5.3–6.5% at $N = 768$. In training, replacing triangle multiplication alone reduces step latency by 2.8–3.0% (Table 22).

I Choosing the Deterministic Reduction

With one recomputation, one thread block (CTA) emits the contributions of each tile to all four gradients, so every gradient that sums along an axis the thread block does not span requires a reduction across thread blocks. Under the work partitioning of Section 2.2, two such cross-block reductions remain: the query gradient over key tiles and the pair-bias gradient over outer groups. TurboPairFormer makes both deterministic by materializing per-tile partials in a global workspace, each written by exactly one thread block, and summing them in index order in FP32 with separate reduction kernels (Section 2.3). This appendix explains that choice. Section I.1 explains why our partitioning requires a cross-block pair-bias reduction, Section I.2 compares the three ways to make such a reduction deterministic (Figure 11), and Sections I.3 to I.5 measure the choice on one H100. At $D = 32$, the fastest tested non-deterministic variant of the same kernel uses atomic pair-bias accumulation and takes 2.8–4.1% less time than the deterministic kernel. The deterministic kernel is faster than assigning one thread block per slice at every shape. At $D = 16$, which only the template stack uses, the fastest tested non-deterministic variant takes 11.0–13.7% less time.

I.1 Which Gradients Need a Cross-Block Reduction

A backward pass that stores no score-sized tensor must recompute P and dP in every traversal that forms dS. Kernels that traverse the scores once per gradient need no cross-block reduction, because each traversal can span the axis along which its own gradient sums. TriFast and Protenix give every gradient element one writer this way, without atomics, at the price of three recomputations (Appendix F). In our code base, the earlier two-pass backward takes 28.3–64.0% longer than the single-recomputation backward (Table 14). With one recomputation, the work partitioning across thread blocks decides which gradients need a cross-block reduction and how many thread blocks a launch contains. Table 17 lists the candidates.

The pair-bias gradient stays within a thread block only if the thread block spans all N_o outer indices of its tile (lower block of Table 17). One block per query–key tile still gives $HJ_QJ_K = 4 \times 24 \times 12 =$

1,152 blocks at $N = 768$, but the query, key, and value gradients all need cross-block reductions. If it also spans queries or keys, only one thread block remains per head and tile: for key tiles, $HJ_K = 48$ thread blocks at $N = 768$ with four heads, fewer than the 132 SMs of an H100. To keep dK/dV accumulation local while exposing more parallel work, our backward pass partitions the outer indices across thread blocks and therefore reduces the pair-bias gradient across them. This is a design choice, not a requirement of every parallel backward algorithm. This reduction exists only because all slices share the bias; attention with a bias private to each slice has none.

Among the partitionings that split the outer indices (upper block), spanning more axes keeps more gradients local but leaves fewer thread blocks. Partitioning by key tile keeps dK and dV in registers and leaves the query gradient, summed over J_K key tiles, as the only slice-shaped gradient that needs a cross-block reduction; partitioning by query tile would leave two. Assigning one thread block per slice, as DS4Sci does, leaves only the pair-bias reduction but launches J_K times fewer thread blocks (Section I.4). Grouping $G = 4$ outer indices per thread block sums their pair-bias contributions on chip in a fixed tree, which divides the number of pair-bias partials by G.

Table 17: Work partitioning for one example. A thread block handles one head, owns the listed part of the (o, q, k) tile space, and iterates over the axes it spans. J_Q and J_K are the numbers of query and key tiles. The partials column counts the Q-sized partial tensors needed when the cross-block reductions of slice-shaped gradients use materialized partials; the upper block also needs $L = N_o/G$ bias-sized partials for the pair bias.

Thread block owns	Local	Cross-block	Partials	Thread blocks
G outer indices per thread block				
Key tile (ours)	dK, dV	dQ, d bias	J_K	HJ_KN_o/G
Query tile	dQ	dK, dV, d bias	$2J_Q$	HJ_QN_o/G
Whole slices (DS4Sci)	dQ, dK, dV	d bias	none	HN_o/G
All N_o outer indices per thread block				
Query–key tile	d bias	dQ, dK, dV	$J_K + 2J_Q$	HJ_QJ_K
Key tile	dK, dV, d bias	dQ	J_K	HJ_K

I.2 Three Ways to Make a Cross-Block Reduction Deterministic

A reduction across thread blocks is bitwise repeatable only if its association order is fixed or its arithmetic does not depend on the order. Order-independent summation can use binned accumulators (Ahrens et al., 2020) or floating-point expansions with superaccumulators (Collange et al., 2015). Attention kernels instead fix the order, in one of three ways (Figure 11a).

