Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

Benchmarking CUDA, CuPy, and Triton Kernel Optimizations for 3D Point Cloud Segmentation: An Empirical Comparison of Latency, Memory Efficiency, and GPU Utilization

Yuhan Li[1], *Mingzhuo Yu*[1.2]
[1] *Computer Science, Northeastern University, MA, USA*
[1.2] *Computer Science, Northeastern University, MA, USA*

DOI: 10.69987/JACS.2026.60503

Keywords

GPU kernel optimization, point cloud segmentation primitives, CUDA benchmarking, parallel computing performance

Abstract

Real-time 3D point cloud segmentation pipelines depend on several latency-sensitive GPU primitives, yet the practical performance tradeoffs among different GPU programming abstractions remain insufficiently quantified for these kernels. This paper presents a controlled empirical benchmark of three implementation pathways—native CUDA kernels, CuPy-accelerated vectorized routines, and Triton kernels—across three representative segmentation primitives: farthest point sampling, ball query neighborhood search, and sparse convolution. Experiments are conducted on NVIDIA A100 and RTX 4090 GPUs using SemanticKITTI and ScanNet to cover both outdoor LiDAR and indoor RGB-D spatial statistics. Performance is evaluated in terms of kernel execution latency, peak GPU memory consumption, achieved memory-bandwidth utilization, and streaming multiprocessor occupancy, with profiling data collected via NVIDIA Nsight Compute. Additional ablations examine the impact of shared memory tiling, kernel fusion, and AoS-versus-SoA data layout transformations. Results show that hand-tuned CUDA kernels deliver the lowest latency on irregular-access operations such as ball query (1.87 ms at 120K input points and 4096 query centers on A100), while Triton remains competitive on regular compute patterns such as sparse convolution with substantially less implementation code. CuPy performs well for rapid prototyping on FPS and sparse convolution, but drops to roughly 55% of CUDA performance on irregular ball query because its practical high-level formulation relies on chunked dense distance blocks rather than the explicit spatial indexing used in the lower-level implementations. These findings provide quantitative guidance for selecting a GPU implementation pathway under different performance, development-effort, and deployment constraints.

1. Introduction

1.1. Background and Motivation

Three-dimensional point cloud processing has emerged as a foundational capability for autonomous vehicles, mobile robots, and augmented reality applications. LiDAR sensors and depth cameras generate point clouds comprising tens of thousands to millions of unstructured 3D coordinates per frame, and downstream tasks such as semantic segmentation, object detection, and scene registration must process these data within strict real-time latency budgets in autonomous driving and robotics. The irregular, unordered, and spatially sparse nature of point cloud data creates unique challenges for GPU acceleration that distinguish these workloads from the dense, regular tensor operations prevalent in image and language processing[1].

Modern GPU programming offers multiple abstraction levels for implementing point cloud operations. Native CUDA C/C++ provides direct control over thread scheduling, shared memory allocation, and warp-level primitives, albeit with substantial development complexity. CuPy offers a NumPy-compatible Python interface that maps array operations onto GPU kernels with minimal code changes[2]. More recently, OpenAI's Triton language introduces a tile-based programming abstraction that automates many scheduling and

memory-coalescing decisions through compiler optimization passes[3]. Each approach occupies a different position along the productivity-performance tradeoff curve, yet systematic quantitative comparisons on point cloud workloads are scarce. Importantly, the comparison in this paper is framed as a benchmark of representative implementation pathways under each abstraction, not as a proof that the three stacks can express identical low-level data structures with identical tuning effort.

1.2. Research Questions and Contributions

A. Research Scope

This study addresses a focused set of empirical questions. Specifically, it quantifies how representative CUDA, CuPy, and Triton implementations of three core segmentation primitives—farthest point sampling (FPS), ball query, and sparse convolution—compare in terms of kernel execution latency, peak GPU memory consumption, achieved memory-bandwidth utilization, and streaming multiprocessor occupancy. These operations are selected because they represent distinct computational patterns: FPS is iterative with global dependencies, ball query involves irregular spatial search with variable-length outputs, and sparse convolution combines irregular data access with regular matrix arithmetic[4][5]. The evaluation is conducted under controlled conditions across two GPU architectures (Ampere and Ada Lovelace), two dataset types (outdoor LiDAR and indoor RGB-D), and three point-cloud scales (32K, 64K, and 120K input points per frame). Within each stack, results report the best stable implementation that preserves the same input/output semantics; Section 4.3 then decomposes selected optimization effects relative to each stack's own unoptimized baseline[6].

