2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3315-1596-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICPADS63350.2024.00057

Efficient Point Cloud Analytics on Edge Devices

Kunxiong Zhu
Microelectronics Thrust
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
kunxiongzhu@hkust-gz.edu.cn

Zhenlin Wu
Microelectronics Thrust
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
zwu065@connect.hkust-gz.edu.cn

Hongyuan Liu
Microelectronics Thrust
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
hongyuanliu@hkust-gz.edu.cn

***Abstract*—Point clouds are crucial for 3D geometry representation, and vital in applications like autonomous driving and augmented reality. Despite advancements in deep learning-based analytics, their high computational cost limits deployment on edge devices with constrained resources.**

To this end, we analyze PointNet++, a leading point cloud analytics framework, identifying two major bottlenecks: 1) GPU is underutilized due to limited parallelism and excessive kernel launches in the sampling stage and voting stage, and 2) irregular memory accesses in the grouping stage. To address these, we propose parallel sampling and voting to enhance GPU utilization and fuse subroutines in grouping to improve memory efficiency. Experimental results demonstrate that our optimizations result in significant speedup (up to 5.0×, 3.2× on average) across various point cloud workloads on edge devices.

I. Introduction

Point clouds have emerged as a vital method for representing 3D geometry and attributes of objects, providing more comprehensive data than traditional 2D images. This representation is crucial in fields such as autonomous driving [1] and augmented reality (AR) [2], where precise and detailed 3D data is necessary for accurate understanding of the environment.

Recent advancements in deep learning-based analytics have significantly enhanced visual perception, enabling more precise detection and measurement of object distances [3], [4]. However, these methods are computationally expensive, creating challenges for their implementation on edge devices, which typically have limited computational power [5].

To achieve faster and accurate point cloud analytics on resource-constrained edge devices, we characterize and analyze a pioneering point cloud analytics deep neural network backbone, PointNet++ [4], on edge devices. PointNet++ relies on three critical procedures, sampling, grouping, and voting, to effectively capture local features and hierarchically build more complex representations from raw point clouds. These processes are essential for handling varying densities and ensuring robust feature learning. However, they are computationally intensive.

Through detailed characterization, we made two key observations regarding the three expensive stages illustrated in Figure 1. **First**, there is an excessive number of kernel launches and global synchronization overhead. However, each kernel performs a minimal amount of computation, causing the kernel launch overhead to significantly contribute to the overall inference latency. **Second**, the grouping stage requires multiple subroutines such as sorting over large chunks of memory. The irregular accesses from these subroutines lead to poor data locality. To address the two observed bottlenecks, we propose new schemes for the performance-critical procedures.

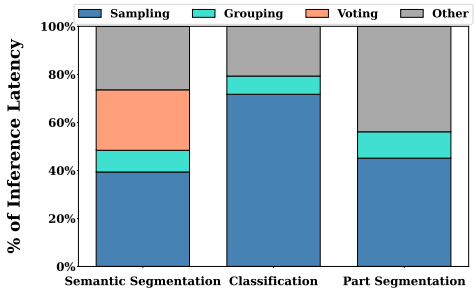


Fig. 1: Inference latency breakdown in three point cloud tasks. Evaluation methodology in Section V.

Parallelizing Sampling and Voting. We identified that the primary reason the sampling stage requires a large number of kernel launches and underutilizes GPUs is due to its iterative nature. Each iteration must wait for the previous one to complete, and not all iterations have sufficient parallelism. This results in frequent synchronization and significant overhead, leading to inefficient GPU utilization. To address this issue, we propose a straightforward yet parallelized sampling procedure for PointNet++. Our approach eliminates the iterative nature of the sampling process, allowing it to be executed in parallel on edge GPUs. Our approach maintains the goal of better coverage of the entire point set with the same number of centroids while significantly improving GPU utilization. Additionally, we also parallelize the voting stage to fully utilize the edge GPUs.

Kernel Fusion for Grouping. The implementations of the grouping stage in PointNet++ heavily rely on sorting operations in their compute kernels, resulting in poor data locality and a large amount of intermediate data. To address this issue, we optimize these stages by fusing the subroutines of grouping. This fusion allows for the reuse of intermediate data, reducing both the memory footprint and the overall computational overhead.