Semaphore-ordered atomic accumulation. Each thread block atomically adds its local contribution to one FP32 buffer in global memory, and a semaphore grants the turns in a fixed order; a thread block whose turn has not come waits. The deterministic mode of FlashAttention-3 accumulates the query gradient this way and, for non-causal attention, grants the turns in ascending key-tile order (Shah et al., 2024; Qiang et al., 2026).[6] DASH chooses the turn order together with the tile schedule so that the waits do not lengthen the critical path under its DAG scheduling model (Qiang et al., 2026). The model assumes zero-cost dependency edges; synchronization latency and register pressure can offset these gains on hardware.

One thread block per slice. A thread block that owns whole slices visits every key tile itself and accumulates the query gradient in program order, so the query gradient needs no cross-block reduction. DS4Sci obtains a deterministic query gradient this way (Song et al., 2023), and the deterministic mode of FlashAttention-2 reduces to it for triangular attention, as explained below. No thread block waits, but the launch has J_K times fewer thread blocks than partitioning by key tile.

Materialized partials and a reduction kernel. Each thread block writes its contribution as a per-tile partial that no other thread block writes, and after these thread blocks finish, a separate reduction kernel sums the partials in index order. The deterministic mode of FlashAttention-2 instead

[6] Fixed source: FA3 Hopper backward mainloop, `98eb7998`, lines 619–657.

keeps one FP32 query-gradient buffer per split[7]: it sets the number of splits to the SM count divided by the number of attention problems, rounded up, and each split's thread block traverses its key tiles in sequence. Triangular attention has BN_oH attention problems, 3,072 at $N = 768$ with four heads, far more than the 132 SMs of an H100, so this rule yields one split per slice, which is the one-thread-block-per-slice partitioning measured in Section I.4. TurboPairFormer keeps one thread block and one partial per key tile, and applies the same partials to the pair-bias gradient across outer groups, a reduction that standard attention does not have. Its reduction kernel adds the query partials in ascending key-tile order, the rule by which FlashAttention-3 grants its turns, but only after all thread blocks have finished, so no thread block waits. The partials cost memory, one additional pass over them, and, because they are stored in BF16, one rounding per partial before the FP32 sum.

Choice for triangular attention. The pair-bias gradient narrows the choice. Once the outer indices are partitioned, no single thread block covers its reduction. Semaphore-ordered accumulation would need, for every pair-bias tile, a chain of turns through all $L = N_o/G$ outer groups, 192 at $N = 768$, compared with the $J_K = 12$ key tiles of the query gradient, and a schedule that keeps both families of chains free of stalls. Materialized partials handle both reductions with one mechanism. Because each sum is ordered by partial index, they also let the planner split the backward pass into memory-bounded launches: FP32 continuation keeps every addition sequence intact across launches, so the gradient bits do not depend on the plan (Proposition D.2). TurboPairFormer therefore uses materialized partials for both reductions. All three mechanisms apply to the query gradient, and Sections I.3 and I.4 compare them within our kernel (Figure 11b).