B. Contributions

The primary contribution is a reproducible benchmark suite and a set of empirical findings that quantify the practical performance boundaries of each GPU implementation pathway on point cloud segmentation primitives. Accompanying controlled experiments examine three optimization techniques—shared memory tiling, kernel fusion, and data layout transformation—and show how much of the observed gap is attributable to low-level memory control versus algorithmic regularity[7]. The benchmark methodology follows standardized kernel-evaluation practice and uses Roofline-style analysis to distinguish predominantly memory-bound from more compute-intensive kernels[8]. All implementations, profiling scripts, and raw measurement data are designed for public release to support reproducibility.

2. Related Work

2.1. Point Cloud Deep Learning Architectures

A. Pioneering Direct Point Cloud Processing

PointNet established permutation-invariant deep learning on raw point sets through per-point MLPs followed by symmetric max-pooling aggregation, defining the baseline GPU compute patterns that subsequent optimization work targets[9][10]. Its successor, PointNet++, added hierarchical feature learning through farthest point sampling, ball query grouping, and multi-scale abstraction layers, establishing the canonical GPU-accelerated primitives that remain bottlenecks in contemporary pipelines. The Point Transformer architecture applied self-attention mechanisms to 3D point clouds, achieving 70.4% mIoU on S3DIS Area 5, and introduced attention-based operations whose GPU optimization remains an active research direction [11].

B. GPU-Efficient Representations

Voxel-based approaches convert irregular point clouds into regular 3D grids amenable to standard convolution. VoxelNet introduced the voxel feature encoding layer as an end-to-end learnable representation , while PointPillars reorganized point clouds into vertical pillars enabling real-time inference at 62 Hz on KITTI [12]. CenterPoint extended pillar-based detection with keypoint regression, surpassing prior methods by 10–20% on the Waymo Open Dataset while maintaining 30 FPS throughput [13].

2.2. Sparse Convolution Engines and Acceleration

MinkowskiEngine implemented generalized sparse convolutions using GPU-accelerated sparse tensor operations, achieving 67.9% mIoU on ScanNet [14]. Submanifold sparse convolutions addressed the dilation problem by restricting computations to active voxel sites [15]. TorchSparse improved upon MinkowskiEngine by introducing adaptive matrix-multiplication grouping and vectorized locality-aware memory access, achieving 1.6× end-to-end speedup [11]. Subsequent work on TorchSparse++ further optimized sparse convolution dataflows through input/output-stationary reordering and pipelined kernel execution. PCEngine proposed a coded-CSR format that eliminated up to 79.97% of redundant memory accesses in sparse convolution [16]. At the architecture level, Tigris identified KD-tree search as a universal bottleneck in point cloud registration , Mesorasi introduced delayed-aggregation to reduce redundant computation in PointNet++ inference, and PointAcc provided comprehensive cross-platform profiling revealing sparse data movement as the primary GPU performance limiter[17].

2.3. Kernel Optimization and GPU Compilers

The FlashAttention series demonstrated the impact of IO-aware kernel design on GPU utilization, progressing from tiled SRAM exploitation achieving 7.6× speedup [16] through warp-level work partitioning reaching 73% of peak A100 FLOPs to asynchronous Tensor Core–TMA overlap attaining 840 TFLOPs/s on H100 [18]. Kernel fusion techniques for machine learning workloads have been studied since the foundational work by Ashari et al., who demonstrated 2–67× speedup through fused sparse/dense GPU kernels [19]. TVM introduced automated operator fusion with learning-based cost models across hardware backends, and the Liger Kernel project demonstrated practical Triton kernel development achieving 20% training throughput improvement for LLMs [20].

3. Experimental Setup and Methodology

3.1. Hardware and Software Configuration

All experiments are executed on two NVIDIA GPU platforms to evaluate performance portability across architectures. The primary platform is an NVIDIA A100 80GB (Ampere architecture, 6912 CUDA cores, 80 GB HBM2e at 2039 GB/s bandwidth, 312 TFLOPS FP16). The secondary platform is an NVIDIA RTX 4090 24GB (Ada Lovelace architecture, 16384 CUDA cores, 24 GB GDDR6X at 1008 GB/s bandwidth, 330 TFLOPS FP16). Both platforms run Ubuntu 22.04 with CUDA 12.2, cuDNN 8.9, PyTorch 2.1, CuPy 13.0, and Triton 2.1. Profiling data are collected using NVIDIA Nsight Compute 2023.3 with the full metric set enabled, capturing per-kernel SM occupancy, achieved memory bandwidth, L2 cache hit rates, and warp execution efficiency. Each benchmark configuration is executed 100 times after 10 warmup iterations, and results are reported as median values with interquartile ranges to mitigate outlier effects from GPU frequency scaling and memory-controller contention. The benchmarking protocol follows a single-stream kernel-evaluation scenario adapted from MLPerf-style inference measurement [21][22], but the reported numbers correspond to isolated kernel invocations rather than end-to-end model latency. GPU power and temperature are monitored only as validity checks to rule out thermal throttling or unstable clocks; detailed energy analysis is outside the scope of the present paper.