979-8-3315-1596-6/24/$31.00 ©2024 IEEE
DOI 10.1109/ICPADS63350.2024.00057

To the best of our knowledge, no prior work focuses on optimizing the sampling, grouping, and voting stages in PointNet++ without changing the representations of points on edge devices. In summary, we make the following contributions:

- Through detailed characterization, we identify that excessive kernel launches and poor data locality are two major bottlenecks across the performance-critical stages.
- We propose a parallel sampling method that requires only a few GPU kernel launches, thereby improving GPU utilization.
- We fuse the subroutines for the grouping stage, eliminating the need for sorting operations and thus avoiding irregular memory accesses, which enhances locality. Additionally, we parallelize the voting stage on edge GPUs.
- Experimental results show that our approach achieves a 3.2× average speedup across various point cloud workloads while maintaining almost the same inference accuracy on edge devices.

II. Background

Point Cloud. A point cloud is a collection of data points in space, often representing a 3D shape or object. Each point in the point cloud has Cartesian coordinates (X, Y, Z) and may include additional data such as RGB colors, normals, and timestamps. Point clouds are typically generated by 3D scanners or photogrammetry software, capturing the external surfaces of objects.

PointNet++ Architecture. Figure 2 presents an overview of PointNet++'s architecture. PointNet++ is an advanced deep-learning model designed for processing 3D point cloud data. It employs a hierarchical approach to progressively capture local and global features. Each layer utilizes Set Abstraction (SA) modules to extract and aggregate features from local neighborhoods. Ultimately, PointNet++ achieves efficient processing and analysis of complex 3D point clouds through multi-level feature propagation (FP). As shown in Figure 2, the stages we focus on are sampling, grouping, and voting stages.

Sampling Stage. The sampling stage performs point sampling on the input point cloud to select representative points. It is desirable for these points to be sampled in a relatively uniform manner. To achieve this, PointNet++ employs the Farthest Point Sampling (FPS) algorithm [6].

Grouping Stage. The grouping stage partitions the sampled points into groups by grouping adjacent points together. The ball query algorithm [4] is employed to find neighboring points in the original point set for each sampled point and group them around the sampled point. PointNet++ uses the ball query algorithm to identify the neighboring points of each sampled point, preparing them for the feature extraction layer.

Voting Stage. The voting stage uses the extracted feature vectors to vote for each point to determine its semantic category. PointNet++ achieves precise semantic segmentation of the entire point cloud by integrating the voting results of all points.

III. Accelerating Point Sampling via Parallelization

A. Inefficiency of Sampling Stage

Figure 1 demonstrates that the sampling stage accounts for 40% to 75% of the entire end-to-end inference process of PointNet++ across different tasks. PointNet++ adopts iterative farthest point sampling (FPS) algorithm [6], aiming to select points that evenly cover the whole set.

Iterative Workflow of FPS. Figure 3a illustrates the FPS algorithm workflow. The algorithm maintains a sampled set ("**Samples**" in Figure 3a). During each iteration, it calculates the distances between unsampled points in the dataset and the points already in the sampled set, based on a chosen distance metric. The point that is farthest from the current sampled set is then added to it. For example, when the sampled set in Figure 3a contains points 1 and 4, the distances between points outside the sampled set and points 1 and 4 are computed. In this case, d_5, d_6, and d_7 are the shortest distances. If d_5 is the largest, point 2 is added to the sampled set. This process repeats until the sampled set reaches the specified size.

GPU Underutilization. The iterative FPS algorithm was implemented using a bulk-synchronous parallel approach. Within each iteration, distances are evaluated in parallel between points outside the sampled set and those within it. However, each subsequent iteration must wait for the completion of the previous one. We observe that two primary bottlenecks result in GPU underutilization on edge devices: 1) Excessive kernel launches lead to significant overhead (further discussed in Section V). 2) The parallelism is limited especially in early iterations, and hence cannot exploit all compute resources.