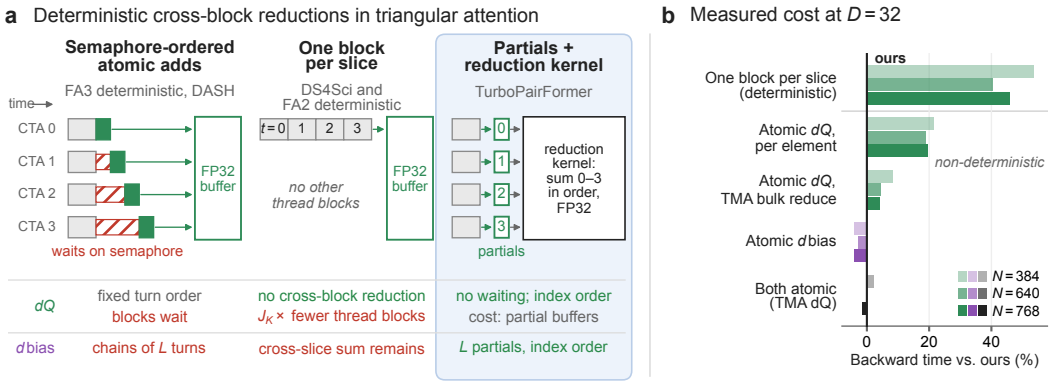


Figure 11: **Making the cross-block reductions deterministic.** (a) Four thread blocks (CTA 0–3) contribute to one query-gradient reduction; the bottom row gives each scheme's consequence for the pair-bias gradient, which sums across slices. With semaphore-ordered atomic adds, the blocks add into one FP32 buffer in turns and wait for them; FlashAttention-3 grants the turns in key-tile order, and DASH chooses the order together with the schedule. With one thread block per slice, a single block visits all J_K key tiles t of the slice, so the launch has J_K times fewer thread blocks; the deterministic mode of FlashAttention-2 falls in this class here, because it keeps one FP32 partial per split and sets the split count from the SM count, which gives one split per slice for the BN_oH attention problems of triangular attention. With materialized partials, blocks write one partial per key tile without waiting and a reduction kernel sums them in index order; TurboPairFormer uses this for dQ (J_K partials) and $d\,\mathrm{bias}$ (L partials). For the pair-bias gradient, semaphore ordering would need chains of L turns, and one thread block per slice still leaves a sum across slices. (b) Change in backward time of variants of the shipped kernel relative to the deterministic kernel at $D = 32$ on one H100 (Tables 18 and 20). The atomic variants are not deterministic; "both atomic" combines TMA bulk reduction for dQ with atomic $d\,\mathrm{bias}$. Atomic $d\,\mathrm{bias}$ alone gives the lowest measured time at each crop. Semaphore-ordered variants are not measured here. The one-block-per-slice variant is compared with the key-tile kernel of its own build.

[7]Official FlashAttention-2 implementation, commit `98eb7998`: FP32 buffers per split and deterministic launch and reduction.

I.3 Measured Cost of the Reduction Implementations

Variants. We replace the deterministic reductions of the shipped backward kernel with non-deterministic atomic accumulation and keep the thread-block partitioning, tiles, and arithmetic outside the replaced reductions unchanged. The pair-bias variant atomically adds the FP32 subtotal of each outer group to an FP32 gradient buffer. The query-gradient variants atomically add each thread block's local contribution to an FP32 buffer, either with per-element atomic adds from the matrix-multiply accumulator layout, or as in FlashAttention-3's accumulation of the query gradient, where each warp stages its 16-row tile in shared memory and adds it with one TMA bulk reduction (`cp.reduce.async.bulk`). Zeroing and converting these buffers are part of the timed backward call; the forward pass, the row correction, and workspace allocation are not. The deterministic variant reproduces the production kernel bit for bit. Each variant is timed with CUDA events in six rounds of 30 calls, each round preceded by five warmup calls, with the variant order rotated between rounds. An independent second process reproduces every percentage within 0.7 percentage points.

Table 18: Backward time of one TurboPairFormer attention call on one H100 ($B = 1$, $N_o = N$, $H = 4$). The deterministic kernel reduces both gradients through materialized partials. Each other column replaces the named reductions with non-deterministic atomic accumulation in FP32 and gives the change in time relative to the deterministic kernel; positive values mean the variant is slower. "Both" combines atomic pair-bias accumulation with the named query-gradient accumulation. Each entry pools six rounds of 30 calls.

			Change with non-deterministic atomic accumulation (%)				
D	N	Deterministic (ms)	Atomic d bias	Per-element dQ	Both, per-element	TMA bulk dQ	Both, TMA bulk
16	384	0.849	−8.0	+1.9	−6.7	−5.2	−11.0
16	640	3.582	−7.3	−0.3	−7.2	−7.1	−12.6
16	768	6.349	−7.9	−0.1	−7.8	−7.2	−13.7
32	384	1.091	−4.1	+21.5	+13.6	+8.3	+2.2
32	640	4.505	−2.8	+18.9	+13.3	+4.6	−0.1
32	768	8.045	−3.9	+19.6	+13.0	+4.2	−1.3