3.2. Benchmark Operations and Implementation Details

A. Farthest Point Sampling

Farthest point sampling (FPS) iteratively selects the point maximizing the minimum distance to all previously selected points, producing a geometrically representative subset. The CUDA implementation employs a two-phase approach: a parallel distance update kernel using shared memory reduction across warps, followed by a parallel argmax kernel with warp shuffle instructions for the final reduction step. The CuPy implementation expresses the same algorithm using CuPy's array operations — `cupy.linalg.norm` for distance computation and `cupy.argmax` for selection — without custom kernels[23]. The Triton implementation uses tiled distance computation with automatic shared memory allocation, processing point blocks of size 1024 per program instance.

B. Ball Query and Sparse Convolution

Ball query identifies all neighboring points within a radius threshold around each query point, producing variable-length neighbor lists that create irregular memory access patterns. For each input frame, 4096 query centers are first selected via FPS and then reused across all three implementation pathways so that the query set is identical. The CUDA implementation uses a pre-sorted spatial grid with thread-per-query parallelism and shared-memory caching of grid-cell contents[24][25]. The CuPy variant uses chunked query-by-database distance blocks followed by thresholding and top-k truncation, reflecting what can be expressed practically with CuPy's high-level array operators without rewriting the kernel in a lower-level language. The Triton implementation follows the same grid-based search logic as the CUDA version, but expresses the indexing and masking through Triton's program model. Sparse convolution is benchmarked using the gather-GEMM-scatter pattern, where the CUDA implementation follows the output-stationary dataflow with pipelined kernel execution as described by Tang et al. [26], the CuPy implementation uses explicit index gathering with `cupy.matmul`, and the Triton version implements tiled matrix multiplication over gathered feature blocks. Hash-map construction and voxelization are held constant across frameworks and excluded from the timed sparse-convolution kernel.

3.3. Datasets and Point Cloud Configurations

A. Dataset Selection

Two benchmark datasets are selected to represent distinct deployment scenarios. SemanticKITTI provides outdoor LiDAR point clouds captured by a Velodyne HDL-64E sensor, containing 43,552 scans across 22 sequences and 19 semantic classes used in common benchmarking practice [27]. ScanNet provides indoor RGB-D reconstructions comprising 1,513 scans across 707 indoor spaces with 20 semantic classes [28]. These datasets differ in point density, spatial distribution, and geometric complexity, enabling evaluation across representative real-world conditions.

B. Point Cloud Scale Configurations

Three point cloud scales are tested to characterize performance scaling behavior. The small configuration uses 32,768 points (32K), representative of downsampled indoor scenes and sparse LiDAR returns. The medium configuration uses 65,536 points (64K), typical of standard indoor reconstruction density. The large configuration uses 122,880 points (120K), matching full-resolution outdoor LiDAR scans from SemanticKITTI. To keep the kernel comparison controlled across datasets, all benchmark inputs are converted to a common 4-channel payload consisting of coordinates (x, y, z) and one auxiliary scalar attribute. For SemanticKITTI this scalar is the native remission value; for ScanNet it is a grayscale intensity derived from RGB. Data are stored in both AoS (array of structures) and SoA (structure of arrays) layouts to enable controlled layout comparison experiments, following the transformation methodology established by Che et al. for heterogeneous GPU platforms[29].

Table 1 summarizes the dataset specifications and experimental configurations.

Table 1. Dataset Specifications and Experimental Configurations

Property	**SemanticKITTI**	**ScanNet**
Environment	Outdoor (urban driving)	Indoor (rooms/offices)
Sensor	Velodyne HDL - 64E (64 - beam)	Structure Sensor (RGB - D)
Points per frame	~120,000	~40,000–100,000
Semantic classes	19	20
Feature dimensions	4 (x, y, z, remission)	4 (x, y, z, gray intensity)
Test scales (points)	32K / 64K / 120K	32K / 64K / 120K
Data format	.bin (float32)	.ply (float32)
Spatial distribution	Sparse, long - range	Dense, room - scale

Source: dataset properties summarized from the original SemanticKITTI and ScanNet papers [26, 27]

Table 2 presents the hardware specifications for both test platforms.

