Efficient Multi-Scale Deformable Attention on GPUs

Anonymous authors
Paper under double-blind review

Abstract

Multi-scale deformable attention (MSDA) is a core operator in DETR-family vision transformers whose scattered bilinear sampling pattern defeats the tile-based strategies on which FlashAttention-style kernels depend. We present a diagnostic study of GPU kernel optimization for MSDA on NVIDIA A100 (SM 8.0) and H100 (SM 9.0), identifying two failure modes of conventional heuristics and a root cause that is both hardware- and compiler-gated. Dispatch-order reordering does not pay: seven query orders (linear, Morton Z-order, random, scanline, Hilbert, centroid, and a clustering-and-packing analogue) produce within-±2% forward latency at K=4, L=4 because L2 locality is tile-set by the query-block kernel rather than by the dispatch order. Throughput proxies mislead: an 85 %-occupancy point-parallel tiling delivers only 5.1 % of A100 peak bandwidth, while a 17 %-occupancy query-block tiling delivers 36 % and runs 7.4× faster. The backward-pass bottleneck is scattered-gradient atomic contention: at BF16, the backward kernel attains 2.4 % of A100 peak bandwidth versus 21.3 % on H100. The gap is hardware- and compiler-gated: Ampere has no native BF16 atomic instruction (forcing a 32-bit compare-and-swap emulation), and on H100 the standard CUDA atomic still lowers to that emulation while a relaxed-ordering variant reaches Hopper's native reduction primitive; an FP32-accumulator variant closes the A100 gap entirely. The resulting backends deliver 2.4–14× forward speedup and up to 88 % peak VRAM reduction over the reference implementation at numerical parity.

1 Introduction

Multi-scale deformable attention (MSDA) is a core component of modern DETR-family detectors and segmentation models, and an efficiency bottleneck: Zhao et al. (2024) report that the MSDA encoder contributes 49 % of GFLOPs but only 11 % of average precision in Deformable DETR, and Cavagnero et al. (2024) identify Mask2Former's deformable-attention pixel decoder as an efficiency bottleneck that limits real-world deployment. Unlike dense self-attention, where FlashAttention-style tiling approaches peak throughput by exploiting contiguous query/key/value layout, MSDA performs scattered bilinear sampling at data-dependent offsets across multiple feature scales; sampled positions are unknown until runtime, precluding the regular tile structure on which efficient attention kernels depend.

We present a diagnostic empirical study of GPU kernel optimization for this operator class on NVIDIA A100 (SM 8.0) and H100 (SM 9.0), isolating two failure modes of conventional optimization heuristics and a hardware-and-compiler-gated root cause of the backward bottleneck. *Dispatch-order reordering does not pay*: seven query-dispatch orders at K=4, L=4 (linear, Morton Z-order, random permutation, scanline, Hilbert, centroid, and a DANMP-style K-means clustering-and-packing analogue) produce within-±2% forward latency because L2 locality is tile-set by the query-block kernel rather than order-set by the global dispatch (Section 3.1). *Throughput proxies mislead*: an 85 %-occupancy Triton tiling delivers 5.1 % of A100 peak bandwidth while a 17 %-occupancy tiling runs 7.4× faster (Section 3.2). At BF16 our Triton backward kernel achieves 2.4 % of A100 peak bandwidth versus 21.3 % on H100 (Section 3.3). The cross-generation gap is hardware-gated: Ampere (SM 8.0) has no native BF16 atomic instruction, so both code paths must emulate half-precision atomic addition as a 32-bit compare-and-swap retry loop. A second, compiler-gated gap appears within Hopper (SM 9.0a): the half-precision atomic addition supplied by the standard CUDA math headers (CUDA 12.8) still lowers through the device compiler to a compare-and-swap loop, while a relaxed-ordering

atomic addition reaches the native Hopper primitive — the PTX assembler promotes a relaxed-ordering BF16 atomic add to the single-instruction L2 reduction form (Section 4.2). An FP32-accumulator variant closes the A100 gap, making the backward choice an accumulator-precision decision rather than a programming-model preference. Applied together, our kernel techniques deliver a median 2.4–2.7× forward speedup at decoder scale and 12–14× at encoder scale, with up to 88 % peak VRAM reduction.

Contributions. Our contribution is the diagnostic characterization and the matched-code cross-GPU comparison: the kernel techniques themselves are standard (Section 2.4), and unlike the nearest prior work (DCNv4 (Xiong et al., 2024), which validates with a theoretical cost model) we report empirical roofline profiling, a cross-paradigm programming-model comparison, and documented negative results. Specifically: (i) two failure modes of conventional GPU optimization heuristics on MSDA, locality reordering and the occupancy-bandwidth inversion, quantified for a deformable attention operator; (ii) a hardware-and-compiler-gated root cause for the backward bottleneck, with an FP32-accumulator mitigation that closes the A100 gap, independently validated by concurrent atomic-reduction work on differentiable rendering (Durvasula et al., 2025; Mallick et al., 2024); and (iii) an open-source library with both CUDA and Triton backends delivering up to 14× forward speedup and 88 % peak VRAM reduction.

2 Background

2.1 Multi-Scale Deformable Attention

Multi-Scale Deformable Attention (Zhu et al., 2021) replaces the $\mathcal{O}(N^2)$ dot-product attention of the original DETR (Carion et al., 2020) with a fixed number of bilinear lookups across multiple feature scales. Given N_q query elements and L feature maps $\{\mathbf{x}^l\}_{l=1}^L$ at different resolutions, the operator is

$$\mathrm{MSDA}\big(\mathbf{z}_q,\, \hat{\mathbf{p}}_q,\, \{\mathbf{x}^l\}_{l=1}^L\big) = \sum_{m=1}^{M} \mathbf{W}_m \left[\sum_{l=1}^{L} \sum_{k=1}^{K} A_{mlqk}\, \mathbf{W}'_m\, \mathbf{x}^l\big(\phi_l(\hat{\mathbf{p}}_q) + \Delta\mathbf{p}_{mlqk}\big) \right], \tag{1}$$

where $m \in \{1, \dots, M\}$ indexes attention heads and $k \in \{1, \dots, K\}$ indexes sampling points per head per level; $\hat{\mathbf{p}}_q \in [0,1]^2$ is a normalized 2D reference point for query q; $\Delta\mathbf{p}_{mlqk} \in \mathbb{R}^2$ are 2D offsets predicted by a linear projection of $\mathbf{z}_q$; $A_{mlqk} \geq 0$ with $\sum_{l=1}^{L}\sum_{k=1}^{K} A_{mlqk} = 1$ are scalar attention weights, also predicted from $\mathbf{z}_q$; $\phi_l(\hat{\mathbf{p}}_q)$ rescales the reference point to level l's pixel coordinates; and $\mathbf{x}^l(\mathbf{p})$ denotes bilinear interpolation on $\mathbf{x}^l$ at a 2D coordinate $\mathbf{p} \in \mathbb{R}^2$ (reading the four nearest integer-coordinate neighbors and computing a weighted average). In typical Deformable DETR configurations, M=8, L=4, K=4, giving 128 bilinear lookups per query element. Compared with dense multi-head self-attention (which scales as $O(MN_q^2)$ in time and memory), MSDA requires only $O(MN_qLK)$ work. The cost is qualitative: the LK sampled locations are determined by learned offsets that vary per query, producing *scattered* irregular memory accesses rather than the regular, tile-friendly access pattern of matrix multiplication.

Computational profile. The forward pass is dominated by irregular DRAM reads: for each of the N_q queries, the kernel reads $M \cdot L \cdot K \cdot 4 = 512$ feature map values (four neighbors per sample point) from potentially non-contiguous addresses. The backward pass additionally requires scattered atomic writes: gradients for each sampled feature map position must be atomically accumulated into the output gradient tensors, causing contention when nearby queries sample overlapping regions. Neither phase contains the large contiguous tile accesses that enable FlashAttention-style SRAM tiling (Dao, 2024; Shah et al., 2024). The optimization challenge is therefore fundamentally different from dense attention: progress requires reducing per-access cost and serialization overhead, not tiling the computation.

2.2 GPU Programming Models

We compare two programming models throughout the paper. *CUDA C++* exposes the thread/warp/block hierarchy and gives the programmer explicit control over shared memory, warp-level shuffle primitives, and atomic operations (atomic-add and compare-and-swap). For MSDA this control is necessary to broadcast sampling coordinates within a warp and to select per-SM atomic primitives, at the cost of hardware-specific

engineering that must be revisited each GPU generation. *Triton* (Tillet et al., 2019) operates on 1-D blocks; its compiler maps blocks to warps, coalesces block-internal memory accesses, and auto-tunes block sizes, excelling at regular-tile workloads. Triton also exposes atomic operations with a configurable memory-ordering qualifier that controls the coherence fences emitted around each transaction. For MSDA, Triton's tiling advantage applies naturally to the *query* dimension (consecutive queries are contiguous), but the sampled feature-map addresses remain data-dependent in both models. The two programming models thus face the same data-dependent I/O problem and differ only in how directly each exposes the scheduling primitives that move bytes.

2.3 Access-Pattern Characterization

We characterize the reference MSDA implementation from Zhu et al. (2021) on an NVIDIA A100-SXM4-40GB (SM 8.0, 1.56 TB/s peak memory bandwidth) using hardware-counter profiling at the standard decoder configuration (B=4, N_q=300, H=W=800, d=256, L=4, BF16). The reference forward kernel sustains only 29.3 % of peak memory bandwidth (Figure 1) while reaching well below the A100 compute ridge, and the reference backward kernel is twice as expensive as the forward for structural reasons we preview here and quantify in Section 3. Two aspects of the access pattern are load-bearing for the rest of the paper and together motivate the diagnostic structure of Section 3.

Read-side: scattered loads with no stable reuse. For each of the N_q queries, the forward pass issues $M{\cdot}L{\cdot}K{\cdot}4{=}512$ bilinear-neighbour reads at addresses that are the output of a learned offset network. Two queries may sample nearby image regions on one iteration and disjoint regions on the next, so there is no iteration-stable reuse pattern for cache or prefetcher policies to exploit. The only regular structure is the sequential advance of the *query index*, which the L2 pre-fetcher can follow as long as the kernel dispatches queries in their original order. Any reordering that attempts to manufacture spatial locality (Section 3.1) replaces this monotonic stream with a data-dependent permutation and cannot recover the lost streaming behavior with software cache hints.

Write-side: scattered atomic gradient accumulation. The backward pass inverts the access direction: each contributing bilinear neighbour of every sampled position receives an atomic accumulation of its gradient share. Nearby queries that happen to sample overlapping feature regions contend on the same cache lines, and because sampling locations are data-dependent the set of contenders is not knowable at dispatch time. Under the encoder-scale feature maps that dominate training wall-clock time, dozens of warps can contend on the same atomic target within microseconds. The cost of this contention depends entirely on the hardware's atomic primitive for the accumulation data type (Section 3.3), not on the programming model used to write the kernel.

A secondary buffer observation. In addition to the two access-pattern properties above, the reference implementation concatenates sampling coordinates and attention weights into a single (B, N_q, L, K, 3) tensor (the three components are the two spatial offsets (x, y) and the scalar attention weight) before launching the CUDA kernel. The buffer is unnecessary (a kernel can read coordinates and weights from disjoint pointers at identical cost), and eliminating it removes a concatenation allocation from the forward critical path. We include this observation because it informs the kernel design described in Section 2.4, but it is not a central diagnostic finding of the paper: its performance impact is within a few percent of peak at the configurations we measured.

2.4 Kernel Design

The kernels we benchmark apply three standard optimizations to the access pattern above: vectorized 128-bit feature-map loads, warp-level sharing of the $M{\cdot}L{\cdot}K$ sampling parameters, and relaxed-ordering atomic addition in the gradient-scatter loop (safe because gradient accumulation is commutative and the kernel-end barrier carries the happens-before ordering). The CUDA backward additionally upcasts gradient tensors to FP32 before the kernel launch, routing through the hardware-native FP32 atomic-addition primitive and down-converting on return.

These techniques narrow but do not eliminate the BF16 atomic cost on SM 8.0; the residual gap is hardware-gated (Section 3.3). Full pseudocode for the two patterns appears in Section E; Section D isolates each technique's contribution.

3 Failure Modes and Mechanisms

This section presents two failure modes that we quantify while optimizing MSDA for commodity NVIDIA GPUs, and the hardware-gated root cause of the second. The failures themselves are, in hindsight, intuitive: scattered access has no exploitable locality, and occupancy need not predict bandwidth. The literature has not previously measured either effect for an attention operator. Each subsection is organized around the same three questions: what fails, by how much, and why. The failure modes are mechanistically distinct (one reflects a property of the operator, the other a property of the hardware), and separating them is the central diagnostic contribution of this paper.

3.1 Locality failure: dispatch-order reordering does not pay

The first optimization we attempted is among the most widely recommended for memory-bound kernels: reorder the work items so that neighboring threads access neighboring addresses. For MSDA this is especially tempting because the dominant cost is scattered bilinear reads of multi-scale feature maps. If queries whose sampling locations fall near one another in image space were grouped onto the same thread blocks, the argument goes, their neighbour fetches would overlap in L2 and bandwidth demand would fall.