Results. Table 18 reports the backward times. At $D = 32$, the materialized query-gradient partials are faster than either atomic alternative: replacing them lengthens the backward pass by 18.9–21.5% with per-element atomic adds and by 4.2–8.3% with TMA bulk reductions. Replacing only the pair-bias partials with atomics gives the fastest tested non-deterministic variant at all three crops, shortening the pass by 2.8–4.1%. Combining atomic pair-bias accumulation with TMA bulk reduction of the query gradient instead changes the time by -1.3% to $+2.2\%$. At $D = 16$, TMA bulk reduction is 5.2–7.2% faster than the query-gradient partials, atomic pair-bias accumulation saves 7.3–8.0%, and combining these two changes gives the fastest tested non-deterministic variant, taking 11.0–13.7% less time than the deterministic kernel. Relative to the fastest tested atomic variant, the deterministic implementation takes approximately 2.9–4.3% longer at $D = 32$ and 12.4–15.9% longer at $D = 16$. This compares BF16 partials followed by fixed FP32 reduction with direct FP32 atomic accumulation, so it does not isolate the cost of determinism. These measurements do not bound the performance of semaphore-ordered accumulation: ordering can also change contention and scheduling, so such variants require direct measurement.

Hardware counters. Table 19 reports Nsight Compute counters for the first launch of the main backward kernel at $N = 768$ and $D = 32$. Per-element atomic adds issue one reduction instruction per accumulator element, and each warp instruction touches eight rows. The L2 cache receives 99.1 million single-sector reduction requests, its throughput rises from 22.6% to 52.8% of peak, long-scoreboard stalls per issued instruction nearly double, and the kernel runs 25.6% longer, although 98.0% of the reduction sectors hit in L2. TMA bulk reductions cut the requests to 12.4 million of four sectors each, but their FP32 read-modify-write traffic still raises L2 throughput to 35.4% and lengthens the kernel by 10.4%. The deterministic path instead writes BF16 partials with plain stores, and its reduction kernel reads the 1.69 GiB of query-gradient partials once, in 0.63 ms under the profiler's clocks. The pair-bias atomics are coalesced and lengthen the main kernel by 1–2%; the

deterministic pair-bias path instead pays for its reduction kernels, which read the BF16 partials of each launch at 2.3–2.4 TB/s.

Table 19: Nsight Compute counters for the first launch of the main backward kernel at $N = 768$ and $D = 32$, for three query-gradient reductions with materialized pair-bias partials. The profiler serializes kernels, flushes caches, and locks clocks to the base frequency, so its durations exceed the CUDA-event times of Table 18.

Query-gradient reduction	Kernel (ms)	L2 reduction requests (M)	Sectors per request	L2 throughput (% of peak)	Long-scoreboard stalls per issue
Materialized partials	2.52	0	–	22.6	0.96
Per-element atomic adds	3.17	99.1	1	52.8	1.82
TMA bulk reduction	2.79	12.4	4	35.4	1.19

I.4 One Thread Block per Slice

Each thread block of TurboPairFormer owns one key tile, so the query gradient needs a cross-block reduction. Assigning a thread block the whole slices of its four outer indices removes this reduction: the thread block then visits every key tile in ascending order, restaging keys, values, and the mask for each, and accumulates the query gradient in a dense FP32 buffer with plain loads and stores, so no query-gradient partials or reduction kernel are needed. This is the work partitioning of DS4Sci, combined with our grouping of four outer indices and unchanged pair-bias path. We built it as a variant of the shipped kernel and timed both under two workspace plans: the default plan, which splits the outer groups across several launches at $N = 768$ for both head dimensions and at $N = 640$ for $D = 16$, and a plan with every outer group resident, which the variant can afford because it stores no query-gradient partials.

Table 20: Backward time in ms of one attention call on one H100 ($B = 1$, $N_o = N$, $H = 4$) for the shipped key-tile partitioning and the one-thread-block-per-slice variant under the same workspace plan. Parentheses give the change relative to the key-tile kernel under that plan. Both kernels come from the same build and are timed in five rounds of 30 calls with rotated order.

		Default workspace		All outer groups resident	
D	N	Key tile	Per slice	Key tile	Per slice
16	384	0.837	1.455 (+73.8%)	0.836	1.437 (+71.9%)
16	640	3.554	6.962 (+95.9%)	3.497	5.801 (+65.9%)
16	768	6.317	11.356 (+79.8%)	5.994	8.700 (+45.2%)
32	384	1.130	1.733 (+53.4%)	1.134	1.742 (+53.6%)
32	640	4.709	6.608 (+40.3%)	4.708	6.610 (+40.4%)
32	768	8.440	12.317 (+45.9%)	8.012	9.837 (+22.8%)