B. Our Solution: Parallel Farthest Point Sampling

Key Insight and Our Approach. Similar to iterative FPS, the goal of our algorithm is to ensure that the sampled points are as far apart from each other as possible, resulting in a more representative and diverse subset of the original points. Our approach is straightforward: *To sample K points out of the N points, we select K points such that the minimum distance to all other points is maximized.*

Illustrative Example of Our Approach. Figure 3b shows an illustrative example of our approach. Initially, we compute the pairwise distances between all points in the dataset in parallel (e.g., every pair of the 5 points in Figure 3b). In contrast to the iterative FPS algorithm, our method eliminates the data dependency between adjacent iterations, and thus does not need to wait for global synchronization. Each point then computes the minimum distance to all other points. We sort the minimum distances descendingly. Finally, we select the top K points with the largest minimum distances as the sampled points.

Implementation. Our sampling implementation consists of three main steps: calculating all pairs of distance (implemented with a customized CUDA kernel), computing the minimum values, and finding the top K maximum values. For the customized CUDA kernel, we allocate $B \times N \times (N-1)$ threads

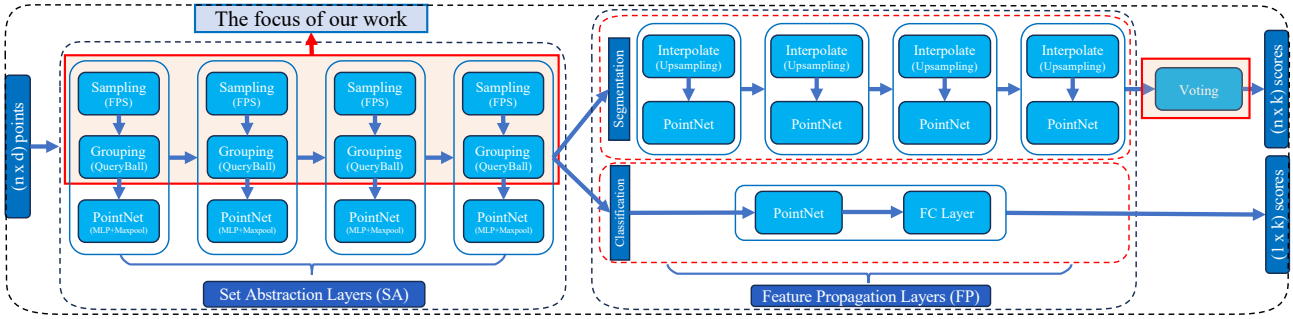
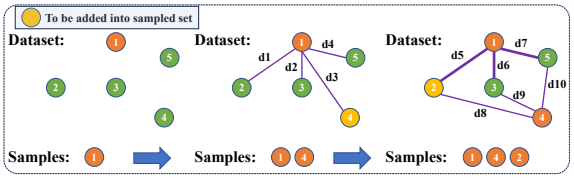
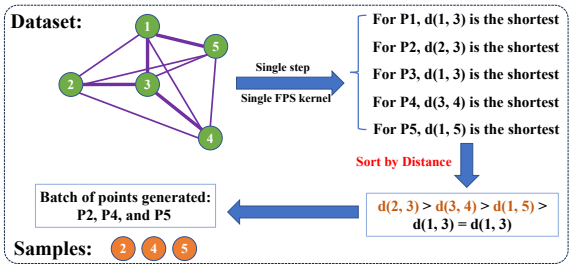


Fig. 2: Architecture overview of PointNet++.



(a) Iterative FPS design



(b) Our proposed parallel design

Fig. 3: An illustrative example of point sampling in point cloud analytics

(where B represents the batch size and N is the number of points in each batch). Each thread computes the distance between point pairs and stores the result in a $B \times N \times (N-1)$ FP32 tensor. Next, we use PyTorch's `min` operator to compute the distance to the nearest neighbor for each point, resulting in a tensor of shape (B, N). This tensor stores the minimum distance values for each point. Finally, we use PyTorch's `TopK` operator to find the K points with the largest distances among all the computed distances as the sampled points. It is worth noting that our approach reduces the number of kernel launches from an order of iterations of FPS to 4.

Summary. Our parallelized approach removes data dependency across iterations, allowing the sampling stage to be fully parallelized, and thus better utilizing the GPU's computational resources.