Measurement. We tested seven query-dispatch orders on the same Triton query-block forward kernel at K=4, L=4, default-configuration encoder/decoder: the original sequential order (linear), Morton Z-order sort by mean sampling position, a uniform random permutation, a scanline (row-major) sort, a Hilbert-curve sort, clustering by reference-point centroid, and a K-means clustering-and-packing (CAP) analogue of DANMP's §6.3 reordering scheme (Li et al., 2026) (argsort-dispatch on K-means labels of mean cross-level reference points). Across both GPUs, three decoder and encoder configurations (B=4 and B=8 decoder; B=2 encoder), and BF16 precision, every reordering's p_{50} latency is within ±2% of the linear baseline; the per-cell ratios and stddevs appear in Table 6 of Section A. No query-dispatch permutation tested, including the random permutation that serves as a lower bound on any structured ordering, produces a measurable change in forward-pass latency at this operating point.[1]

Mechanism. The null result has a structural explanation. The sampling locations of MSDA are the output of a learned offset network, and the reference points of each query are already monotonic in the query index (Section 2). From the memory subsystem's perspective, the sampled feature-map addresses are data-dependent and effectively random regardless of query ordering; no static permutation of the dispatch order can manufacture cache-line sharing that the kernel's tile-set does not already provide, because L2 locality on the MSDA forward kernel is set by the query-block tiling (Section 3.2), not by the global sort order of queries into blocks. The hardware L2 pre-fetcher exploits the monotonic advance of the *query index* itself (contiguous loads over sampling parameters and output buffers), and this streaming pattern is equally available under any permutation because each query's parameter block is fetched independently. The random-permutation control confirms this: a permutation that destroys all spatial structure performs identically to linear dispatch, proving that the kernel's memory traffic is query-order-invariant. Li et al. (2026) reach a compatible conclusion on a near-memory- processing substrate: MSDA's data-dependent targets defeat locality-based reuse under conventional caching.

Scope. The null scopes to *dispatch-order reordering* (permuting the global query-to-thread-block assignment, kernel held fixed); it is not a negative result for memory-access reorganization in general. The CAP analogue

[1]Per-cell relative standard deviation (σ/μ) is in the 1–3% band on the shared compute nodes (GPU clocks are not externally locked, so we exceed nvbench's 0.5% reproducibility floor); the n=100-trial mean is nevertheless discriminating to a much tighter level. Median linear-baseline σ is 0.0083 ms on A100 and 0.0074 ms on H100; with n=100 and a 95% two-sided confidence interval, the minimum detectable effect on the mean p_{50} is therefore ≈ 0.32% on A100 and ≈ 0.36% on H100, well below the ±2% band the categorical null is reported at.

Table 1: Roofline metrics for two Triton forward tilings at decoder scale, BF16, on both GPUs. "Block" denotes the query-block tiling and "Point" the point-parallel tiling; both are defined at the start of Section 3.2. Peak memory bandwidth is 1.56 TB/s (A100) and 3.35 TB/s (H100). The kernel with lower occupancy delivers higher bandwidth and lower latency.

GPU	Tiling	Occupancy (%)	L2 hit (%)	BW (% peak)	Time (ms)
A100	Point	85.2	87.3	5.1	0.612
A100	Block	17.1	34.3	36.3	0.083
H100	Point	83.3	93.2	3.8	0.356
H100	Block	14.2	35.1	18.7	0.069

tested here clusters reference centroids with fixed-k K-means and dispatches argsort-style, deviating on four axes from DANMP's full sampling-point + DBSCAN + 20%-subsample + packing recipe. A partial port that closes three of the four deviations (Section F) leaves the null intact under realistic input-dependent offsets, so the §3.1 result does not falsify the broader claim that memory-access reorganization via a packing kernel or channel-dimension remap (Xiong et al., 2024; Huang et al., 2025) could benefit structurally-related operators.

Implication. Practitioners should not invest in space-filling-curve or argsort-dispatch strategies for MSDA's forward pass at default-configuration K=4, L=4: the irregular access pattern is tile-set rather than order-set, and no query-dispatch permutation we tested, including a faithful GPU port of DANMP's clustering substep, unlocks speedup. Memory-access reorganization through a different primitive — a packing kernel that physically fuses bilinear reads sharing sub-targets, or the channel-dimension remap that DCNv4 exploits on deformable convolution — remains open for future work.

3.2 Throughput-proxy failure: high occupancy is not high throughput

The second failure concerns the metrics by which GPU kernels are usually tuned. SM occupancy (the fraction of the theoretical maximum resident warps per SM that a kernel actually sustains) is the single most widely cited kernel health indicator, and the standard optimization playbook recommends raising it until it saturates. For MSDA on the decoder configuration we identified two Triton tilings that span a wide occupancy range. We compare two Triton tilings that span a wide occupancy range. In a *point-parallel* tiling each program instance processes a single (query, head, point) triple; in a *query-block* tiling each program instance fuses all points and heads for a contiguous block of queries. The point-parallel tiling achieves a sustained SM occupancy of 85.2 % on the A100; the query-block tiling achieves only 17.1 %. By the occupancy heuristic the point-parallel tiling is preferable; in practice it is the point-parallel tiling that is a *throughput failure*.

Measurement. Table 1 summarizes the roofline metrics for both tilings on the A100 at BF16, decoder configuration. The point-parallel tiling sustains 85 % occupancy and 87 % L2 hit rate, but delivers 78.9 GB/s of DRAM bandwidth, 5.1 % of the A100 peak (1.56 TB/s). The query-block tiling runs at 17 % occupancy, 34 % L2 hit rate, and delivers 561.7 GB/s, or 36.1 % of peak, a 7.1× improvement in effective bandwidth at one-fifth of the occupancy. Forward latency follows bandwidth rather than occupancy: the point-parallel tiling takes 0.612 ms, the query-block tiling 0.083 ms, a 7.4× speedup for the kernel that looks less healthy on the occupancy metric. The H100 roofline is qualitatively identical (Table 1).

Mechanism. The point-parallel tiling satisfies every condition the occupancy heuristic is designed to detect (warps resident, scheduler slots filled, high L2 hit rate because each thread touches four bilinear neighbours) but issues those loads as tiny independent transactions, one per program, and the memory pipeline fills with many small in-flight requests that cannot be coalesced post hoc. The query-block tiling fetches sampling parameters once per block, broadcasts coordinates across the block's threads, and issues coalesced wide bilinear loads; resident warp count drops because the per-program register footprint is larger, but each request moves more useful bytes. Hardware-counter warp-stall attribution confirms the mechanism directly: 67.2 % of active warp-issue slots on the point-parallel A100 kernel stall on *long-scoreboard* dependencies, the stall

Table 2: Roofline metrics for the Triton backward kernel at decoder scale (B=4, 800×1333, BF16/FP32). At FP32 both GPUs sustain 26 % to 29 % of peak bandwidth. At BF16 the A100 collapses to 2.4 % while the H100 holds at 21.3 %, an order-of-magnitude gap at matched code.

GPU	Precision	BW (GB/s)	BW (% peak)	Time (ms)
A100	FP32	401.4	25.8	0.791
A100	BF16	38.5	2.5	3.997
H100	FP32	984.4	29.4	0.292
H100	BF16	714.2	21.3	0.190

class that indicates a warp waiting for an outstanding DRAM load. High occupancy does not hide DRAM latency if every warp is waiting for a different cache miss.

Implication. For memory-bound operators with irregular access patterns, the binding constraint is the DRAM transaction size distribution, not warp availability, and the optimization that raises occupancy may lower bandwidth. Although the low-occupancy-beats-high-occupancy principle is well established for general GPU kernels (Volkov, 2010; Shobaki et al., 2020) and for dense attention (Spector et al., 2025), ours is the first quantitative demonstration of the divergence for a scattered-access attention operator. It is the reason we settled on query-block tiling for the forward pass even though no conventional tuning metric preferred it.

3.3 Atomic contention: hardware-gated under native precision

The forward-pass throughput-proxy failure has a backward-pass counterpart that is larger in magnitude, more consequential for training, and caused by a different mechanism. The MSDA backward pass accumulates gradients at the sampled feature positions using atomic adds: every query that samples a given neighbour contributes additively to that neighbour's gradient, and because sampling locations are data-dependent the set of contending queries is not known until runtime. Under dense workloads (encoder-scale 1536×2048 feature maps with hundreds of queries per point), the same cache line is updated by many warps within microseconds of each other. The performance of these atomics depends on hardware support. Before SM 9.0 (Hopper), the GPU had no native BF16 atomic-add instruction; the CUDA runtime emulated it with a multi-step compare-and-swap (CAS) loop that retries on every concurrent modification (Section 4.2 confirms this empirically: on Hopper the compiled kernel reduces to a single hardware atomic; on Ampere the same source compiles to an approximately ten-instruction CAS retry loop). SM 9.0 adds a single-transaction hardware primitive for BF16 atomics. Separately, Triton exposes a relaxed memory-ordering qualifier on its atomic-addition intrinsic, weakening the default acquire-release semantics to permit the memory subsystem to coalesce atomics more aggressively.

Measurement. Our Triton backward kernel at decoder scale on the A100 sustains 401 GB/s (25.8 % of peak) at FP32 and 38.5 GB/s (2.4 % of peak) at BF16, a drop of ∼11× in effective bandwidth purely from switching the accumulation precision from FP32 to BF16 on the same kernel, same GPU, and same configuration (Table 2). At encoder scale the collapse translates directly into wall-clock time. The same Triton backward kernel takes 123.4 ms to process one encoder batch on the A100 at BF16. The CUDA backward kernel sidesteps the BF16 atomic contention regime entirely by upcasting gradient tensors to FP32 in the Python wrapper before launching the kernel, and takes 17.2 ms on the same input — a 7× gap. On the H100 the same comparison inverts entirely: the Triton kernel completes in 4.33 ms and the CUDA kernel in 10.19 ms (2.4× the other direction). Neither kernel was modified between GPUs; the code path is bit-identical.

Mechanism. The root cause is a single hardware capability that A100 (SM 8.0) lacks and H100 (SM 9.0) supplies: a native BF16 atomic-add instruction. On SM 8.0 the runtime emulates a BF16 atomic-add as a compare-and-swap (CAS) loop (NVIDIA Corporation, 2025a); under MSDA's scattered gradient pattern, where dozens of warps write to the same cache line in a small time window, retry serialization dominates. The CUDA backward kernel sidesteps the loop by upcasting inputs and the upstream gradient to FP32

in the host-side wrapper, then issuing hardware-native FP32 atomics and down-converting on return. A control backend that disables the upcast and forces native-BF16 atomic-add lands at 1.706 ms / 94 GB/s on A100 decoder BF16, splitting the 5.3× CUDA/Triton decoder gap roughly evenly between the FP32-upcast precision rescue and the nvcc-vs-Triton codegen advantage within BF16 atomics. SM 9.0 promotes the BF16 atomic to a hardware primitive (`atom.add.noftz.bf16` (NVIDIA Corporation, 2025b; 2022)) that completes in a single transaction regardless of contention, and the gap closes (Section 4.2 refines the mechanism: the gap is hardware-*and*-compiler-gated, requiring Triton's specific atomic lowering path).

Software mitigations. The bandwidth collapse is gated only when backward atomics operate at native BF16 precision; accumulating gradients into an FP32 scratch buffer (the *FP32-accumulator* variant) replaces the contended BF16 CAS loop with a hardware-native FP32 atomic-add at the cost of a larger buffer. At encoder scale on A100 it runs 18.2 ms, 6.8× faster than the native-precision Triton backward (123.4 ms) and within 6 % of our CUDA backward (17.2 ms) at 45 % of its peak workspace; at decoder scale (B=8, BF16) it runs 1.54 ms, 33 % faster than native Triton and 7 % faster than CUDA. On H100, where hardware BF16 atomics are already fast, the path regresses 18 % at encoder scale (5.1 ms vs 4.3 ms). The FP32 buffer is transparent to PyTorch's automatic-mixed-precision context, preserves numerical parity with the native-precision backend (Table 12), and raises encoder backward peak VRAM from 448 MB to 576 MB (+29%), still 45 % below the CUDA workspace. Three reduce-by-key alternatives (warp, block, and sort/segscan, all numerically matched to the baseline) fail: on A100 decoder BF16 they run 2.8–12.9× slower than the plain atomic (5.56 ms/22.59 ms/4.85 ms versus 1.75 ms); on H100 1.5–6.1× slower. The software-dedup structures cost more than the contention they eliminate even on SM 8.0. The only software lever that pays at MSDA's contention regime is accumulator precision, not pre-reduction.

Implication. The three failure modes are mechanistically distinct. Locality is a property of the operator's access pattern; throughput-proxy is a property of the tuning metric; atomic contention is a joint property of target architecture and accumulator precision. Under native-precision BF16 atomics the backward gap is an order of magnitude and resolved only on SM 9.0; under FP32 accumulation it closes on both GPU generations. The practical consequence is that the backward-pass choice is an *accumulator-precision* decision gated by the target hardware: use FP32 accumulators on A100 and native BF16 atomics on H100; Triton is the fastest or tied backend in both cases (Section 4.1). Both paths use relaxed memory ordering, which is safe because gradient accumulation is commutative and the kernel-end barrier provides the required happens-before relationship (NVIDIA Corporation, 2025a). No prior attention-kernel study isolates this effect, and it independently validates concurrent atomic-reduction work for differentiable rendering (Durvasula et al., 2025) on a second operator class.

4 Implications and Generalization

4.1 Backward-pass programming model: an accumulator-precision question

Which backend should a downstream user pick on which GPU? Section 3.3 established that the backward-pass atomic contention is hardware-gated under native-precision atomics, and that an FP32-accumulator variant closes the gap on A100. This subsection turns those findings into wall-clock guidance for the two programming models we implemented.

Forward pass. On both A100 and H100, the Triton forward kernel using query-block tiling matches or slightly beats the best CUDA forward kernel at the standard decoder configuration. At B=4, BF16, decoder scale the Triton forward latency is 0.534 ms on A100 versus 0.564 ms for the corresponding CUDA kernel, and 0.445 ms versus 0.473 ms on H100 (5 %–6 % Triton advantage on both GPUs). The advantage comes from the Triton compiler's better exploitation of the query-block tiling described in Section 3.2: the same technique is available in CUDA but requires hand-written warp-broadcast code that is harder to tune. For the forward pass the programming-model choice is therefore a minor latency difference; both models have access to the same bandwidth, and both deliver it.