The per-slice variant is deterministic: its gradients repeat bitwise, its key, value, and pair-bias gradients equal those of the shipped kernel bit for bit, and its query gradient agrees with FP32 atomic accumulation to a relative L_2 error of at most 2.4×10^{-5}. It is nevertheless 22.8–95.9% slower (Table 20). The gap narrows when more outer groups run concurrently, from 45.9% to 22.8% at $D = 32$ and $N = 768$, which points to its lower parallelism: a launch contains J_K times fewer thread blocks, and each traverses every key tile in sequence. At $D = 32$ its instantiation also uses a larger stack frame than the key-tile kernel of the same build (224 versus 144 bytes), which may contribute to the gap, although its effect was not isolated; at $D = 16$ both use 112 bytes and the gap is larger. Partitioning by key tile therefore spends the cubic query-partial buffer on J_K times more thread blocks. The key-tile loop added to this build raises the shipped kernel's stack frame at $D = 32$ from 80 to 144 bytes, and its absolute times at $D = 32$ are 3.6–4.9% higher than in Table 18; the table therefore supports comparisons within the build only.

I.5 Workspace Plans, Memory, and Precision

Plan invariance. The same build tests Proposition D.2. At every shape, the shipped kernel's four gradients are bitwise identical under the default plan and under the plan with every outer group resident, although at $N = 768$ the default plan processes the 192 outer groups in four launches and the other processes them in one. The resident plan shortens the backward pass by 5.1% at $N = 768$, at the cost of an 864 MiB pair-bias workspace instead of 252 MiB; the default budget gives up this time for memory headroom in the training step (Appendix E).

Memory. The materialized partials are the main cost of the design. At $D = 32$ and $N = 768$, the BF16 query-gradient partials occupy 1.69 GiB, whereas the FP32 query-gradient buffer of semaphore-ordered or per-slice accumulation occupies 288 MiB; the partial buffer grows cubically with crop size (Appendix E).

Precision. Storing the partials in BF16 halves their memory but rounds each partial before the FP32 sum. Relative to FP32 atomic accumulation, this changes the query gradient by a relative L_2 error of 2.7–2.8 $\times 10^{-3}$ and the pair-bias gradient by 1.6–1.7 $\times 10^{-3}$ at every shape, whereas FP32 accumulations that differ only in addition order agree within 2.4 $\times 10^{-5}$. The accuracy comparison of Section 3.2 includes this rounding. Appendix G.8 reports bitwise replay of all four gradients over 600 fixed inputs.

J Triangle Multiplication

Figure 12 shows the fused forward stages and the saved output gate.

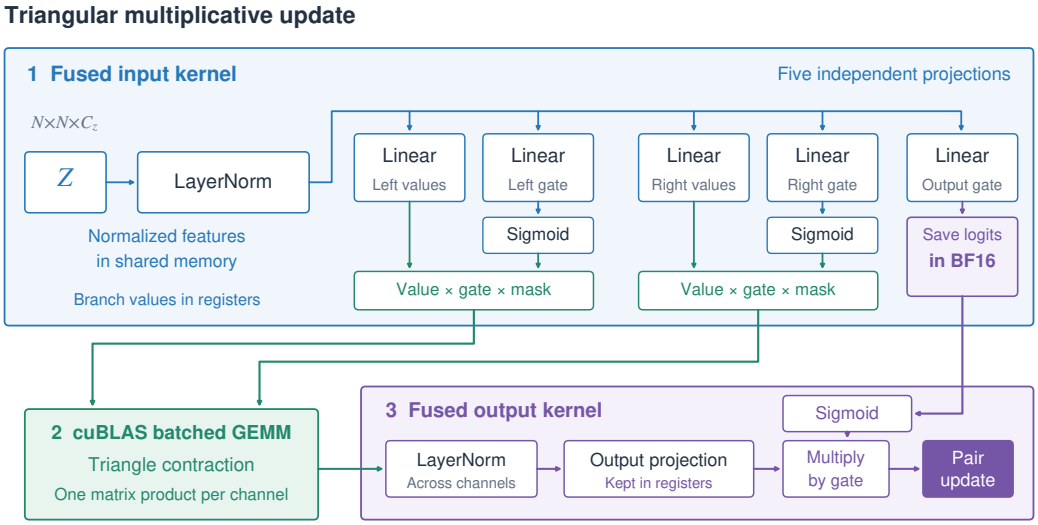


Figure 12: TurboPairFormer outgoing triangular multiplicative update. The fused input kernel normalizes the pair features and computes five independent projections: two branch values, two branch gates, and the output gate logits. The normalized features use shared memory, while branch values remain in registers until gating and masking. A separate cuBLAS batched GEMM contracts the branch operands independently for each channel. The fused output kernel combines normalization, output projection, and multiplication by the sigmoid of the saved output gate logits. The output projection remains in registers until gating. In the depicted implementation, the output gate logits are computed in the input kernel and saved in BF16; the compared cuEquivariance 0.10.0 path instead computes them in the output stage and retains them in FP32 registers.