IV. Kernel-Fused Grouping and Parallel Voting

A. Bottleneck Analysis for the Grouping Stage

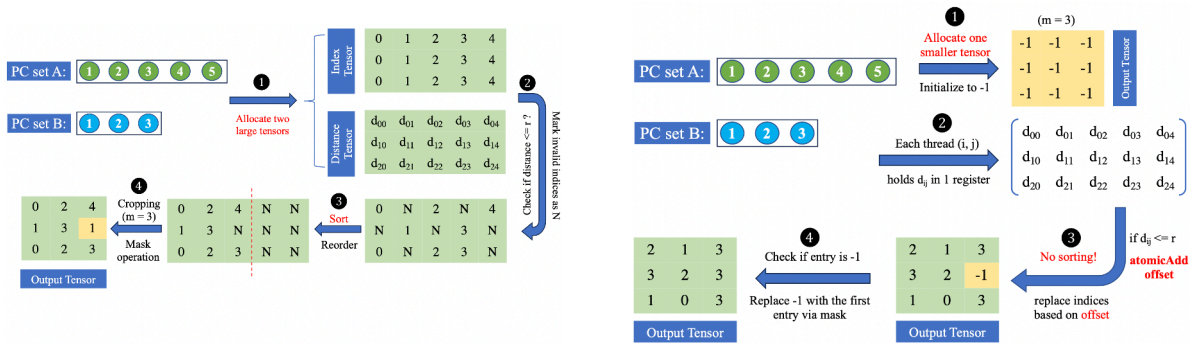
The grouping stage substantially impacts the end-to-end inference latency in PointNet++ across various tasks as illustrated in Figure 1. PointNet++ uses the ball query algorithm during the grouping stage, which identifies points within a specified radius relative to the sampled points.

Baseline Implementation of Grouping. As depicted in Figure 4a, the ball query algorithm takes two sets A and B of points as its input, where set A is the unsampled set of points, and set B is the sampled points. For each point p_b in set B, the algorithm selects every point p_a in set A such that the distance between p_a and p_b is not greater than r, where r is a predefined radius. If the number of points within the radius for a point in set B exceeds a predefined value m, the algorithm only keeps m points for the point. For instance, consider Figure 4a, where set A contains 5 points, set B contains 3 points, and $m = 3$. Initially, the ball query algorithm allocates a large tensor (`Index Tensor`, with the shape of (5, 3), corresponding to the number of points in the PC set A and PC set B) to store the indices of each point in set A (ranging from 0 to 4) (Figure 4a (❶)). Next, it calculates the Euclidean distances between every pair of points across the two sets and stores these values in another tensor (`Distance Tensor`) (Figure 4a (❶)). Then, it identifies and marks invalid indices by determining if the distances exceed the specified radius (Figure 4a (❷)). A sorting operation is performed to move the valid indices to the front (Figure 4a (❸)). The algorithm proceeds to select the first m points, as the objective is to find m neighboring points (Figure 4a (❹)). When fewer than m valid points are available, a mask operation replaces any invalid indices with the index of the first valid point (Figure 4a (❹)).

Memory Bottlenecks. As illustrated in Figure 4a, the baseline demonstrates irregular memory access patterns during the grouping stage. For example, sorting operations require irregular accesses and excessive data movement (Figure 4a (❸)). Furthermore, as shown in Figure 4a, the baseline allocates two large, memory-intensive tensors to temporarily store intermediate computation results during the grouping stage, resulting in redundant memory usage.

B. Kernel-Fused Grouping

Key Insight and Our Approach. Our aim is to replicate the functionality of the ball query algorithm–specifically, identifying neighboring points for each sampled point–while



(a) Implementation of grouping in PointNet++

(b) Our optimized grouping

Fig. 4: Comparison between the grouping implementations of PointNet++ and our approach

mitigating the irregular memory access patterns encountered during the grouping stage in the baseline. To achieve this, we implement custom kernels, effectively bypassing the sorting operations and minimizing redundant memory allocations.