Table 3: Backward-pass latency at encoder scale (B=2, 1536×2048, L=4, BF16) for both programming models on both GPUs. Under native-precision atomics, CUDA wins on A100 by 7× and Triton wins on H100 by 2.4×. The FP32-accumulator Triton variant closes the A100 gap, making Triton the fastest or tied backend on both GPUs. Neither kernel was modified between GPUs.

Backend	A100 bwd (ms)	H100 bwd (ms)
Reference impl.	692.3	49.0
CUDA path	17.2	10.2
Triton path (BF16 acc)	123.4	4.33
Triton path (FP32 acc)	18.2	5.1

Backward pass, decoder scale. At small batch sizes the backward pass does not saturate the atomic pipeline, and the Triton and CUDA backends are within single-digit percent of each other on both GPUs. At B=8 the picture changes. On the A100 the CUDA backward is 1.581 ms and the Triton backward is 2.043 ms: CUDA is 29 % faster. On the H100 the same kernels run in 1.195 ms and 1.177 ms respectively: Triton is 1.5 % faster. The sign flip is produced by moving to hardware that has a native BF16 atomic-add; no kernel code changes between the two runs.

Backward pass, encoder scale. At encoder scale the native-precision backward gap is 7× rather than 30 %, and the sign is the same as the decoder B=8 result. On the A100 the CUDA backward processes a B=2, 1536×2048 encoder batch in 17.2 ms while the native-precision Triton backward takes 123.4 ms; on the H100 the same two kernels run in 10.2 ms and 4.33 ms, respectively. The FP32-accumulator Triton variant (Section 3.3) closes the A100 gap on latency to within 6 % of CUDA (18.2 ms vs 17.2 ms) while using 45 % less peak VRAM (Table 5). Table 3 presents all three backends on both GPUs. Two observations follow. First, under native-precision atomics neither kernel is universally preferable, but the FP32-accumulator variant resolves the crossover in favor of Triton on both GPUs. Second, the native-precision gap is dominated entirely by the atomic-hardware discontinuity (Section 3.3): at FP32, where the SM 8.0 software CAS loop is not in the path, the Triton backward is within a factor of two of the CUDA backward on the A100 even at encoder scale.

4.2 Generalization beyond MSDA: the gap is compiler-gated

The backward gap reported in Section 3.3 is hardware-*and*-compiler-gated: Triton's relaxed-semantics atomic reaches Hopper's native BF16 reduction primitive, while nvcc's half-precision atomic-add overload does not. The gap should therefore not reproduce on PyTorch-native operators compiled through nvcc. We test this on PyTorch's standard scatter-add operator, the only PyTorch-native scattered reduction whose dispatch chain bottoms out in the same BF16 atomic-add call as our CUDA kernel; the embedding-lookup and index-add backward paths instead go through sort-based reductions with no atomics.

Sweeping the scatter-add operator across a 72-cell matrix of (M, D, N) sizes, index distributions (uniform and Zipf-s=1.5), and precisions (FP32/FP16/BF16) on both GPUs reveals two properties: *FP32 is indifferent to contention* (p_{50} stays under 2 ms in every cell), and *every half-precision atomic collapses two-to-three orders of magnitude under Zipf on both architectures*, including native FP16 (1499 ms / 1823 ms at the worst-contention cell). The full 72-cell sweep is in Section C, supported by Tables 8 and 9. The A100→H100 BF16 ratio for the scatter-add operator at the worst cell is order-unity (0.9–1.5×), far below the reproducible 7× gap we measure on MSDA's backward at the same precision. The hardware-gated claim, as initially stated, does not reproduce on the PyTorch-native operator.

Counter-level diagnosis. Hardware-counter instrumentation at the worst-contention cell locates the divergence. The MSDA backward kernel on H100 emits `red.*`-type sectors (the hardware reduction path), the only quadrant of the table that does; A100 at BF16, and both GPUs for the scatter-add operator, stay on `atom.*` (full counters in Table 9). The mechanism has three components. (i) Hardware: shell-level SASS disassembly of our Triton-compiled H100 kernel shows the atomic gradient writes lower to `REDG.E.ADD.BF16x8.RN.STRONG.GPU`, a single instruction that reduces eight BF16 lanes at once through

Table 4: MSDA operator latency (p_{50}, ms) and speedup over the reference on the same GPU. Config: B4 800×1333, BF16, eager. Best latency per column in **bold**.

	A100				H100			
	Fwd (ms)	Fwd×	Bwd (ms)	Bwd×	Fwd (ms)	Fwd×	Bwd (ms)	Bwd×
Reference	1.416	1.00×	5.440	1.00×	1.053	1.00×	2.142	1.00×
CUDA	0.564	2.53×	**1.522**	3.56×	0.472	2.26×	1.206	1.78×
Triton, point-parallel	0.561	2.55×	1.777	3.07×	**0.441**	2.42×	**1.160**	1.84×
Triton, query-block	**0.534**	2.68×	1.572	3.46×	0.445	2.39×	1.187	1.80×

the L2 cache; Ampere SASS has no equivalent, because SM 8.0 lacks the native BF16 atomic primitive that PTX ISA 7.8 introduces for SM 9.0 and later (NVIDIA Corporation, 2025b). (ii) Triton codegen: the atomic-addition intrinsic in our backward kernel is invoked with relaxed memory-ordering semantics (Triton Project, 2024) (the intrinsic defaults to acquire-release, so the relaxed lowering this paper relies on requires that the explicit qualifier be supplied at the call site). With an unused return, the intrinsic emits an eight-register vectorized PTX atomic-add at BF16 (`atom.global.gpu.relaxed.add.noftz.v8.bf16`) whose return is dead, which the PTX assembler promotes to the hardware reduction form wherever supported. (iii) nvcc codegen: the standard CUDA half-precision atomic-add overload (as shipped with CUDA 12.8) lowers to `ATOM.E.CAS.STRONG.GPU` even when the return value is discarded, empirically confirmed by the zero `red` sector counts for the scatter-add operator. The CUDA-library specialization that selects the native lowering is toolkit-version-dependent; we report behaviour observed under CUDA 12.8, with later toolkits potentially closing this gap as the standard CUDA half-precision atomic-add overload evolves. Consumers whose compiler marks the atomic as a reducible no-return operation inherit the H100 advantage; consumers that bottom out in the standard half-precision atomic-add overload do not. The practical recommendation therefore stands: on A100, use an FP32-accumulator backward (Section 3.3); on H100, Triton kernels with relaxed-ordering atomics need no workaround, while nvcc-compiled consumers may still benefit until the CUDA runtime adds the `red.bf16` lowering.

5 Experimental Evaluation

Setup. All measurements run on a single-GPU partition of a shared academic HPC cluster on two nodes: an A100-SXM4-40GB (SM 8.0) and an H100 SXM (SM 9.0), both under CUDA 12.8, PyTorch 2.9.0, and the matching Triton release. Unless noted, we use eager mode and the standard Deformable DETR decoder configuration: batch size B=4, input resolution 800×1333, hidden dimension D=256, and L=4 feature levels. Latencies are p_{50} over 100 iterations after 10 warmup iterations with explicit device synchronisation barriers; standard deviations and tail statistics are reported in Section A. We compare four implementations: the reference of Zhu et al. (2021), our optimized CUDA kernel, the high-occupancy point-parallel Triton tiling, and our query-block Triton tiling. Full benchmark parameters, the encoder-scale configuration, a cumulative technique ablation, and per-GPU sweep tables are all deferred to Section A. We report absolute latencies alongside every ratio, since the reference backward benefits disproportionately from the A100→H100 upgrade ($\sim$2.5×) while our backends benefit by $\sim$1.3× in the same comparison (already closer to the bandwidth ceiling on A100); read as ratios, the H100 speedups therefore appear smaller than the A100 ones even though absolute latencies are lower on H100.

Operator-level latency (Table 4). Our Triton query-block tiling is the fastest forward kernel on both GPUs at the decoder scale (0.534 ms on A100, 0.445 ms on H100, corresponding to 2.68×/2.39× over the reference). At decoder scale the backward-pass latencies of three of the four custom backends (our CUDA kernel, our query-block Triton kernel, and its FP32-accumulator variant) converge to within 5 % of each other on A100 and 4 % on H100 because the atomic pipeline is not saturated; the Triton point-parallel tiling is 19 % slower on A100, consistent with the throughput-proxy failure of Section 3.2. The architectural backward crossover appears only at encoder scale (Table 3). Encoder-scale forward speedup reaches 12.18× on A100 and 14.00× on H100, again driven by the query-block tiling of Section 3.2.

Table 5: Peak GPU memory (MB) per backend. A100-SXM4-40GB, BF16, B4 800×1333, eager. Best peak per phase in **bold**.

	Fwd only		Fwd+Bwd	
	Alloc (MB)	Peak (MB)	Alloc (MB)	Peak (MB)
Reference	54.7	117.9	0.6	**151.7**
CUDA	45.0	**45.9**	0.6	221.7
Triton, point-parallel	45.0	**45.9**	0.6	**151.7**
Triton, query-block	45.0	**45.9**	0.6	**151.7**

Memory footprint. All custom backends cut the decoder forward-only peak from 117.9 MB to 45.9 MB (−61%) by eliminating the sampling-parameter concatenation buffer of Section 2.3. At encoder scale the combined forward + backward peak drops from 3607.8 MB for the reference (FP32 accumulator) to 1055.9 MB for our CUDA path (FP32 accumulator), to 575.9 MB for the Triton FP32-accumulator variant, and to 447.9 MB for the native-BF16 Triton path (−88% vs reference), enabling encoder-scale backward at batch sizes that would otherwise require gradient checkpointing. The deltas trace a three-operating-point Pareto frontier between accumulator precision and peak footprint: kernel fusion of the per-level interpolation chain (which the graph-recorded reference materialises end-to-end) governs the reference-to-CUDA step, while the accumulator-dtype choice governs the CUDA-to-Triton step (Section 3.3). Per-allocation memory profiling on the A100 partition at the same encoder cell confirms the cross-backend deltas: 1334 MB for the CUDA backend, 663 MB for our query-block Triton backend (a 671 MB kernel-driven saving), and a 129 MB addition for the FP32-accumulator variant (toggling the standard linear-algebra workspace allocator shifts peaks by <3 MB, ruling out caching-allocator artefacts). Table 5 reports the decoder decomposition.

Roofline analysis (Figure 1). Every backend on every GPU is memory-bandwidth-bound: the highest AI we measure is 15.1 FLOP/byte (Triton point-parallel backward at FP32 on H100), below the H100 FP32 ridge of ∼17.9 FLOP/byte. FP32 bandwidth utilization is broadly consistent across backends (26–41% of peak on A100, 19–30% on H100), but at BF16 the A100 Triton query-block backward collapses to 2.5 % of peak while the CUDA path holds 23.1 %; on H100 both backends recover above 19 %. This is the direct Roofline signature of the hardware-gated analysis of Section 3.3.

Technique ablation. A cumulative technique ablation at encoder scale (Table 13) isolates the kernel-design contributions: on the CUDA path the read-side optimizations (vectorized loads, warp-broadcast coordinates) deliver the majority of the 8× forward improvement, while the write-side relaxed atomics add 7 %; the tiling strategy is the dominant Triton lever (8.2× forward).

6 Related Work

Deformable operators. We take the MSDA algorithm as given and ask a systems question about it. The algorithm traces back to DCN's learnable 2D offsets (Dai et al., 2017; Zhu et al., 2019; Wang et al., 2023), which Zhu et al. (2021) transposed to attention, replacing DETR's dense $O(N^2)$ matmul (Carion et al., 2020) with a small fixed number of bilinearly sampled points per query; MSDA is now a core component of DINO (Zhang et al., 2023) and Mask2Former (Cheng et al., 2022). The closest systems-level peer is DCNv4 (Xiong et al., 2024), which reports 3× over DCNv3 through vectorized 128-bit loads, thread-to-channel remapping within a fixed 3×3 deformable-convolution footprint, and half-precision arithmetic (DCNv4 Table 10 indicates that the remapping in isolation is mildly anti-helpful; the speedup materialises only with vectorisation and FP16 together). DCNv4 §3.2 states that these optimizations also apply to deformable attention, but the published benchmarks are on deformable convolution rather than MSDA; we are unaware of a Hopper MSDA benchmark using DCNv4's optimizations. Its optimization axis is channel-within-group remapping plus vectorized loads; ours, in Section 3.1, is query-dispatch order on a fixed query-block tiling. Along that narrower axis the two are orthogonal.