Triangle multiplication updates a pair by combining two transformed pair features that share a third token. For the outgoing update, the two pairs share their second token; for the incoming update, they share their first. The contraction sums over this shared token separately for each feature channel. These updates operate on the same pair representation Z as triangular attention.

Fused forward execution. The outgoing implementation places two fused CUDA kernels around a cuBLAS batched matrix multiplication. The input kernel normalizes each pair feature vector and evaluates five independent projections: a value and gate for each of the two contraction branches, and the output gate logits. It applies the branch sigmoid gates and mask before writing the two contraction operands. The normalized input resides in shared memory for reuse by the projections; intermediate branch values reside in registers. For the illustrated 128-channel configuration, the input has logical shape $[N, N, 128]$, and each contraction operand has shape $[128, N, N]$.

cuBLAS performs one matrix multiplication per channel, contracting the shared token axis. The output kernel normalizes the contraction result, applies the output projection, and multiplies it by the sigmoid of the saved output gate logits. The projected update remains in registers until gating. Fusion therefore removes separate global-memory writes and reads for intermediates within the input and output stages; the branch tensors and contraction result still connect the three stages through global memory.

Backward execution and saved precision. Backward recomputes projection tiles and partitions weight-gradient contributions into chunks that are combined in a fixed FP32 order. This is a separate reduction design from the shared pair-bias reduction in triangular attention. The attention replay results in Figure 10 do not measure TriMul replay.

Table 21 compares the fusion boundaries with cuEquivariance 0.10.0. A consequential difference is when the output gate is evaluated. TurboPairFormer computes its logits during input projection and stores them in BF16. cuEquivariance computes them in the output stage and retains them in FP32 registers. Thus the implementations share the same mathematical update but have different saved precision and intermediate rounding points.

Table 21: Triangle multiplication execution in cuEquivariance 0.10.0 and TurboPairFormer. The comparison describes the outgoing update.

Stage	cuEquivariance	TurboPairFormer
Input	LayerNorm followed by fused branch projections and gates	One kernel for LayerNorm, branch projections, gates, mask, and output gate projection
Contraction	Triangle einsum	cuBLAS batched matrix multiplication per channel
Output	Normalization and transpose followed by fused output	One kernel for normalization, output projection, and output gating
Output gate	Logits computed at output stage, retained in FP32 registers	Logits computed at input stage, stored in BF16

Component ablation. Table 22 compares replacing attention, TriMul, or both against a cuEquivariance-enabled OpenFold3 baseline. This separate study reports 100 optimizer steps per crop and configuration, gradient accumulation of 16, and five timing warmup steps. These settings differ from the eight-H200 comparison in Section G.2, which uses 400 global steps without gradient accumulation. The percentages below are reductions in full training step latency, not isolated TriMul kernel speedups. Each combined result is measured directly.

Table 22: Training step latency reduction (%) in the separate attention/TriMul component ablation against a cuEquivariance-enabled baseline.

Crop	Attention only	TriMul only	Both
384	2.21	2.86	5.55
640	5.59	2.78	8.45
768	6.43	2.99	9.63

Precision comparison. Table 23 reports median relative L_2 errors for the TriMul output and input gradient against a PyTorch FP32 reference. The measured TurboPairFormer and cuEquivariance BF16 values are close: TurboPairFormer is slightly higher for the output and slightly lower for dZ. This

comparison evaluates the complete TriMul implementations, including their fusion boundaries and saved precision; it does not isolate the effect of BF16 gate storage. The synthetic and captured-input FP64 studies in Section G.7 instead concern triangular attention.

Table 23: TriMul median relative L_2 error (%) against a PyTorch FP32 reference. All three evaluated implementations use BF16 inputs.

Quantity	PyTorch	cuEquivariance	TurboPairFormer
Output	0.440	0.384	0.386
Input gradient dZ	0.823	0.607	0.604