Illustrative Example of Our Approach. Similar to the baseline objective, as depicted in Figure 4b, our approach aims to identify m neighboring points within a distance not exceeding r in PC set A (which contains 5 points) for each point in PC set B (which contains 3 points), where m = 3 and r is a radius. Unlike the baseline, which relies on two large tensors, our method allocates a smaller `Output Tensor` (only with the shape of (3, 3), corresponding to the number of points in the PC set B and m) throughout the entire grouping stage, thereby saving substantial memory–a benefit that becomes even more pronounced in real-world point cloud datasets. Initially, each value in the `Output Tensor` is set to -1, indicating invalid indices (Figure 4b (❶)). We then compute the Euclidean distances between each pair of points in PC set A and PC set B (Figure 4b (❷)). Next, we determine the first m valid indices for each point in the PC set B (where valid indices refer to the indices of points in PC set A within a distance r from PC set B), and replace the invalid indices in the `Output Tensor` with those valid ones (Figure 4b (❸)). Finally, we further inspect the `Output Tensor` to replace any remaining invalid indices with the value of the first valid index (Figure 4b (❹)). In summary, we effectively avoid redundant memory allocations required for storing intermediate values during the grouping stage.

Implementation. Our grouping workflow consists of four primary steps (implemented with custom CUDA kernels): initializing the `Output Tensor`, calculating the distances between all pairs of points across the two point cloud sets, replacing invalid indices in the `Output Tensor` with the first m valid indices for each point in PC set B (recall that m is the required number of neighboring points for each point in set B), and finally replacing any remaining invalid indices (if possible). For the custom CUDA kernels, we allocate $B \times S \times N$ threads (where B represents the batch size, S and N denote the number of points in the two respective point cloud sets) and an `Output Tensor` of size $B \times S \times m$ INT32. The process begins by initializing all values in the `Output Tensor` to -1, representing invalid indices. Subsequently, each thread computes the distance between a pair of points in PC set A and PC set B. Following this, the algorithm determines whether each distance is not greater than the specified radius r; if the condition is met, for each point in PC set B, the current neighboring point's *offset* (ranging from 0 to $m-1$, as we only consider the first m valid indices) is recorded using CUDA's `atomicAdd` method, and the index of the current point is then used to replace the invalid index in the initialized `Output Tensor` based on the offset (Figure 4b (❸)) (a step where we eliminate the need for the sorting operations). Finally, we check the `Output Tensor` to replace any remaining invalid indices (resulting from an insufficient number of valid neighboring points) with the first valid index, ensuring the `Output Tensor` is fully populated.

C. Parallelizing Voting on GPUs

Based on Figure 1, we observe that the voting stage used in semantic segmentation tasks presents another significant bottleneck, accounting for ~25% of the end-to-end inference latency. We identified that the computational tasks at this stage in the baseline exhibit no data dependencies, making them highly suitable for acceleration using the GPU. Thus, we designed a custom CUDA kernel to offload all computational tasks from the CPU to the GPU during this stage.

Implementation. Our voting workflow, implemented with a custom CUDA kernel, primarily involves condition-based voting, aligned with the baseline objective. For the custom CUDA kernel, we allocate $B \times N$ threads (where B represents the batch size, and N denotes the number of points in each batch) and $B \times N \times k$ INT32 memory to store the voting results (where k represents the number of semantic categories in the dataset). First, each thread verifies that the weight value of the semantic category of the current point is neither zero nor infinite. Upon meeting this condition, the corresponding position in the voting pool is incremented by one. Ultimately, we document the voting results, thereby concluding the entire voting stage.

V. Evaluation Methodology

a) Platform: In our experiments, we utilize the NVIDIA Jetson AGX Orin [7], an edge development board equipped with powerful hardware components. This board includes a 2048-core Ampere GPU with a 256KB L2 cache and a 128KB L1 cache per streaming multiprocessor (SM), along with 64 Tensor Cores. Moreover, for our performance measurement and analysis, we employ the Nsight System [8], the Nsight Compute [9] tools, as well as the built-in `tegrastats` [10] tool, on the Jetson development board. These tools provide valuable insights into system performance and enable us to gather detailed metrics for our evaluation.

b) Workloads: To validate the effectiveness of our approach, we ensured the same experimental setup as PointNet++, including the same tasks, datasets, and batch sizes. The S3DIS dataset is utilized for indoor semantic segmentation task. ModelNet40 serves as a 3D image classification dataset for classification task. ShapeNet, on the other hand, is employed for part segmentation task as a 3D shape dataset. The baseline PointNet++ has preprocessed these datasets, which can be directly accessed and downloaded from the official PointNet++ [4] GitHub repository.