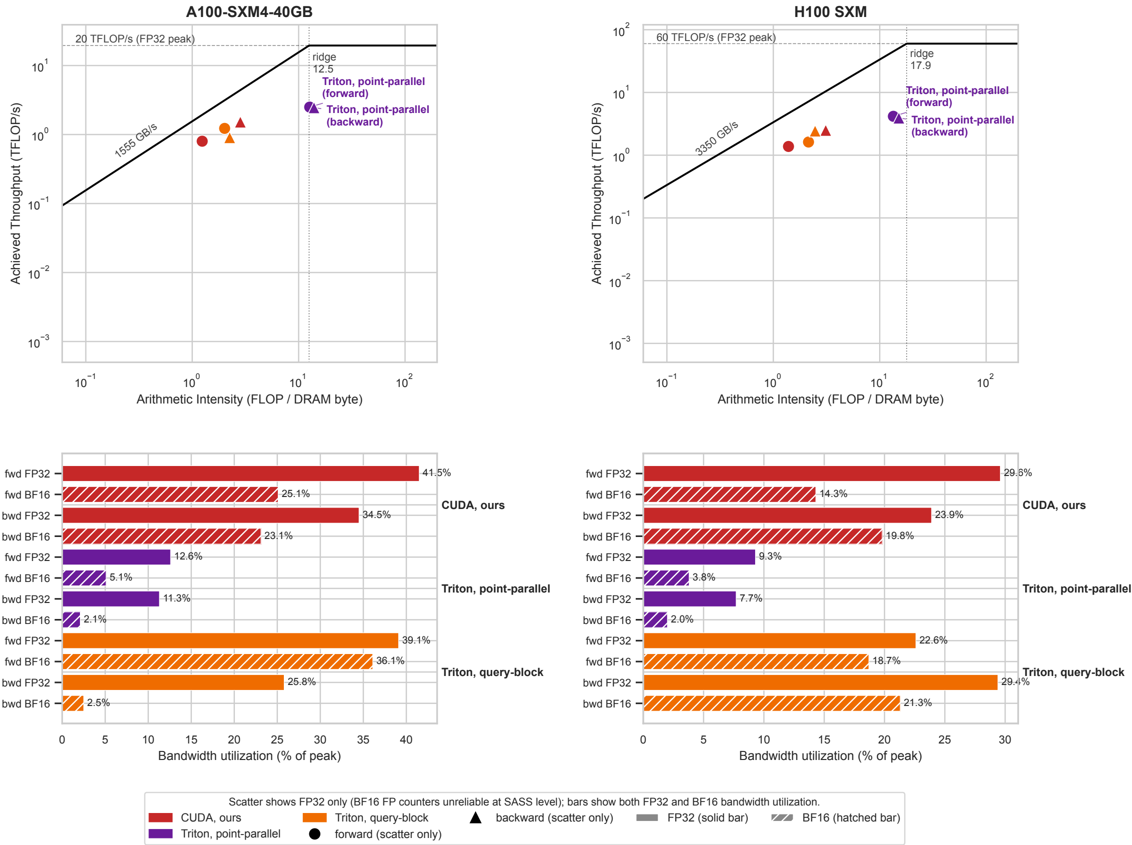

Figure 1: Roofline and bandwidth-utilization summary at decoder scale. *Top:* FP32 arithmetic intensity versus sustained TFLOP/s; all backends fall below the ridge, confirming that MSDA is memory-bandwidth-bound on both GPUs. *Bottom:* bandwidth utilization (% of peak) at FP32 and BF16. The BF16 backward collapse on A100 is the atomic-contention effect of Section 3.3. BF16 arithmetic intensity is omitted because the SASS floating-point counters are unreliable on SM 8.0 at BF16.

Dense attention kernels. FlashAttention (Dao, 2024; Shah et al., 2024) and FlashInfer (Ye et al., 2025) achieve substantial speedups on dense self-attention by tiling $QK^\top V$ to keep the attention matrix in SRAM; FlashAttention-3 adds Hopper warp specialization. FlexAttention (Dong et al., 2024) generalizes tiled attention to user-defined score modifications. None of these transfer to MSDA, whose per-query learned offsets vary the LK sampled positions and preclude the regular tile structure these kernels exploit. The fundamental difference is access-pattern, not algorithmic: dense matmul attention becomes compute-bound on large N; MSDA is memory-bandwidth-bound throughout (Figure 1) with cost dominated by scattered irregular reads.

MSDA-specific acceleration. DEFA (Xu et al., 2024) combines algorithm-level pruning with a custom ASIC/FPGA for 10–32× speedup; its GPU-side profiling independently confirms the memory-boundedness diagnosis (multi-scale sampling and aggregation account for over 60 % of MSDA inference time on an RTX 3090 Ti despite only 3.25 % of FLOPs). Huang et al. (2025) achieve 5.9× forward / 8.9× backward on Huawei Ascend NPUs via staggered gradient computation, complementary to our relaxed-ordering atomics. DANMP (Li et al., 2026) places compute units adjacent to HBM banks for 97× over an A6000 baseline. Their GPU-side profiling (DANMP §3.1, RTX A6000) reports the MSDA cache hit rate declining monotonically as batch size grows and attributes the effect to data-dependent targets across batches preventing conventional locality-based reuse. DANMP also proposes a software-side *Clustering-and-Packing* (CAP) reordering scheme (§6.3) that achieves a 1.45× speedup on CPU; CAP is not evaluated on GPU in that work, a gap we address

with a direct GPU analogue in Section 3.1. UEDA (Sun et al., 2025) and QUILL (Oh et al., 2026) are ASICs for deformable-attention variants; QUILL's dynamic query-order optimization overlaps reordering with compute in hardware where our null result on commodity GPUs would otherwise apply. We target NVIDIA CUDA and Triton, provide a cross-paradigm comparison across two GPU generations, and document a locality negative result specific to the GPU execution model.

Locality reordering in sparse operators. Our Morton Z-order null (Section 3.1) echoes mixed GNN experience with vertex reordering (Merkel et al., 2025; Wang et al., 2021) and the structured-pattern conclusion of Yuan et al. (2025) for LLM sparse attention; space-filling-curve reordering *does* succeed when the tokens inhabit a fixed geometric substrate (HilbertA (Zheng et al., 2025) on diffusion transformers, Point Transformer V3 (Wu et al., 2024) on point clouds), a precondition MSDA's learned, per-iteration sampling does not meet.

Atomic optimization. Concurrent work on differentiable rendering reduces atomic contention through warp-level reduction primitives (Durvasula et al., 2024; 2025; Mallick et al., 2024) and queueing-theoretic modelling (Dong & Pai, 2025); Section 3.3 complements these by quantifying the magnitude on MSDA and locating the architectural close at the SM 8.0→SM 9.0 boundary.

The Triton ecosystem. Triton (Tillet et al., 2019) is widely adopted for LLM inference and dense attention (Dao et al., 2022; Lefaudeux et al., 2022; Hsu et al., 2025), all of which exploit contiguous $QK^\top V$ tile structure that MSDA lacks; our scattered-access comparison finds an architecture-dependent verdict (Triton wins the forward pass via query-dimension tiling; the backward winner is gated by the target atomic hardware) rather than a paradigm-dependent one.

7 Discussion and Conclusion

We characterized how scattered-access attention operators interact with the conventional GPU optimization toolkit and found two failure modes and one hardware-gated root cause. Dispatch-order reordering does not pay for MSDA at $K{=}4$, $L{=}4$: seven query-dispatch orders, including a faithful GPU port of DANMP's K-means clustering substep, produce within ±2% of linear because L2 locality is tile-set by the query-block kernel, not by the global dispatch order (Section 3.1). High occupancy is not high throughput either: an 85 %-occupancy tiling delivers 5.1 % of A100 peak bandwidth where a 17 %-occupancy tiling achieves 36 % and runs 7.4× faster (Section 3.2). The backward BF16 kernel then collapses to 2.4 % of A100 peak bandwidth because SM 8.0 lacks a native BF16 atomic-add primitive and the CUDA runtime emits a CAS loop, while the same code recovers to 21.3 % on H100 (Section 3.3). An FP32-accumulator variant closes the A100 gap, making the backward-pass choice an *accumulator-precision* decision gated by target hardware (Section 3.3 and Table 3).

Lessons for practitioners. Four scoped hypotheses follow, to be tested on other scattered-access operators: (i) do not sort the query dispatch order by space-filling curves without a tile-set exploitability check; (ii) do not equate high occupancy with high throughput on memory-bound operators, tuning instead for register reuse and wide DRAM transactions; (iii) avoid native half-precision atomic addition on architectures predating dedicated reduction hardware, substituting FP32 accumulators or compiler-controlled relaxed atomics; (iv) profile DRAM bandwidth, not FLOPs, as the roofline axis for memory-bound operators. All three failure modes motivating (i)–(iii) were invisible to FLOP-based metrics.

Limitations and future work. We benchmark two NVIDIA generations; AMD and Intel hardware would need analogous profiling. TensorRT's deformable-attention plugin is inference-only, precluding a training-time comparison, and we do not report end-to-end training curves; we have, however, verified inference-time task parity by swapping our backends into released Deformable DETR and Mask2Former checkpoints and reproducing their reported COCO, Cityscapes, and ADE20K metrics within rounding (no end-to-end retraining). The FP32-accumulator mitigation trades a 29 % larger backward buffer for faster atomics; a future driver or SRAM-based Triton backward could narrow the gap without the memory cost. We will release our implementation upon acceptance.

References

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020. arXiv:2005.12872.

Niccolò Cavagnero, Gabriele Rosi, Claudia Cuttano, Francesca Pistilli, Marco Ciccone, Giuseppe Averta, and Fabio Cermelli. PEM: Prototype-based efficient MaskFormer for image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 15804–15813, 2024.

Bowen Cheng, Ishan Misra, Alexander G. Schwing, Alexander Kirillov, and Rohit Girdhar. Masked-attention mask transformer for universal image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. arXiv:2112.01527.

Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017. arXiv:1703.06211.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2307.08691.

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)*, 2022. Triton reference implementation included in repository.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Rongcui Dong and Sreepathi Pai. Modeling utilization to identify shared-memory atomic bottlenecks. In *Proceedings of the 17th Workshop on General Purpose Processing Using GPU (GPGPU)*, 2025. arXiv:2503.17893.

Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. DISTWAR: Fast differentiable rendering on raster-based rendering pipelines. *arXiv preprint arXiv:2401.05345*, 2024.

Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, Yushi Guan, Christina Giannoula, and Nandita Vijaykumar. ARC: Warp-level adaptive atomic reduction in GPUs to accelerate differentiable rendering. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient Triton kernels for LLM training. *arXiv preprint arXiv:2410.10989*, 2025. ICML 2025 Workshop.

Chenghuan Huang, Zhigeng Xu, Chong Sun, Chen Li, and Ziyang Ma. Towards efficient multi-scale deformable attention on NPU. *arXiv preprint arXiv:2505.14022*, 2025.

Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library, 2022. `https://github.com/facebookresearch/xformers`.

Huize Li, Qinggang Wang, Bin Gao, Dan Chen, Yu Huang, and Xin Xin. Accelerating multi-scale deformable attention using near-memory-processing architecture. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2026. arXiv:2603.00959.

Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Francisco Vicente Carrasco, Markus Steinberger, and Fernando De La Torre. Taming 3DGS: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers*, 2024. arXiv:2406.15643.

Nikolai Merkel, Pierre Toussing, Ruben Mayer, and Hans-Arno Jacobsen. Can graph reordering speed up graph neural network training? An experimental study. *Proceedings of the VLDB Endowment*, 18(2): 293–307, 2025. doi: 10.14778/3705829.3705846.

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

NVIDIA Corporation. *CUDA C++ Programming Guide*, 2025a. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

NVIDIA Corporation. *Parallel Thread Execution ISA Version 8.7*, 2025b. `https://docs.nvidia.com/cuda/archive/12.8.0/pdf/ptx_isa_8.7.pdf`.

Hyunwoo Oh, Hanning Chen, Sanggeon Yun, Yang Ni, Wenjun Huang, Tamoghno Das, Suyeon Jang, and Mohsen Imani. QUILL: An algorithm-architecture co-design for cache-local deformable attention. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2026. arXiv:2511.13679.

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 (NeurIPS)*, 2024. arXiv:2407.08608.

Ghassan Shobaki, Austin Kerbow, and Stanislav Mekhanoshin. Optimizing occupancy and ILP on the GPU using a combinatorial approach. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, pp. 133–144, 2020.

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. ThunderKittens: Simple, fast, and adorable AI kernels. In *International Conference on Learning Representations (ICLR)*, 2025. Spotlight, arXiv:2410.20399.

Kairui Sun, Meiqi Wang, Junhai Zhou, and Zhongfeng Wang. UEDA: A universal and efficient deformable attention accelerator for various vision tasks. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 163–169, 2025.

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

Triton Project. `triton.language.atomic_add` — language reference, 2024. `https://triton-lang.org/main/python-api/generated/triton.language.atomic_add.html`.

Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference (GTC)*, 2010. Presentation, NVIDIA GTC 2010, `https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf`.

Wenhai Wang, Jifeng Dai, Zhe Chen, Zhenhang Huang, Zhiqi Li, Xizhou Zhu, Xiaowei Hu, Tong Lu, Lewei Lu, Hongsheng Li, Xiaogang Wang, and Yu Qiao. InternImage: Exploring large-scale vision foundation models with deformable convolutions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023. arXiv:2211.05778.

Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

Xiaoyang Wu, Li Jiang, Peng-Shuai Wang, Zhijian Liu, Xihui Liu, Yu Qiao, Wanli Ouyang, Tong He, and Hengshuang Zhao. Point transformer V3: Simpler, faster, stronger. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Yuwen Xiong, Zhiqi Li, Yuntao Chen, Feng Wang, Xizhou Zhu, Jiapeng Luo, Wenhai Wang, Tong Lu, Hongsheng Li, Yu Qiao, Lewei Lu, Jie Zhou, and Jifeng Dai. Efficient deformable ConvNets: Rethinking dynamic and sparse operator for vision applications. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. arXiv:2401.06197.

Yansong Xu, Dongxu Lyu, Zhenyu Li, Yuzhou Chen, Zilong Wang, Gang Wang, Zhican Wang, Haomin Li, and Guanghui He. DEFA: Efficient deformable attention acceleration via pruning-assisted grid-sampling and multi-scale parallel processing. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024. arXiv:2403.10913.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025. arXiv:2501.01005.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 23078–23097, 2025. doi: 10.18653/v1/2025.acl-long.1126.

Hao Zhang, Feng Li, Shilong Liu, Lei Zhang, Hang Su, Jun Zhu, Lionel M. Ni, and Heung-Yeung Shum. DINO: DETR with improved denoising anchor boxes for end-to-end object detection. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2203.03605.

Yian Zhao, Wenyu Lv, Shangliang Xu, Jinman Wei, Guanzhong Wang, Qingqing Dang, Yi Liu, and Jie Chen. DETRs beat YOLOs on real-time object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. arXiv:2304.08069.