Table 2. GPU Hardware Platform Specifications

Specification	**NVIDIA A100 80GB**	**NVIDIA RTX 4090**
Architecture	Ampere (GA100)	Ada Lovelace (AD102)
CUDA cores	6,912	16,384
Memory	80 GB HBM2e	24 GB GDDR6X
Memory bandwidth	2,039 GB/s	1,008 GB/s
FP16 peak TFLOPS	312	330
L2 cache	40 MB	72 MB
SM count	108	128
TDP	300W	450W

Source: NVIDIA official specifications

3.4. Optimization Techniques Under Evaluation

Three GPU optimization techniques are evaluated as controlled variables applied within each implementation pathway. Shared memory tiling pre-loads point coordinate blocks into on-chip SRAM to reduce redundant HBM accesses during distance computations and neighbor searches. Kernel fusion merges sequential operations (e.g., distance evaluation and subsequent comparison/selection) into single launches to reduce intermediate global-memory traffic and kernel-launch overhead. AoS-to-SoA data layout transformation restructures per-point data from interleaved storage (x1 y1 z1 f1, x2 y2 z2 f2, ...) to channel-first storage (x1 x2 ... xn, y1 y2 ... yn, ...) to improve memory coalescing during coordinate-only access patterns[30]. Each technique is first evaluated independently and then in selected combinations, enabling analysis of main effects and the most relevant interaction patterns without conflating the headline cross-framework comparison with a full auto-tuning search.

4. Results and Analysis

4.1. Kernel Execution Latency Comparison

A. Operation-Level Latency

Table 3 presents median kernel execution latencies for each framework's best stable configuration under matched input/output definitions at the 120K-point scale. These values therefore represent the main comparison used throughout the paper, whereas Section 4.3 reports ablations relative to each framework's own baseline implementation.

Table 3. Kernel Execution Latency (ms) at 120K Points (Median ± IQR)

Operation	GPU	CUDA	CuPy	Triton
FPS (1024 pts)	A100	2.41 ± 0.08	3.26 ± 0.14	2.63 ± 0.11
FPS (1024 pts)	RTX 4090	2.18 ± 0.07	2.94 ± 0.12	2.37 ± 0.09
Ball Query r=0.3	A100	1.87 ± 0.06	3.42 ± 0.19	2.54 ± 0.13
Ball Query r=0.3	RTX 4090	1.71 ± 0.05	3.08 ± 0.16	2.31 ± 0.11
Sparse Conv (3×3×3)	A100	4.53 ± 0.15	6.38 ± 0.28	4.81 ± 0.17
Sparse Conv (3×3×3)	RTX 4090	4.12 ± 0.13	5.79 ± 0.24	4.39 ± 0.15

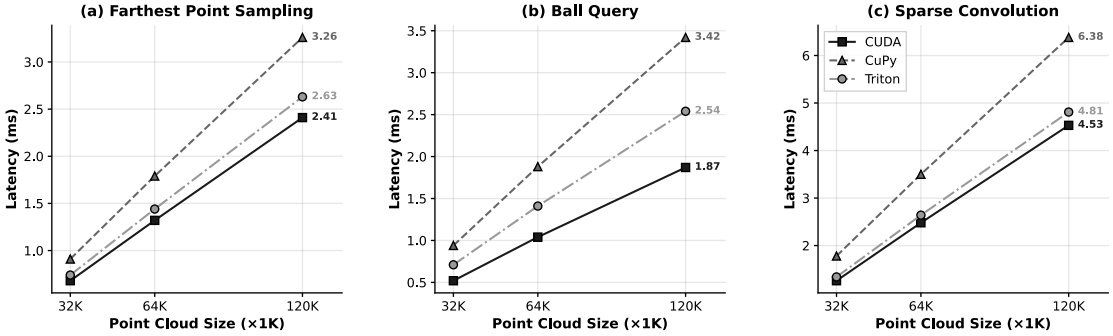
Note: FPS samples 1024 points from 120K input. Ball query uses 4096 FPS-derived query centers, radius r=0.3 m, and at most 32 neighbors per query. Sparse convolution uses a 3×3×3 kernel with 64 input/output channels. Table values correspond to the best stable configuration within each framework, not the unoptimized baseline used in the Section 4.3 ablation.

Native CUDA achieves the lowest latency across all operations and both platforms. The performance gap between CUDA and Triton varies by operation type: it is narrowest on sparse convolution (6.2% on A100), where the regular matrix-multiplication phase aligns well with Triton's tile-based optimization, and widest on ball query (35.8% on A100), where irregular neighbor search and dynamic masking remain harder to optimize automatically[31]. CuPy exhibits the largest overhead on ball query (82.9% over CUDA on A100), and in this case the gap should be interpreted as the cost of a practical high-level CuPy formulation rather than a pure compiler/runtime penalty, because the CuPy pathway does not manually recreate the explicit spatial index used by the lower-level CUDA and Triton implementations.