TABLE I: Evaluated workloads

Workload	Task	Dataset	#Points/Batch
W1	Semantic Segmentation	S3DIS [11]	4,096
W2	Classification	ModelNet40 [12]	1,024
W3	Part Segmentation	ShapeNet [13]	2,048

c) Configurations:

Baseline: We use PointNet++ as the experimental baseline, and evaluate key metrics such as inference latency, accuracy and IoU across different tasks on the Jetson AGX Orin.

S&G: Beyond the end-to-end inference latency, we focus on the sampling and grouping stages during the inference process of PointNet++, which are critical bottlenecks affecting the entire end-to-end inference. We evaluate the inference latency of these stages in both the baseline and our method, comparing the acceleration in inference latency provided by our approach.

S&G&V: Similar to S&G, but with an additional focus on the voting stage. We found that the voting stage significantly contributes to the end-to-end inference latency in the semantic segmentation task of PointNet++. Therefore, we specifically evaluate the sampling, grouping, and voting stages for semantic segmentation task.

VI. Experimental Results

Inference Latency. We assess the inference latency of our approach compared to PointNet++ in each stage and the end-to-end inference process for different tasks. Figure 6 demonstrates the speedup achieved by our method compared to the baseline. Specifically, we observed speedups of 3.0×, 5.0×, and 2.1× for semantic segmentation, classification, and part segmentation tasks, respectively. On average, our method achieved a speedup of 3.2× across these tasks. Additionally, we profiled the performance of each stage across different tasks. In summary, we have made two observations from Figure 5 and Figure 6: 1) The sampling stage exhibited the highest speedup across all tasks, with a remarkable speedup of nearly 50× over the baseline implementation in the classification task. 2) The classification task benefited the most from our method. This is because our proposed optimizations lead to a significant reduction in inference latency introduced by the sampling and grouping stages.

TABLE II: Inference Latency Comparison

(a) Semantic Segmentation

Time (s)	Sampling	Grouping	Voting	S&G&V	E2E
PointNet++	6,703	1,337	3,593	11,633	15,469
Ours	427	280	422	1,129	5,090

(b) Classification

Time (s)	Sampling	Grouping	S&G	E2E
PointNet++	104.9	11.1	116.0	146.3
Ours	2.1	2.6	4.7	29.1

(c) Part Segmentation

Time (s)	Sampling	Grouping	S&G	E2E
PointNet++	145.5	35.4	180.9	322.6
Ours	4.3	12.1	16.4	156.5

GPU Utilization. To evaluate GPU utilization in our implementation, we compute the achieved occupancy as a key metric, weighted by the duration of each kernel. As illustrated in Figure 7, our method demonstrates a substantial improvement in achieved occupancy across the sampling, grouping, and voting stages compared to the baseline implementation. For instance, in the classification task, the achieved occupancy in the sampling stage increases from 27.1% to 79.3%, and in the grouping stage, it rises from 45.9% to 81.5%. These significant improvements result from our meticulous optimization. Overall, our approach better leverages the computational resources of the Orin GPU, providing robust support for deploying point cloud analysis network models on edge devices.

Computation Reduction. We measured the average number of executed instructions per inference across different tasks and stages. Figure 8 indicates that our approach significantly reduces the number of executed instructions across the evaluated stages. For the semantic segmentation task, the average number of executed instructions in the sampling stage drops by ∼2.4×, from 2.66 billion to 1.13 billion per inference. In the grouping stage, the average count decreases by about 12.5×. The reason behind the improvement is that our approach enables parallel sampling, bypasses expensive sorting operations, and optimizes memory access patterns. Additionally, the significant fusion of kernels further contributes to the reduction in computational load.

Kernel Launch Overhead. We measure the number of

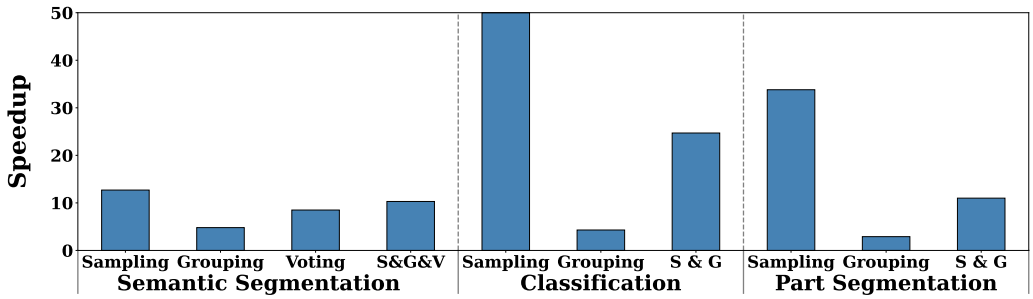


Fig. 5: Speedup over PointNet++ for Different Stages.