Shaoyi Zheng, Wenbo Lu, Yuxuan Xia, Haomin Liu, and Shengjie Wang. HilbertA: Hilbert attention for image generation with diffusion models, 2025. arXiv:2509.26538.

Xizhou Zhu, Han Hu, Stephen Lin, and Jifeng Dai. Deformable ConvNets v2: More deformable, better results. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. arXiv:1811.11168.

Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable DETR: Deformable transformers for end-to-end object detection. In *International Conference on Learning Representations (ICLR)*, 2021. arXiv:2010.04159.

A Full Benchmark Tables

Table 6 reports the full per-cell dispatch-order measurements summarized in Section 3.1: p_{50} latency, per-cell standard deviation σ, relative standard deviation σ/μ, and ratio r vs. the (config, dtype) linear baseline, for all eight dispatch orders across the three decoder batch sizes and three precisions on both GPUs ($n = 100$ trials per cell). Cells exceeding nvbench's 0.5% relative- standard-deviation reproducibility floor are flagged with †. The shared compute nodes used in our measurements are not externally clock-locked, so per-trial σ/μ is typically in the 1–3% band rather than the 0.5% nvbench-locked floor; the $n = 100$-trial mean is nevertheless discriminating to a much tighter level (see methods footnote in Section 3.1).

Table 6: Per-cell locality-controls measurements (mean ± std, $n = 100$ trials each). r is the ratio of the cell's p_{50} latency to its (config, dtype) linear baseline. The σ/μ column reports relative standard deviation; cells exceeding nvbench's 0.5% reproducibility floor are marked with †.

GPU	Config	B	Order	p_{50} (ms)	σ (ms)	σ/μ	r
A100	B1/FP32	1	`linear`	0.4932	0.0110	2.23%†	1.0000
A100	B1/FP32	1	`morton`	0.4931	0.0082	1.66%†	0.9998
A100	B1/FP32	1	`random`	0.4918	0.0085	1.73%†	0.9972

(continued on next page)

(Table 6 continued from previous page.)

GPU	Config	B	Order	p_{50} (ms)	σ (ms)	σ/μ	r
A100	B1/FP32	1	`scanline`	0.4889	0.0080	1.64%†	0.9913
A100	B1/FP32	1	`centroid`	0.4918	0.0066	1.34%†	0.9972
A100	B1/FP32	1	`hilbert`	0.4924	0.0069	1.40%†	0.9984
A100	B1/FP32	1	`kmeans_cap`	0.4948	0.0111	2.24%†	1.0032
A100	B1/FP32	1	`danmp_partial`	0.4823	0.0146	3.03%†	0.9779
A100	B1/FP16	1	`linear`	0.5276	0.0077	1.46%†	1.0000
A100	B1/FP16	1	`morton`	0.5263	0.0059	1.12%†	0.9975
A100	B1/FP16	1	`random`	0.5260	0.0082	1.56%†	0.9970
A100	B1/FP16	1	`scanline`	0.5268	0.0060	1.14%†	0.9985
A100	B1/FP16	1	`centroid`	0.5271	0.0064	1.21%†	0.9991
A100	B1/FP16	1	`hilbert`	0.5259	0.0060	1.14%†	0.9968
A100	B1/FP16	1	`kmeans_cap`	0.5313	0.0067	1.26%†	1.0070
A100	B1/FP16	1	`danmp_partial`	0.5161	0.0071	1.38%†	0.9782
A100	B1/BF16	1	`linear`	0.5012	0.0185	3.69%†	1.0000
A100	B1/BF16	1	`morton`	0.5020	0.0064	1.27%†	1.0016
A100	B1/BF16	1	`random`	0.4994	0.0061	1.22%†	0.9964
A100	B1/BF16	1	`scanline`	0.5011	0.0073	1.46%†	0.9998
A100	B1/BF16	1	`centroid`	0.5003	0.0083	1.66%†	0.9982
A100	B1/BF16	1	`hilbert`	0.5008	0.0068	1.36%†	0.9992
A100	B1/BF16	1	`kmeans_cap`	0.5049	0.0077	1.53%†	1.0074
A100	B1/BF16	1	`danmp_partial`	0.4895	0.0067	1.37%†	0.9767
A100	B2/FP32	2	`linear`	0.5006	0.0105	2.10%†	1.0000
A100	B2/FP32	2	`morton`	0.5036	0.0103	2.05%†	1.0060
A100	B2/FP32	2	`random`	0.5062	0.0104	2.05%†	1.0112
A100	B2/FP32	2	`scanline`	0.5000	0.0087	1.74%†	0.9988
A100	B2/FP32	2	`centroid`	0.5020	0.0088	1.75%†	1.0028
A100	B2/FP32	2	`hilbert`	0.4997	0.0086	1.72%†	0.9982
A100	B2/FP32	2	`kmeans_cap`	0.5054	0.0128	2.53%†	1.0096
A100	B2/FP32	2	`danmp_partial`	0.4895	0.0107	2.19%†	0.9778
A100	B2/FP16	2	`linear`	0.5218	0.0080	1.53%†	1.0000
A100	B2/FP16	2	`morton`	0.5199	0.0061	1.17%†	0.9964
A100	B2/FP16	2	`random`	0.5194	0.0071	1.37%†	0.9954
A100	B2/FP16	2	`scanline`	0.5187	0.0056	1.08%†	0.9941
A100	B2/FP16	2	`centroid`	0.5196	0.0057	1.10%†	0.9958
A100	B2/FP16	2	`hilbert`	0.5188	0.0053	1.02%†	0.9943
A100	B2/FP16	2	`kmeans_cap`	0.5219	0.3384	64.84%†	1.0002
A100	B2/FP16	2	`danmp_partial`	0.5089	0.3239	63.65%†	0.9753
A100	B2/BF16	2	`linear`	0.4988	0.0080	1.60%†	1.0000
A100	B2/BF16	2	`morton`	0.4987	0.0072	1.44%†	0.9998
A100	B2/BF16	2	`random`	0.4981	0.0073	1.47%†	0.9986
A100	B2/BF16	2	`scanline`	0.4992	0.0079	1.58%†	1.0008
A100	B2/BF16	2	`centroid`	0.4991	0.0088	1.76%†	1.0006
A100	B2/BF16	2	`hilbert`	0.4968	0.0056	1.13%†	0.9960
A100	B2/BF16	2	`kmeans_cap`	0.5051	0.0062	1.23%†	1.0126
A100	B2/BF16	2	`danmp_partial`	0.4884	0.0073	1.49%†	0.9791
A100	B4/FP32	4	`linear`	0.8051	0.0057	0.71%†	1.0000
A100	B4/FP32	4	`morton`	0.8048	0.0054	0.67%†	0.9996
A100	B4/FP32	4	`random`	0.8059	0.0066	0.82%†	1.0010
A100	B4/FP32	4	`scanline`	0.8061	0.0053	0.66%†	1.0012
A100	B4/FP32	4	`centroid`	0.8055	0.0056	0.70%†	1.0005
A100	B4/FP32	4	`hilbert`	0.8059	0.0063	0.78%†	1.0010
A100	B4/FP32	4	`kmeans_cap`	0.9115	0.0110	1.21%†	1.1322
A100	B4/FP32	4	`danmp_partial`	0.9708	0.0127	1.31%†	1.2058
A100	B4/FP16	4	`linear`	0.4982	0.0085	1.71%†	1.0000
A100	B4/FP16	4	`morton`	0.4970	0.0061	1.23%†	0.9976

(continued on next page)

(Table 6 continued from previous page.)

GPU	Config	B	Order	p_{50} (ms)	σ (ms)	σ/μ	r
A100	B4/FP16	4	`random`	0.4943	0.0057	1.15%†	0.9922
A100	B4/FP16	4	`scanline`	0.4964	0.0057	1.15%†	0.9964
A100	B4/FP16	4	`centroid`	0.4964	0.0058	1.17%†	0.9964
A100	B4/FP16	4	`hilbert`	0.4944	0.0056	1.13%†	0.9924
A100	B4/FP16	4	`kmeans_cap`	0.4987	0.0062	1.24%†	1.0010
A100	B4/FP16	4	`danmp_partial`	0.4854	0.0050	1.03%†	0.9743
A100	B4/BF16	4	`linear`	0.4933	0.0301	6.10%†	1.0000
A100	B4/BF16	4	`morton`	0.4907	0.0057	1.16%†	0.9947
A100	B4/BF16	4	`random`	0.4921	0.0075	1.52%†	0.9976
A100	B4/BF16	4	`scanline`	0.4906	0.0057	1.16%†	0.9945
A100	B4/BF16	4	`centroid`	0.4910	0.0074	1.51%†	0.9953
A100	B4/BF16	4	`hilbert`	0.4913	0.0057	1.16%†	0.9959
A100	B4/BF16	4	`kmeans_cap`	0.4952	0.0060	1.21%†	1.0039
A100	B4/BF16	4	`danmp_partial`	0.4838	0.0065	1.34%†	0.9807
A100	B8/FP32	8	`linear`	1.6387	0.0108	0.66%†	1.0000
A100	B8/FP32	8	`morton`	1.6372	0.0117	0.71%†	0.9991
A100	B8/FP32	8	`random`	1.6385	0.0112	0.68%†	0.9999
A100	B8/FP32	8	`scanline`	1.6374	0.0123	0.75%†	0.9992
A100	B8/FP32	8	`centroid`	1.6386	0.0107	0.65%†	0.9999
A100	B8/FP32	8	`hilbert`	1.6379	0.0113	0.69%†	0.9995
A100	B8/FP32	8	`kmeans_cap`	1.6330	0.0115	0.70%†	0.9965
A100	B8/FP32	8	`danmp_partial`	1.7546	0.0124	0.71%†	1.0707
A100	B8/FP16	8	`linear`	0.5268	0.0071	1.35%†	1.0000
A100	B8/FP16	8	`morton`	0.5272	0.0056	1.06%†	1.0008
A100	B8/FP16	8	`random`	0.5257	0.0054	1.03%†	0.9979
A100	B8/FP16	8	`scanline`	0.5263	0.0106	2.01%†	0.9991
A100	B8/FP16	8	`centroid`	0.5256	0.0055	1.05%†	0.9977
A100	B8/FP16	8	`hilbert`	0.5242	0.0058	1.11%†	0.9951
A100	B8/FP16	8	`kmeans_cap`	0.5307	0.0062	1.17%†	1.0074
A100	B8/FP16	8	`danmp_partial`	0.5146	0.0062	1.20%†	0.9768
A100	B8/BF16	8	`linear`	0.4935	0.0074	1.50%†	1.0000
A100	B8/BF16	8	`morton`	0.4934	0.0059	1.20%†	0.9998
A100	B8/BF16	8	`random`	0.4931	0.0081	1.64%†	0.9992
A100	B8/BF16	8	`scanline`	0.4919	0.0070	1.42%†	0.9968
A100	B8/BF16	8	`centroid`	0.4916	0.0057	1.16%†	0.9961
A100	B8/BF16	8	`hilbert`	0.4918	0.0067	1.36%†	0.9966
A100	B8/BF16	8	`kmeans_cap`	0.4957	0.0060	1.21%†	1.0045
A100	B8/BF16	8	`danmp_partial`	0.4833	0.0067	1.39%†	0.9793
H100	B1/FP32	1	`linear`	0.4155	0.0100	2.41%†	1.0000
H100	B1/FP32	1	`morton`	0.4184	0.0081	1.94%†	1.0070
H100	B1/FP32	1	`random`	0.4180	0.0091	2.18%†	1.0060
H100	B1/FP32	1	`scanline`	0.4151	0.0068	1.64%†	0.9990
H100	B1/FP32	1	`centroid`	0.4198	0.0086	2.05%†	1.0103
H100	B1/FP32	1	`hilbert`	0.4224	0.0071	1.68%†	1.0166
H100	B1/FP32	1	`kmeans_cap`	0.4141	0.0103	2.49%†	0.9966
H100	B1/FP32	1	`danmp_partial`	0.4145	0.0107	2.58%†	0.9976
H100	B1/FP16	1	`linear`	0.4022	0.0075	1.86%†	1.0000
H100	B1/FP16	1	`morton`	0.4034	0.0067	1.66%†	1.0030
H100	B1/FP16	1	`random`	0.4026	0.0067	1.66%†	1.0010
H100	B1/FP16	1	`scanline`	0.4025	0.0062	1.54%†	1.0007
H100	B1/FP16	1	`centroid`	0.4023	0.0089	2.21%†	1.0002
H100	B1/FP16	1	`hilbert`	0.4015	0.0070	1.74%†	0.9983
H100	B1/FP16	1	`kmeans_cap`	0.4035	0.0076	1.88%†	1.0032
H100	B1/FP16	1	`danmp_partial`	0.4006	0.0075	1.87%†	0.9960
H100	B1/BF16	1	`linear`	0.4059	0.0245	6.04%†	1.0000

(continued on next page)

(Table 6 continued from previous page.)