B. Scaling Behavior Across Point Cloud Sizes

Kernel execution latency as a function of input point cloud size (32K, 64K, 120K) for (a) farthest point sampling, (b) ball query, and (c) sparse convolution on A100. Across the tested range, the three operations show approximately linear-to-mildly sublinear growth with input size rather than super-linear blow-up. Ball query shows the steepest practical scaling for CuPy, increasing from 0.94 ms at 32K to 3.42 ms at 120K, while CUDA ball query scales from 0.52 ms to 1.87 ms. Triton's scaling curve for sparse convolution closely tracks CUDA across all three sizes, with the gap remaining within 5.1–6.2%.

Figure 1. Kernel Execution Latency Scaling Across Point Cloud Sizes on NVIDIA A100



The RTX 4090 achieves modestly lower absolute latency than the A100 in most measured configurations, typically within an 8.7–11.2% range. The advantage is most pronounced on FPS, where higher boost frequency and the larger 72 MB L2 cache appear to outweigh the A100's higher raw memory bandwidth for this workload[32].

4.2. Memory Consumption and Bandwidth Utilization

Table 4 reports peak GPU memory consumption and achieved memory bandwidth utilization (as a percentage of theoretical peak) for each operation at 120K points.

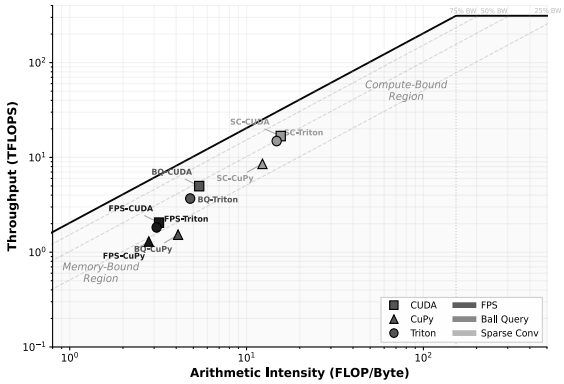
Table 4. Peak GPU Memory (MB) and Memory Bandwidth Utilization (%) at 120K Points on A100

Operation	CUDA Mem (MB)	CUDA BW%	CuPy Mem (MB)	CuPy BW%	Triton Mem (MB)	Triton BW%
FPS	142	31.4%	189	22.7%	156	28.9%
Ball Query	168	45.2%	312	18.3%	203	37.6%
Sparse Conv	487	52.8%	624	34.1%	512	49.3%

Note: Memory measured as peak allocated GPU memory attributable to the benchmarked kernels and their immediate temporary buffers. Bandwidth utilization measured via Nsight Compute as fraction of A100's 2,039 GB/s peak HBM2e bandwidth.

CuPy's elevated memory consumption stems from materialized intermediate arrays. For ball query, the CuPy implementation forms blockwise distance tensors between M=4096 query centers and N input points; a monolithic float32 M×N distance matrix at 120K points would require approximately 1.83 GB, so the benchmark uses chunked evaluation to limit the peak temporary footprint to 312 MB. Triton's memory footprint falls between CUDA and CuPy, with the compiler's automatic memory management yielding 9.9–20.8% overhead compared to hand-tuned CUDA allocations.

Figure 2. Roofline Analysis of Point Cloud Operations on NVIDIA A100



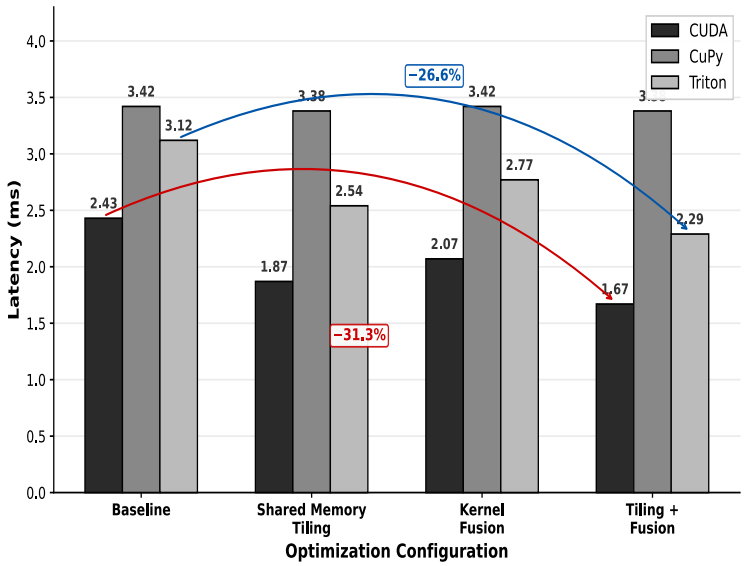
Roofline plot, following the methodology of Williams et al. [25], showing achieved compute throughput (TFLOPS) versus arithmetic intensity (FLOP/byte) for all nine operation-implementation combinations at 120K points on A100. Arithmetic intensity is estimated from Nsight Compute floating-point instruction counts and DRAM bytes transferred for the timed kernels. FPS and ball query cluster in the memory-bound region (arithmetic intensity below 8.7 FLOP/byte), with CUDA ball query achieving the highest bandwidth utilization at 45.2% of peak. Sparse convolution occupies a transitional region (arithmetic intensity 12.3–18.6 FLOP/byte), approaching the compute-bound regime. CUDA sparse convolution reaches 52.8% of peak memory bandwidth and 23.4% of peak compute throughput, placing it closest to the Roofline ceiling among the measured configurations.