GPU	Config	B	Order	p_{50} (ms)	σ (ms)	σ/μ	r
H100	B1/BF16	1	`morton`	0.4044	0.0072	1.78%†	0.9963
H100	B1/BF16	1	`random`	0.4032	0.0070	1.74%†	0.9933
H100	B1/BF16	1	`scanline`	0.4022	0.0066	1.64%†	0.9909
H100	B1/BF16	1	`centroid`	0.4018	0.0066	1.64%†	0.9899
H100	B1/BF16	1	`hilbert`	0.4022	0.0065	1.62%†	0.9909
H100	B1/BF16	1	`kmeans_cap`	0.4007	0.0069	1.72%†	0.9872
H100	B1/BF16	1	`danmp_partial`	0.4005	0.0072	1.80%†	0.9867
H100	B2/FP32	2	`linear`	0.4075	0.0067	1.64%†	1.0000
H100	B2/FP32	2	`morton`	0.4142	0.0068	1.64%†	1.0164
H100	B2/FP32	2	`random`	0.4147	0.0075	1.81%†	1.0177
H100	B2/FP32	2	`scanline`	0.4093	0.0060	1.47%†	1.0044
H100	B2/FP32	2	`centroid`	0.4090	0.0070	1.71%†	1.0037
H100	B2/FP32	2	`hilbert`	0.4092	0.0068	1.66%†	1.0042
H100	B2/FP32	2	`kmeans_cap`	0.4092	0.2786	68.08%†	1.0042
H100	B2/FP32	2	`danmp_partial`	0.4097	0.0069	1.68%†	1.0054
H100	B2/FP16	2	`linear`	0.4021	0.0073	1.82%†	1.0000
H100	B2/FP16	2	`morton`	0.4045	0.0061	1.51%†	1.0060
H100	B2/FP16	2	`random`	0.4037	0.0069	1.71%†	1.0040
H100	B2/FP16	2	`scanline`	0.4018	0.0066	1.64%†	0.9993
H100	B2/FP16	2	`centroid`	0.4025	0.0068	1.69%†	1.0010
H100	B2/FP16	2	`hilbert`	0.4032	0.0068	1.69%†	1.0027
H100	B2/FP16	2	`kmeans_cap`	0.3994	0.3012	75.41%†	0.9933
H100	B2/FP16	2	`danmp_partial`	0.4042	0.2651	65.59%†	1.0052
H100	B2/BF16	2	`linear`	0.4022	0.0075	1.86%†	1.0000
H100	B2/BF16	2	`morton`	0.4025	0.0069	1.71%†	1.0007
H100	B2/BF16	2	`random`	0.4032	0.0063	1.56%†	1.0025
H100	B2/BF16	2	`scanline`	0.4031	0.0063	1.56%†	1.0022
H100	B2/BF16	2	`centroid`	0.4025	0.0064	1.59%†	1.0007
H100	B2/BF16	2	`hilbert`	0.4035	0.0068	1.69%†	1.0032
H100	B2/BF16	2	`kmeans_cap`	0.3989	0.0066	1.65%†	0.9918
H100	B2/BF16	2	`danmp_partial`	0.4044	0.2253	55.71%†	1.0055
H100	B4/FP32	4	`linear`	0.4138	0.0085	2.05%†	1.0000
H100	B4/FP32	4	`morton`	0.4160	0.0087	2.09%†	1.0053
H100	B4/FP32	4	`random`	0.4144	0.0080	1.93%†	1.0014
H100	B4/FP32	4	`scanline`	0.4139	0.0082	1.98%†	1.0002
H100	B4/FP32	4	`centroid`	0.4143	0.0083	2.00%†	1.0012
H100	B4/FP32	4	`hilbert`	0.4153	0.0078	1.88%†	1.0036
H100	B4/FP32	4	`kmeans_cap`	0.4108	0.0082	2.00%†	0.9928
H100	B4/FP32	4	`danmp_partial`	0.4149	0.0084	2.02%†	1.0027
H100	B4/FP16	4	`linear`	0.4050	0.0069	1.70%†	1.0000
H100	B4/FP16	4	`morton`	0.4044	0.0069	1.71%†	0.9985
H100	B4/FP16	4	`random`	0.4043	0.0066	1.63%†	0.9983
H100	B4/FP16	4	`scanline`	0.4021	0.0067	1.67%†	0.9928
H100	B4/FP16	4	`centroid`	0.4033	0.0072	1.79%†	0.9958
H100	B4/FP16	4	`hilbert`	0.4044	0.0062	1.53%†	0.9985
H100	B4/FP16	4	`kmeans_cap`	0.4015	0.0060	1.49%†	0.9914
H100	B4/FP16	4	`danmp_partial`	0.4038	0.0066	1.63%†	0.9970
H100	B4/BF16	4	`linear`	0.4087	0.0371	9.08%†	1.0000
H100	B4/BF16	4	`morton`	0.4040	0.0066	1.63%†	0.9885
H100	B4/BF16	4	`random`	0.4020	0.0068	1.69%†	0.9836
H100	B4/BF16	4	`scanline`	0.4035	0.0058	1.44%†	0.9873
H100	B4/BF16	4	`centroid`	0.4046	0.0061	1.51%†	0.9900
H100	B4/BF16	4	`hilbert`	0.4055	0.0062	1.53%†	0.9922
H100	B4/BF16	4	`kmeans_cap`	0.3991	0.0065	1.63%†	0.9765
H100	B4/BF16	4	`danmp_partial`	0.4046	0.0070	1.73%†	0.9900

(continued on next page)

Table 7: Backward and forward latency dispersion: median (p_{50}), tail (p_{99}), and standard deviation (σ) per backend on A100 and H100. Config: B4 800×1333, BF16, eager, external softmax; N=100 samples.

		A100			H100		
Backend	Phase	p50 (ms)	p99 (ms)	σ (ms)	p50 (ms)	p99 (ms)	σ (ms)
Reference	Fwd	1.416	2.681	0.127	1.053	2.193	0.114
	Bwd	5.440	6.656	0.163	2.142	3.274	0.143
CUDA, reference	Fwd	0.544	0.567	0.005	0.446	0.474	0.007
	Bwd	1.495	2.793	0.130	1.170	2.382	0.121
CUDA	Fwd	0.564	0.607	0.008	0.472	0.504	0.007
	Bwd	1.522	2.788	0.127	1.206	2.612	0.141
Triton, point-parallel	Fwd	0.561	0.632	0.010	0.441	0.473	0.006
	Bwd	1.777	2.351	0.058	1.160	2.353	0.124
Triton, query-block	Fwd	0.534	0.587	0.008	0.445	0.481	0.007
	Bwd	1.572	2.297	0.074	1.187	2.414	0.124

(Table 6 continued from previous page.)

GPU	Config	B	Order	p_{50} (ms)	σ (ms)	σ/μ	r
H100	B8/FP32	8	`linear`	0.6732	0.0056	0.83%†	1.0000
H100	B8/FP32	8	`morton`	0.6743	0.0057	0.85%†	1.0016
H100	B8/FP32	8	`random`	0.6741	0.0061	0.90%†	1.0013
H100	B8/FP32	8	`scanline`	0.6754	0.0058	0.86%†	1.0033
H100	B8/FP32	8	`centroid`	0.6758	0.0064	0.95%†	1.0039
H100	B8/FP32	8	`hilbert`	0.6745	0.0065	0.96%†	1.0019
H100	B8/FP32	8	`kmeans_cap`	0.6684	0.0062	0.93%†	0.9929
H100	B8/FP32	8	`danmp_partial`	0.6706	0.0062	0.92%†	0.9961
H100	B8/FP16	8	`linear`	0.4024	0.0061	1.52%†	1.0000
H100	B8/FP16	8	`morton`	0.4020	0.0063	1.57%†	0.9990
H100	B8/FP16	8	`random`	0.4017	0.2002	49.84%†	0.9983
H100	B8/FP16	8	`scanline`	0.4025	0.0116	2.88%†	1.0002
H100	B8/FP16	8	`centroid`	0.4004	0.0067	1.67%†	0.9950
H100	B8/FP16	8	`hilbert`	0.4018	0.0067	1.67%†	0.9985
H100	B8/FP16	8	`kmeans_cap`	0.4008	0.0065	1.62%†	0.9960
H100	B8/FP16	8	`danmp_partial`	0.4030	0.0072	1.79%†	1.0015
H100	B8/BF16	8	`linear`	0.4040	0.0061	1.51%†	1.0000
H100	B8/BF16	8	`morton`	0.4025	0.0059	1.47%†	0.9963
H100	B8/BF16	8	`random`	0.4028	0.0062	1.54%†	0.9970
H100	B8/BF16	8	`scanline`	0.4020	0.0061	1.52%†	0.9950
H100	B8/BF16	8	`centroid`	0.4036	0.0059	1.46%†	0.9990
H100	B8/BF16	8	`hilbert`	0.4023	0.0057	1.42%†	0.9958
H100	B8/BF16	8	`kmeans_cap`	0.3996	0.0066	1.65%†	0.9891
H100	B8/BF16	8	`danmp_partial`	0.4017	0.0073	1.82%†	0.9943

Figure 2 shows the block-size histogram of the per-allocation memory snapshot at the encoder cell, comparing the CUDA backend against our query-block Triton backend during the backward pass. The asymmetry is dominated by three FP32 scratch buffers in the CUDA path (the topmost 127.50/63.75/31.88 MB blocks) that are absent from the Triton path; this is the FP32-accumulator cost the design choice removes (Section 3.3).

Table 7 reports median (p_{50}), tail (p_{99}), and standard deviation (σ) for the forward and backward passes of every backend at the decoder configuration on both GPUs. All five backends exhibit sub-percent relative standard deviations, confirming that the p_{50} values in Table 4 are stable.

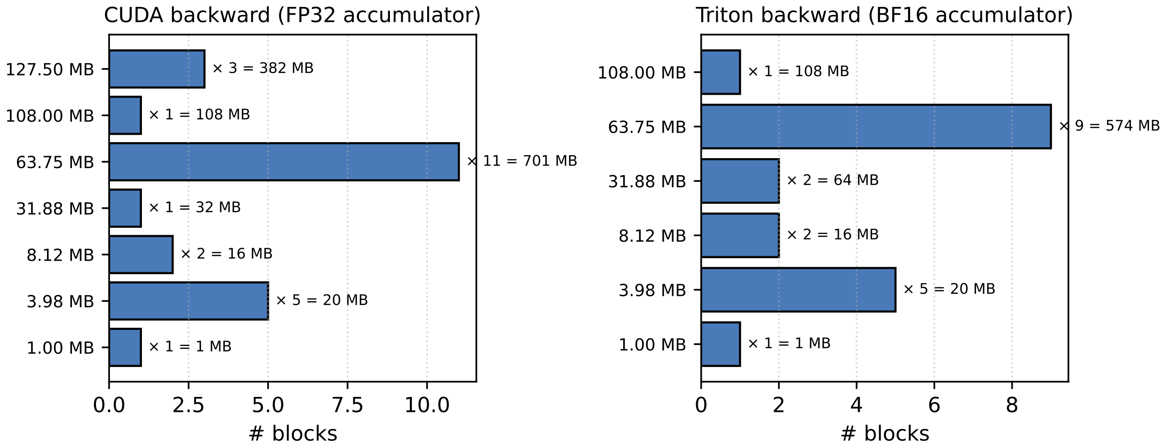

Figure 2: Per-allocation block-size histogram during the encoder-scale backward pass at BF16, captured on the A100 partition. Blocks $\geq$ 1 MB are shown; sub-MB scaffolding is elided. The CUDA path's top three 127.50/63.75/31.88 MB FP32 scratch buffers (the gradient-of-value, gradient-of-sampling-location, and gradient-of-attention accumulators, allocated in single precision regardless of the input dtype) are absent from the Triton path. Sizes are reported with the binary mebibyte convention (1 MB = 2^{20} B) for consistency with PyTorch's `max_memory_allocated` reporting.

Table 8: Scatter-add p_{50} latency at six representative cells (M=32,768, D=256, $N \in \{16k, 64k, 256k\}$, uniform or Zipf-s=1.5 index distribution) on A100 and H100. FP32 stays fast across all collision rates; BF16 collapses two to three orders of magnitude under Zipf contention on both architectures. The full 72-cell sweep accompanies the source release upon acceptance.

N	dist.	worst-row coll.	A100 bf16 (ms)	A100 fp32 (ms)	H100 bf16 (ms)	H100 fp32 (ms)
16k	uniform	5	0.109	0.0799	0.0395	0.0477
16k	zipf	6,251	130.6	0.114	72.3	0.0824
64k	uniform	10	0.366	0.254	0.125	0.151
64k	zipf	25,174	249.4	0.386	369.3	0.285
256k	uniform	23	1.42	0.954	0.464	0.561
256k	zipf	100,797	718.7	1.48	1074.5	1.09

Tables 8 and 9 support the generalization analysis of Section 4.2.

Tables 10 and 11 list the complete operator-level sweep across all twelve configurations, three precisions, and five backends, one table per GPU. One anomalous cell: the Triton point-parallel backward on H100 at B4_800×1333_enc FP16 (126.6 ms) is 2.5× slower than the same kernel at BF16 (51.0 ms) and 5.5× slower than FP32 (23.0 ms). This inversion affects only the point-parallel tiling, which is already identified as a throughput failure in Section 3.2; the recommended query-block tiling shows the expected precision ordering at the same configuration (3.1 ms FP16 vs. 3.1 ms BF16). The anomaly is specific to the point-parallel tiling at FP16 on the Triton version shipped with PyTorch 2.9.0 (pinned in Section 5); the recommended query-block tiling shows the expected precision ordering at the same configuration and is unaffected. Decoder configurations correspond to the standard Deformable DETR decoder stage; encoder configurations (suffixed "enc") correspond to encoder-stage feature maps of roughly the resolutions used by Deformable DETR, DINO, and Mask2Former. All latencies are p_{50} in milliseconds over 100 measurement iterations after 10 warmup iterations.

Table 9: Kernel-level counter fingerprint for the nvcc-compiled scatter-add operator and our Triton-compiled MSDA backward at BF16, on A100 (SM 8.0) and H100 (SM 9.0). The `atom/red` column inversion across the SM 8.0 / SM 9.0 boundary for MSDA is absent for scatter-add: the two operators take different compiler paths on the same hardware. *Kernel-replay profiling underreports wall time by ∼1.8–2.2×; benchmark p_{50} values are used for absolute latencies.

Counter	scatter-add, A100	scatter-add, H100	MSDA bwd, A100	MSDA bwd, H100
Kernel time* (ms)	603.90	426.98	109.70	0.74
L1 `atom` sectors ($\times 10^6$)	525	492	447	0
L1 `red` sectors ($\times 10^6$)	0	0	0	35
L2 `atom` sectors ($\times 10^6$)	633	739	672	0
L2 `red` sectors ($\times 10^6$)	0	0	0	52
DRAM traffic (MB)	148	155	170	162

Table 10: Full operator-level latency sweep on A100. p_{50} in milliseconds, eager mode, external softmax.

Config	Precision	Reference Fwd	Reference Bwd	CUDA, reference Fwd	CUDA, reference Bwd	CUDA Fwd	CUDA Bwd	Triton, point-parallel Fwd	Triton, point-parallel Bwd	Triton, query-block Fwd	Triton, query-block Bwd
B1_640_d256_L4	FP32	1.127	2.530	0.504	1.348	0.598	1.454	0.554	1.484	0.564	1.519
	FP16	1.137	2.549	0.573	1.492	0.603	1.544	0.563	1.507	0.572	1.536
	BF16	1.102	4.204	0.546	1.445	0.576	1.494	0.537	1.469	0.547	1.498
B2_1280_d256_L4	FP32	1.465	3.589	0.502	1.384	0.579	1.458	0.570	1.481	0.546	1.512
	FP16	1.417	3.272	0.551	1.515	0.563	1.519	0.558	1.564	0.532	1.500
	BF16	1.417	6.617	0.551	1.516	0.565	1.526	0.560	2.420	0.536	1.776
B2_1536x2048_d256_L4	FP32	1.739	5.236	0.501	1.398	0.580	1.459	0.580	1.722	0.550	1.518
	FP16	1.597	4.198	0.588	1.505	0.571	1.530	0.564	1.746	0.541	1.505
	BF16	1.601	6.796	0.585	1.514	0.570	1.530	0.568	2.160	0.539	1.741
B2_1536x2048_d256_L4_enc	FP32	30.956	80.523	6.720	24.938	5.292	19.705	15.138	51.214	3.430	20.756
	FP16	25.606	85.265	6.687	24.320	2.989	17.274	16.301	70.987	2.135	13.103
	BF16	25.569	692.335	6.649	24.280	2.920	17.212	16.447	94.253	1.997	123.404
B2_800x1333_d256_L2	FP32	1.026	2.222	0.498	1.369	0.582	1.454	0.541	1.487	0.563	1.537
	FP16	1.002	2.209	0.570	1.501	0.591	1.525	0.551	1.483	0.575	1.537
	BF16	0.983	2.365	0.551	1.457	0.573	1.477	0.545	1.453	0.553	1.503
B2_800x1333_d256_L4	FP32	1.323	2.841	0.501	1.371	0.583	1.458	0.542	1.482	0.550	1.510
	FP16	1.310	2.838	0.575	1.510	0.597	1.535	0.556	1.503	0.564	1.515
	BF16	1.293	4.368	0.560	1.468	0.578	1.496	0.537	1.458	0.549	1.486
B2_800x1333_d256_L4_enc	FP32	10.264	30.562	2.193	7.898	1.396	6.330	5.156	18.343	1.173	7.402
	FP16	8.593	32.406	2.167	7.574	0.862	5.649	5.513	26.963	0.728	4.671
	BF16	8.593	350.315	2.168	7.553	0.863	5.636	5.546	50.364	0.710	86.413
B2_800x1333_d256_L5	FP32	1.951	6.730	0.490	1.359	0.575	1.452	0.536	1.544	0.544	1.499
	FP16	1.801	5.366	0.598	1.496	0.589	1.527	0.550	1.479	0.558	1.506
	BF16	1.783	7.389	0.583	1.456	0.571	1.486	0.532	1.522	0.542	1.853
B2_800x1333_d512_L4	FP32	1.533	3.949	0.503	1.366	0.584	1.447	0.544	1.481	0.554	1.512
	FP16	1.458	4.047	0.570	1.504	0.591	1.530	0.551	1.483	0.560	1.519
	BF16	1.445	6.789	0.545	1.464	0.565	1.486	0.528	1.816	0.537	2.255
B4_800x1333_d256_L4	FP32	1.497	3.775	0.492	1.384	0.576	1.465	0.563	1.491	0.543	1.526
	FP16	1.410	3.396	0.546	1.495	0.567	1.522	0.563	1.475	0.534	1.505
	BF16	1.416	5.440	0.544	1.495	0.564	1.522	0.561	1.777	0.534	1.572
B4_800x1333_d256_L4_enc	FP32	19.982	59.279	4.312	15.288	2.724	12.164	10.244	34.546	2.277	14.294
	FP16	16.674	64.131	4.284	14.881	1.701	11.054	11.009	53.709	1.436	9.140
	BF16	16.616	686.856	4.263	14.819	1.673	11.001	11.195	100.071	1.367	167.570
B8_800x1333_d256_L4	FP32	1.947	6.258	0.520	1.639	0.575	1.560	0.629	2.253	0.543	1.674
	FP16	1.755	4.882	0.618	1.719	0.593	1.599	0.589	2.311	0.565	1.539
	BF16	1.721	7.468	0.602	1.701	0.564	1.581	0.560	3.175	0.544	2.043

B Numerical Parity

Table 12 reports maximum absolute error and mean relative error of each backend against the FP32 reference implementation (cast to float64 before comparison). At FP32 the custom backends match the reference to $1.6{\cdot}10^{-6}$, the magnitude expected from reordered parallel summation of single-precision floats. At FP16 and BF16 the custom backends track the reference within 0.2 % of its own precision-bounded error; the mean

Table 11: Full operator-level latency sweep on H100. p_{50} in milliseconds, eager mode, external softmax.

Config	Precision	Reference Fwd	Reference Bwd	CUDA, reference Fwd	CUDA, reference Bwd	CUDA Fwd	CUDA Bwd	Triton, point-parallel Fwd	Triton, point-parallel Bwd	Triton, query-block Fwd	Triton, query-block Bwd
B1_640_d256_L4	FP32	0.860	1.977	0.415	1.128	0.484	1.095	0.445	1.113	0.452	1.264
	FP16	0.855	1.869	0.435	1.231	0.457	1.267	0.426	1.126	0.437	1.149
	BF16	0.850	1.858	0.434	1.228	0.458	1.160	0.428	1.126	0.434	1.145
B2_1280_d256_L4	FP32	1.086	2.383	0.408	1.070	0.476	1.139	0.472	1.151	0.451	1.173
	FP16	1.021	2.112	0.444	1.176	0.468	1.206	0.463	1.243	0.442	1.179
	BF16	1.018	2.099	0.443	1.170	0.467	1.198	0.465	1.249	0.441	1.180
B2_1536x2048_d256_L4	FP32	1.245	3.289	0.414	1.073	0.479	1.238	0.475	1.154	0.454	1.173
	FP16	1.145	2.584	0.445	1.175	0.473	1.204	0.470	1.242	0.446	1.183
	BF16	1.146	2.591	0.449	1.167	0.474	1.204	0.467	1.227	0.445	1.182
B2_1536x2048_d256_L4_enc	FP32	21.179	51.437	4.015	13.559	2.867	11.639	9.329	34.150	1.843	6.677
	FP16	16.484	49.477	3.966	12.983	1.653	10.169	10.045	56.283	1.143	4.309
	BF16	16.488	48.983	3.968	12.986	1.657	10.188	9.982	50.662	1.151	4.334
B2_800x1333_d256_L2	FP32	0.780	1.626	0.411	1.056	0.481	1.126	0.448	1.254	0.456	1.285
	FP16	0.761	1.623	0.444	1.145	0.468	1.179	0.435	1.261	0.444	1.290
	BF16	0.760	1.618	0.448	1.148	0.471	1.177	0.434	1.260	0.441	1.286
B2_800x1333_d256_L4	FP32	0.988	2.199	0.412	1.143	0.476	1.118	0.447	1.130	0.451	1.156
	FP16	0.974	2.207	0.446	1.149	0.461	1.173	0.435	1.140	0.440	1.157
	BF16	0.971	2.207	0.445	1.144	0.467	1.172	0.434	1.139	0.442	1.154
B2_800x1333_d256_L4_enc	FP32	7.202	18.344	1.325	4.308	0.731	3.637	3.176	11.588	0.647	2.387
	FP16	5.638	18.427	1.276	4.074	0.502	3.283	3.362	63.864	0.477	1.571
	BF16	5.644	18.360	1.275	4.075	0.503	3.303	3.341	25.184	0.477	1.570
B2_800x1333_d256_L5	FP32	1.397	4.114	0.411	1.060	0.482	1.120	0.449	1.140	0.456	1.164
	FP16	1.333	3.187	0.445	1.262	0.469	1.188	0.435	1.149	0.444	1.175
	BF16	1.326	3.199	0.446	1.159	0.469	1.187	0.434	1.148	0.445	1.176
B2_800x1333_d512_L4	FP32	1.147	2.632	0.439	1.113	0.508	1.179	0.476	1.196	0.482	1.218
	FP16	1.087	2.385	0.444	1.158	0.468	1.191	0.436	1.147	0.443	1.176
	BF16	1.087	2.372	0.445	1.153	0.470	1.188	0.438	1.147	0.446	1.178
B4_800x1333_d256_L4	FP32	1.091	2.537	0.409	1.070	0.478	1.236	0.448	1.144	0.454	1.170
	FP16	1.055	2.150	0.447	1.170	0.465	1.306	0.440	1.163	0.445	1.185
	BF16	1.053	2.142	0.446	1.170	0.472	1.206	0.441	1.160	0.445	1.187
B4_800x1333_d256_L4_enc	FP32	14.021	35.468	2.607	8.466	1.411	7.078	6.308	22.952	1.231	4.600
	FP16	10.974	35.854	2.552	8.044	0.943	6.460	6.737	126.566	0.782	3.057
	BF16	10.986	35.886	2.553	8.045	0.947	6.468	6.702	50.998	0.786	3.062
B8_800x1333_d256_L4	FP32	1.315	4.107	0.413	1.163	0.476	1.129	0.475	1.450	0.454	1.167
	FP16	1.272	3.100	0.470	1.272	0.470	1.198	0.460	1.735	0.441	1.183
	BF16	1.274	3.101	0.470	1.269	0.473	1.195	0.462	1.730	0.442	1.177

relative error exceeding 1 at BF16 reflects BF16's 7-bit mantissa blowup on small-magnitude values under uniform random inputs rather than any kernel defect.

Table 12: Numerical parity versus the FP32 reference (cast to float64). All backends pass a 10× threshold test.

	FP32		FP16		BF16	
Backend	max abs	mean rel	max abs	mean rel	max abs	mean rel
Reference	0	0	0.367	0.225	0.573	1.49
CUDA path	$1.6{\cdot}10^{-6}$	$5.9{\cdot}10^{-6}$	0.366	0.221	0.561	1.46
Triton path	$1.6{\cdot}10^{-6}$	$5.8{\cdot}10^{-6}$	0.367	0.264	0.561	1.47

C Full Scatter-Add Sweep

We sweep PyTorch's standard scatter-add operator across $(M, D, N) \in \{32{,}768, 131{,}072\} \times \{64, 256, 1024\} \times \{16{,}384, 65{,}536, 262{,}144\}$, over uniform and Zipf-$s$=1.5 index distributions, at BF16, FP16, and FP32 precision, on A100 and H100. The Zipf distribution produces single-address cross-warp contention of up to $\sim 10^5$ writes to the worst row. Table 8 reports six representative cells at D=256; the full 72-cell matrix will accompany the source release upon acceptance. At low contention FP16 is 20 %–30 % faster than BF16 on A100, consistent with the native-vs-CAS expectation, but at the worst Zipf cell FP16 runs 1.5–2.1× slower than BF16 on both

GPUs: the native FP16 atomic hardware scales worse than the software CAS loop under extreme contention. Between-run spread reaches ∼55 % on A100 BF16 at the worst cell, intrinsic to CAS-retry contention rather than thermal drift (FP32 at the same cell is stable to 1 %). Table 9 reports the GPU hardware counters at the worst-contention cell for both operators; the atomic-lowering *flips* between `atom.*` and `red.*` across SM 8.0/9.0 for MSDA's Triton-compiled backward but not for the nvcc-compiled scatter-add operator.

D Technique Ablation

Table 13 isolates the contribution of each kernel-design technique at the encoder-scale configuration (B=2, 1536×2048, L=4, BF16), where the differences are largest; at decoder scale all custom backends converge within 10 % of each other. The CUDA and Triton paths are independent tracks; the Triton rows are not cumulative with the CUDA rows.

Table 13: Cumulative technique ablation at encoder scale (B=2, 1536×2048, L=4, BF16).

		A100		H100	
Backend	Cumulative techniques	Fwd (ms)	Bwd (ms)	Fwd (ms)	Bwd (ms)
Reference impl.	(none)	25.57	692.33	16.49	48.98
CUDA baseline	rebuild only	6.65	24.28	3.97	12.99
CUDA + read-side	+ vectorized loads, warp broadcast	3.12	16.86	1.67	9.99
CUDA + read + write	+ relaxed atomics	2.92	17.21	1.66	10.19
Triton point-parallel	occupancy-maximizing tiling	16.45	94.25	9.98	50.66
Triton query-block	query-block tiling + relaxed ordering	2.00	123.40	1.15	4.33

Read-side optimizations account for the majority of the CUDA forward improvement on A100 (6.65 → 3.12 ms, 2.1×); write-side relaxed atomics add only 7 % more (3.12 → 2.92 ms), confirming that the forward bottleneck is read bandwidth, not atomic contention. The tiling strategy is the single largest Triton lever (8.2× on the A100 forward when switching from point-parallel to query-block). The backward pass does not benefit from kernel-design techniques on A100 at BF16: the CUDA backward is unchanged between the read-side and write-side rows (16.86 vs 17.21 ms) and the Triton query-block backward (123.4 ms) is *slower* than the point-parallel backward (94.3 ms) because the former emits one relaxed BF16 atomic per corner where the latter amortizes atomics across a larger tile. Both observations reflect the hardware-gated CAS loop of Section 3.3. On H100, where hardware BF16 atomics resolve the contention, the Triton query-block backward drops to 4.3 ms, confirming the residual is architectural rather than algorithmic.