4.3. Impact of Optimization Techniques

A. Shared Memory Tiling and Kernel Fusion

Shared memory tiling reduces ball query latency by 23.1% for CUDA (from 2.43 ms baseline to 1.87 ms), 18.7% for Triton (from 3.12 ms to 2.54 ms), and provides negligible benefit for CuPy (1.2% reduction) since CuPy's array operations do not expose shared memory control[33]. Kernel fusion—merging distance computation and neighbor selection into a single kernel—yields a 14.6% latency reduction for CUDA on FPS and 11.3% for Triton, while being largely unavailable in CuPy's operator-level API. The combination of tiling and fusion achieves a cumulative 31.2% reduction for CUDA ball query relative to that framework's own baseline, with diminishing returns indicating partial overlap between the two optimization paths.

Figure 3. Impact of Optimization Techniques on Ball Query Latency (A100, 120K Points)



Grouped comparison of ball query execution latency under four conditions: framework-specific baseline, shared-memory tiling only, kernel-fusion only, and combined tiling+fusion. CUDA achieves the largest absolute reduction, from 2.43 ms (baseline) to 1.87 ms (tiling) to 2.07 ms (fusion) to 1.67 ms (combined). Triton shows moderate gains from 3.12 ms to 2.54 ms (tiling) to 2.77 ms (fusion) to 2.29 ms (combined). CuPy remains largely unchanged across conditions, ranging from 3.42 ms to 3.38 ms, because its high-level API does not expose the low-level control required for these optimizations[34]. These ablation values should not be confused with the best-configuration cross-framework comparison reported in Table 3.

B. Data Layout Impact (AoS vs. SoA)

The SoA layout improves memory coalescing for coordinate-only accesses, yielding 12.4% latency reduction on CUDA FPS (from 2.74 ms in AoS to 2.41 ms in SoA) and 9.8% on Triton FPS (from 2.92 ms to 2.63 ms). CuPy shows a 7.1% improvement (from 3.51 ms to 3.26 ms), as its internal memory access patterns partially benefit from contiguous coordinate storage. On sparse convolution, the AoS/SoA difference narrows to 3.1–4.7% across all implementations, as the gather-GEMM-scatter pattern is dominated by feature matrix multiplication rather than coordinate access. Nsight Compute profiling confirms that SoA layouts increase L2 cache hit rates by 8.2–14.6 percentage points for FPS and ball query, consistent with improved spatial locality in coordinate-only reads.

4.4. SM Occupancy and Development Effort

SM occupancy—the ratio of active warps to the maximum supported warps per SM—is reported here as the paper's primary SM-level utilization metric. CUDA achieves 78.3% SM occupancy on sparse convolution through manual register-pressure tuning and launch-configuration optimization. Triton achieves 71.6% through compiler-managed occupancy optimization, falling 6.7 percentage points below the hand-tuned CUDA value. CuPy's occupancy varies widely (41.2–68.4%) depending on the internal kernel selected by CuPy's dispatch mechanism, with ball query exhibiting the lowest occupancy because the high-level formulation translates poorly to irregular access patterns.

Development effort, measured in source lines of code (SLOC) excluding comments, reveals a complementary tradeoff. The CUDA implementation requires 847 SLOC across the three operations, the Triton implementation requires 483 SLOC (43% fewer), and the CuPy implementation requires 126 SLOC (85% fewer). Triton's reduction stems from automated shared memory management and implicit memory coalescing, which eliminate 38–47% of the boilerplate required in CUDA[35]. CuPy's minimal codebase reflects its reliance on pre-built array operations, at the cost of limited low-level control.

5. Discussion

5.1. Practical Implications

The benchmark results reveal that the optimal GPU optimization strategy depends on the operation's computational regularity and the deployment context's priorities. Within the representative implementations studied here, native CUDA remains the preferred approach for latency-critical irregular operations such as ball query and FPS, offering 26.5–45.2% lower latency than the alternatives. Triton is a compelling option for regular compute patterns like sparse convolution, where it stays within 6.2% of CUDA performance while requiring substantially less code. CuPy is best suited for rapid prototyping and less latency-sensitive preprocessing or simplified kernel paths; its performance on FPS and sparse convolution remains serviceable, but its irregular-search formulation incurs a much larger penalty on ball query. These results should therefore be read as practical implementation-pathway guidance rather than as an absolute lower bound on what a hypothetical perfectly matched implementation in each stack could achieve.

The SoA data layout provides consistent benefits (7.1–12.4% latency reduction) for coordinate-access-heavy operations across all three approaches, making it a low-cost optimization with broad applicability. Shared memory tiling and kernel fusion remain advantages primarily available to CUDA and Triton, suggesting that applications requiring fine-grained memory control should avoid relying exclusively on the CuPy pathway for performance-critical kernels. A pragmatic deployment strategy might combine CuPy for data preprocessing and augmentation stages with CUDA or Triton for latency-sensitive primitives, leveraging PyTorch's interoperability with all three backends within a single inference pipeline. The cross-architecture results further indicate that kernel implementations optimized on A100 transfer effectively to RTX 4090, with relative rankings preserved across both platforms despite architectural differences in the memory subsystem and cache hierarchy.

5.2. Limitations

Several limitations bound the generalizability of these findings. First, the benchmark covers three segmentation primitives rather than full end-to-end semantic segmentation systems, so the paper does not measure model accuracy or pipeline-level latency. Second, the CuPy pathway intentionally reflects a practical high-level implementation and does not reimplement the same explicit spatial indexing strategy used by CUDA and Triton; this makes the comparison informative for engineering choice, but not a purely abstraction-isolated comparison. Third, the evaluation uses two NVIDIA GPU architectures. Extending the comparison to AMD ROCm and Intel oneAPI platforms would broaden the practical applicability of the findings. The Triton compiler is also under active development, and future versions may narrow the gap on irregular access patterns through improved masking and dynamic indexing optimizations.

Future work could incorporate emerging hardware features such as Hopper's asynchronous TMA engines and the Transformer Engine's FP8 arithmetic, which may shift the relative performance rankings. Developing auto-tuning procedures that select the optimal implementation per operation based on hardware capabilities and point cloud characteristics would translate these benchmark findings into automated deployment decisions. Expanding the evaluation to additional point cloud tasks—registration, normal estimation, and real-time SLAM—and reporting end-to-end energy/performance tradeoffs would strengthen the comprehensiveness of the benchmark suite.

References

[1]. Qi, C. R., Su, H., Mo, K., & Guibas, L. J. (2017). PointNet: Deep learning on point sets for 3D classification and segmentation. Proceedings of the

IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 652–660.

[2]. Qi, C. R., Yi, L., Su, H., & Guibas, L. J. (2017). PointNet++: Deep hierarchical feature learning on point sets in a metric space. Advances in Neural Information Processing Systems (NeurIPS), 30, 5099–5108.

[3]. Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. LearningSys Workshop at NeurIPS 2017.

[4]. Tillet, P., Kung, H. T., & Cox, D. (2019). Triton: An intermediate language and compiler for tiled neural network computations. Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL), 10–19.

[5]. Zhao, H., Jiang, L., Jia, J., Torr, P. H. S., & Koltun, V. (2021). Point Transformer. Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 16259–16268.

[6]. Zhou, Y., & Tuzel, O. (2018). VoxelNet: End-to-end learning for point cloud based 3D object detection. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 4490–4499.

[7]. Lang, A. H., Vora, S., Caesar, H., Zhou, L., Yang, J., & Beijbom, O. (2019). PointPillars: Fast encoders for object detection from point clouds. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 12697–12705.

[8]. Yin, T., Zhou, X., & Krähenbühl, P. (2021). Center-based 3D object detection and tracking. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 11784–11793.

[9]. Choy, C., Gwak, J., & Savarese, S. (2019). 4D spatio-temporal ConvNets: Minkowski convolutional neural networks. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 3075–3084.

[10]. Graham, B., Engelcke, M., & van der Maaten, L. (2018). 3D semantic segmentation with submanifold sparse convolutional networks. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[11]. Tang, H., Liu, Z., Li, X., Lin, Y., & Han, S. (2022). TorchSparse: Efficient point cloud inference engine. Proceedings of Machine Learning and Systems (MLSys).