E Kernel Pseudocode

This appendix gives pseudocode for the two kernel patterns that are load-bearing for the speedups reported in the body: warp-level sharing of sampling coordinates in the CUDA forward path, and relaxed-ordering scattered atomic accumulation in the Triton backward path. Both listings elide error checks, boundary masking, and template parameters for readability; the full sources will accompany the release upon acceptance.

F Partial DANMP Port: Sampling-Point Clustering Ablation

The CAP analogue cited in Section 3.1 deviates from DANMP's full recipe on four axes: (i) we cluster the mean cross-level reference points (deterministic per query, nearly uniform in image space), whereas DANMP clusters the sampling points (reference + learned offset, where the hotspot signal lives); (ii) we use fixed-k K-means with $k \approx 2\sqrt{Q}$, whereas DANMP uses a 9×9 pixel-radius DBSCAN-style variable-k hotspot detector; (iii) we cluster all queries, whereas DANMP samples 20% and maps the rest to nearest centroids; (iv) our downstream primitive is argsort-dispatch of the stock kernel, whereas DANMP's is packing (a gather-plus-batch fusion of bilinears that share four-corner sub-targets). All four deviations make our test weaker at detecting the locality signal DANMP targets.

```
// Forward: each warp processes one (batch, query, head) tuple.
// Sampling coordinates are loaded once per warp by lane 0 and
// broadcast to the other 31 lanes via __shfl_sync.
__global__ void forward_kernel(
    const scalar_t* __restrict__ p_value,  // feature maps
    const scalar_t* __restrict__ p_loc,    // sampling coords (disjoint)
    const scalar_t* __restrict__ p_attn,   // attention weights (disjoint)
    const int64_t*  data_spatial_shapes,
    const int64_t*  data_level_start_index,
    scalar_t*       p_output) {

  // Cache attention weights for this (query, head) in shared memory
  // once; every lane reads its LK values locally thereafter.
  __shared__ scalar_t p_mask_shm[L * K];
  if (threadIdx.x < L * K)
    p_mask_shm[threadIdx.x] = p_attn_ptr[threadIdx.x];
  __syncwarp();

  const uint32_t lane_mask = 0xFFFFFFFF;
  for (int li = 0; li < L; ++li) {
    const int spatial_h = data_spatial_shapes[li * 2];
    const int spatial_w = data_spatial_shapes[li * 2 + 1];
    for (int ki = 0; ki < K; ++ki) {

      // Lane 0 issues the single global load for this sample point;
      // all 32 lanes of the warp then read the same (loc_w, loc_h)
      // via a register-to-register shuffle.
      opmath_t loc_w, loc_h;
      if (threadIdx.x == 0) {
        loc_w = (opmath_t)p_loc_ptr[li * K * 2 + ki * 2    ];
        loc_h = (opmath_t)p_loc_ptr[li * K * 2 + ki * 2 + 1];
      }
      loc_w = __shfl_sync(lane_mask, loc_w, 0);
      loc_h = __shfl_sync(lane_mask, loc_h, 0);

      const opmath_t attn = p_mask_shm[li * K + ki];
      const opmath_t w_im = loc_w * spatial_w - 0.5;
      const opmath_t h_im = loc_h * spatial_h - 0.5;

      // Each lane is responsible for a disjoint channel slice.
      // Wide (ulonglong4) loads on the feature row below.
      accumulate_bilinear_sample(p_value, spatial_h, spatial_w,
                                 w_im, h_im, attn, p_output);
    }
  }
}
```

Figure 3: Warp-level sharing of sampling coordinates in the CUDA forward kernel. Lane 0 issues a single 16 byte load for each (`loc_w`, `loc_h`) pair per sample point and broadcasts the value to the other 31 lanes of the warp via `__shfl_sync`, amortizing one coordinate fetch across the whole warp. Disjoint `p_loc` and `p_attn` pointers eliminate the concatenated sampling-parameter buffer of the reference implementation (Section 2.3).

A partial port closing three of the four deviations (sampling-point synthesis from reference + uniform ΔP, DBSCAN-style 9-pixel-radius variable-k, 20% query subsample) and retaining argsort-dispatch reproduces the H100 null cleanly (12/12 cells with $|r-1| \leq 2\%$) and shows no regression on 10/12 A100 cells under

a one-sided regression test (latency ratio $\leq$ 1.02); two A100 FP32 cells (B=4, B=8 decoder) exhibit a reproducible ∼15–18% slowdown under uniformly-random ΔP synthesis. The nine A100 BF16/FP16 cells run $\sim$ 2% faster under this reordering, but the magnitude is comparable to per-trial standard deviation and we do not interpret it as a meaningful speedup.

A follow-up control replaces the uniform-random offsets with offsets produced by an init-perturbed projection layer (Gaussian weights, σ=0.01), making them input-dependent. The A100 FP32 asymmetry vanishes: across five $B \in \{2, 4, 8\}$ BF16/FP32 cells, every cell falls within ±0.2% of linear (mean ratios 0.999–1.002, three-trial means). The A100 FP32 effect is therefore a property of the uniform-random ΔP synthesis distribution rather than of sampling-point clustering itself, and disappears under more realistic offsets.

G Supplementary Figures

This appendix collects the full set of generated benchmark visualizations. Figure 1 appears in the main body; Figures 5 and 6 are included here for completeness.

```python
# Backward: each program processes one (batch, query, head) tile
# of channels. Gradient writes to grad_value use sem="relaxed"
# atomics to skip the L1 coherence sectors that would otherwise
# serialize contending warps at the same cache line.
@triton.jit
def backward_kernel(
    value_ptr, loc_ptr, attn_ptr,
    grad_out_ptr, grad_value_ptr, grad_loc_ptr, grad_attn_ptr,
    # ... strides and shapes elided ...
):
    b  = tl.program_id(0)
    q  = tl.program_id(1)
    g  = tl.program_id(2)

    # For each of the L * K sample points in this (batch, query, head)
    # tile, compute the bilinear neighbour offsets and gradient share.
    for lk in range(L_TIMES_K):
        loc_w = tl.load(loc_ptr + q * stride_locq + lk * 2    )
        loc_h = tl.load(loc_ptr + q * stride_locq + lk * 2 + 1)
        attn  = tl.load(attn_ptr + q * stride_attnq + lk)

        wx0, wy0, wx1, wy1, p00, p01, p10, p11, m00, m01, m10, m11 = \
            bilinear_neighbours(loc_w, loc_h, spatial_shape(lk))

        weighted_grad = grad_out * attn

        # Four relaxed atomic scatter-accumulations per neighbour.
        gv = grad_value_ptr
        tl.atomic_add(gv + p00 * stride_gvs + d * stride_gvd,
                      (weighted_grad * wx0 * wy0).to(gv.dtype.element_ty),
                      mask=m00, sem="relaxed")
        tl.atomic_add(gv + p01 * stride_gvs + d * stride_gvd,
                      (weighted_grad * wx1 * wy0).to(gv.dtype.element_ty),
                      mask=m01, sem="relaxed")
        tl.atomic_add(gv + p10 * stride_gvs + d * stride_gvd,
                      (weighted_grad * wx0 * wy1).to(gv.dtype.element_ty),
                      mask=m10, sem="relaxed")
        tl.atomic_add(gv + p11 * stride_gvs + d * stride_gvd,
                      (weighted_grad * wx1 * wy1).to(gv.dtype.element_ty),
                      mask=m11, sem="relaxed")

        # grad_loc and grad_attn write to disjoint output tensors;
        # no concatenated gradient buffer is materialized.
        tl.store(grad_loc_ptr  + q * stride_glq + lk * 2,     grad_w)
        tl.store(grad_loc_ptr  + q * stride_glq + lk * 2 + 1, grad_h)
        tl.store(grad_attn_ptr + q * stride_gaq + lk,         grad_a)
```

Figure 4: Relaxed-ordering scattered atomic accumulation in the Triton backward kernel. Relaxed memory-ordering semantics on the atomic-addition intrinsic instruct the Triton compiler to elide the L1 coherence-sector traffic that the default acquire-release semantics would require around each transaction, which is safe for gradient accumulation because only the final value is observed. On SM 8.0 (A100) the relaxed path is necessary but not sufficient: the software compare-and-swap loop for BF16 still dominates at encoder scale (Section 3.3). On SM 9.0 (H100) the relaxed path combined with hardware BF16 atomics recovers full memory-pipeline throughput.

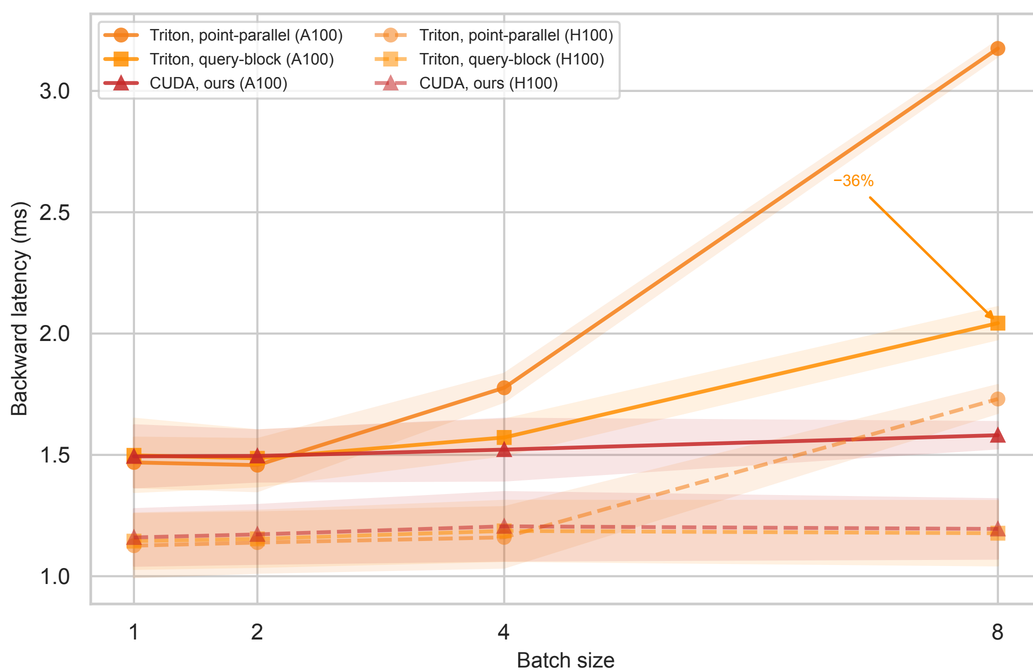

Figure 5: Backward-pass latency as a function of batch size at BF16, decoder configuration. Solid lines denote A100; dashed lines denote H100. The Triton point-parallel tiling (orange circles) diverges sharply from the query-block tiling (orange squares) at $B \geq 4$ on A100, reflecting the atomic-contention scaling analyzed in Section 3.3.

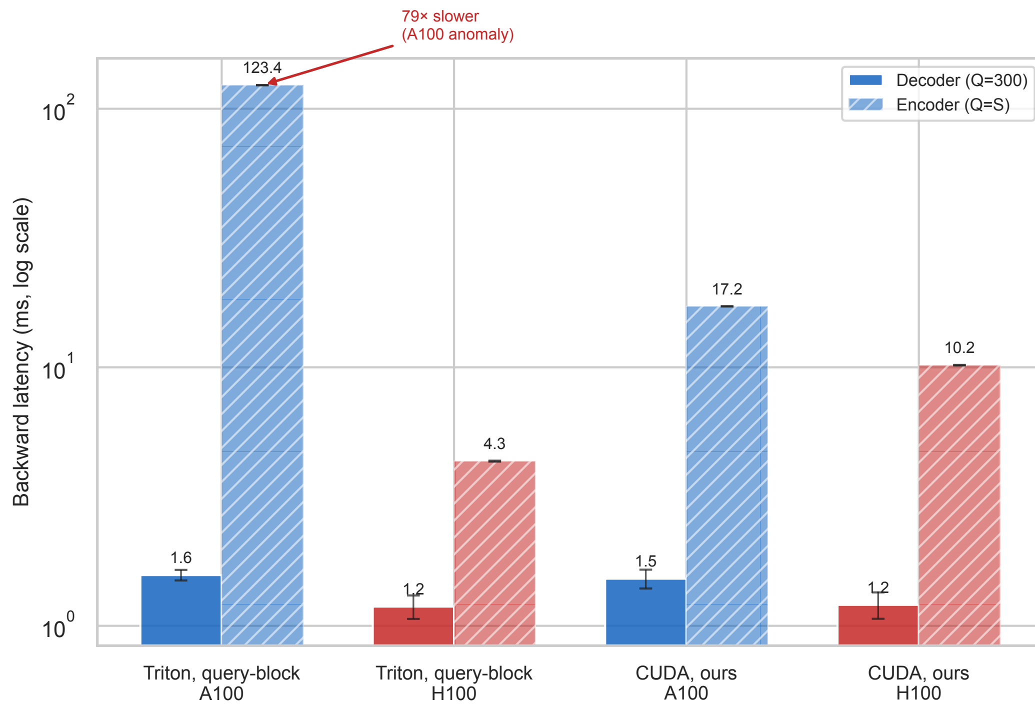

Figure 6: Decoder-scale versus encoder-scale backward latency (BF16, log scale) for both programming models on both GPUs. At encoder scale on A100, the Triton query-block backward takes 123.4 ms (79× its decoder-scale value), while the CUDA backward takes 17.2 ms. On H100 the Triton kernel recovers to 4.3 ms, faster than the CUDA path (10.2 ms), illustrating the architectural crossover analyzed in Sections 3.3 and 4.1.