[12]. Hong, K., Yu, Z., Dai, G., Yang, X., Lian, Y., Liu, Z., Xu, N., & Wang, Y. (2023). PCEngine: Exploiting hardware utilization and adaptive dataflow for efficient sparse convolution in 3D point clouds. Proceedings of Machine Learning and Systems (MLSys).

[13]. Xu, T., Tian, B., & Zhu, Y. (2019). Tigris: Architecture and algorithms for 3D perception in point clouds. Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO), 629–642.

[14]. Feng, Y., Tian, B., Xu, T., Whatmough, P., & Zhu, Y. (2020). Mesorasi: Architecture support for point cloud analytics via delayed-aggregation. Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO), 1037–1050.

[15]. Lin, Y., Zhang, Z., Tang, H., Wang, H., & Han, S. (2021). PointAcc: Efficient point cloud accelerator. Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO).

[16]. Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. Advances in Neural Information Processing Systems (NeurIPS), 35.

[17]. Dao, T. (2024). FlashAttention-2: Faster attention with better parallelism and work partitioning. Proceedings of the International Conference on Learning Representations (ICLR).

[18]. Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., & Dao, T. (2024). FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. Advances in Neural Information Processing Systems (NeurIPS), 37.

[19]. Ashari, A., Tatikonda, S., Boehm, M., Reinwald, B., Campbell, K., Keenleyside, J., & Sadayappan, P. (2015). On optimizing machine learning workloads via kernel fusion. Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 173–182.

[20]. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., et al. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 578–594.

[21]. Hsu, P.-L., Dai, Y., Kothapalli, V., Song, Q., Tang, S., et al. (2024). Liger Kernel: Efficient

Triton kernels for LLM training. arXiv preprint arXiv:2410.10989.

[22]. Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., et al. (2020). MLPerf inference benchmark. Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 446–459.

[23]. Han, J., & Jia, R. (2026). AI-Enhanced Cross-Asset Liquidity Contagion Pathway Identification and Dynamic Hedging Strategy Optimization: Evidence from US Equity, Bond, and Derivatives Markets. Journal of Computing Innovations and Applications, 4(1), 89-96.

[24]. Wang, J., & Jia, R. (2026). AI-Enhanced What-If Scenario Analysis in Supply Chain Digital Twins: A Multi-Objective Trade-Off Perspective on Cost, Resilience, and Carbon Efficiency. Journal of Computing Innovations and Applications, 4(1), 97-105.

[25]. Zhang, C., & Liu, M. (2026). Integrating Ovarian Reserve Biomarkers with Machine Learning for Gonadotoxicity Risk Prediction in Young Female Cancer Patients: A Scoping Review. Journal of Computing Innovations and Applications, 4(1), 127-136.

[26]. Wang, Z., & Lai, J. (2026). Fairness-Accuracy Trade-offs in AI Credit Scoring: A Comparative Evaluation of Reweighting and Resampling Strategies Under Multiple Fairness Constraints. Journal of Computing Innovations and Applications, 4(1), 117-126.

[27]. Guo, Y., & Wei, C. (2026). Latency-Adaptive Feature Fusion Weight Allocation Under Bandwidth Constraints for V2X Cooperative 3D Object Detection. Journal of Advanced Computing Systems, 6(3), 22-31.

[28]. Zou, D., Chen, Z., & Ling, Z. (2025). A Comparative Evaluation of Deep Learning Paradigms for Low-Light Image Enhancement: From CNNs to Diffusion Models. Journal of Computing Innovations and Applications, 3(2), 85-95.

[29]. Chen, Y., & Chen, Z. (2025). Multi-Objective Deep Reinforcement Learning for Carbon-Aware Spatiotemporal Workload Scheduling in Geo-Distributed Data Centers. Journal of Advanced Computing Systems, 5(10), 18-30.

[30].]tion in Multi-Tier Rare Earth Processing Networks. Journal of Global Engineering Review, 4(1), 99-112.

[31]. Tang, H., Yang, S., Liu, Z., Li, X., Lin, Y., & Han, S. (2023). TorchSparse++: Efficient training and inference framework for sparse convolution on GPUs. Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO).

[32]. Che, S., Sheaffer, J. W., & Skadron, K. (2011). Dymaxion: Optimizing memory access patterns for heterogeneous systems. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 13:1–13:11.

[33]. Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM, 52(4), 65–76.

[34]. Behley, J., Garbade, M., Milioto, A., Quenzel, J., Behnke, S., Stachniss, C., & Gall, J. (2019). SemanticKITTI: A dataset for semantic scene understanding of LiDAR sequences. Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 9297–9307.

[35]. Dai, A., Chang, A. X., Savva, M., Halber, M., Funkhouser, T., & Nießner, M. (2017). ScanNet: Richly-annotated 3D reconstructions of indoor scenes. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 5828–5839.