TABLE III: Number of Kernel Launches

Task / Stage	Semantic Segmentation		Classification		Part Segmentation	
	PointNet++	**Ours**	PointNet++	**Ours**	PointNet++	**Ours**
Sampling	6,925	4	6,081	4	6,081	4
Grouping	32	2	28	2	30	2
Voting	0	1	0	0	0	0

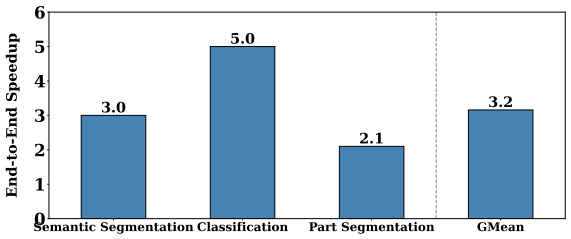


Fig. 6: End-to-end speedup over PointNet++.

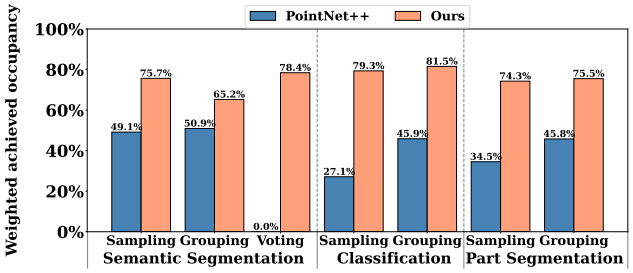


Fig. 7: GPU Utilization

kernel launches in each stage across different tasks for both our method and the baseline. Table III highlights the remarkable decrease in the total number of kernel launches accomplished by our approach, particularly in the sampling and grouping stages. We conducted an experiment to evaluate kernel execution efficiency, demonstrating that excessive kernel launches degrade performance. Specifically, the efficiency is calculated as the ratio of the kernel execution time to the sum of the kernel execution time and the launch overhead:

$$Kernel\ Efficiency = \frac{Kernel\_time}{Kernel\_time + Launch\_time} \quad (1)$$

Figure 9 illustrates that in the baseline implementation, the

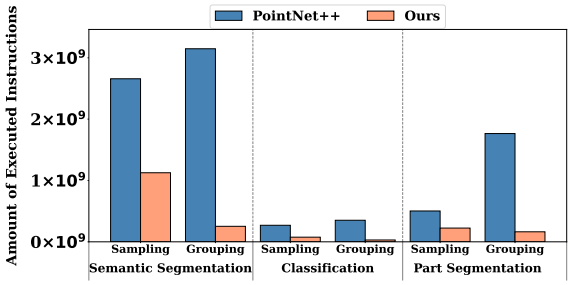


Fig. 8: Comparison of Instructions Executed per Inference

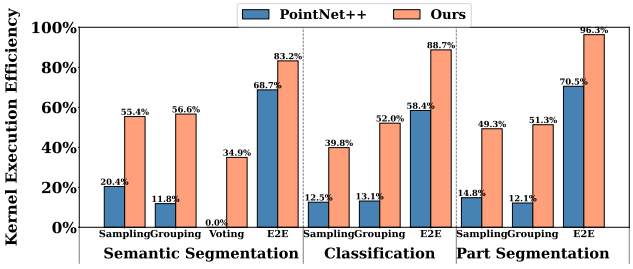


Fig. 9: Kernel Execution Efficiency

kernel launch time accounts for over 30% of the total time in the end-to-end inference, indicating a significant launch overhead. In contrast, our method consistently exhibits superior kernel execution efficiency across all three evaluated tasks.

Data Movement. Table IV and Table V demonstrate the substantial reduction in both the number of memory copies and the amount of data movement achieved by our method compared to PointNet++. Specifically, for the semantic segmentation task, our method achieves an impressive reduction of up to 56× in the number of memory copy operations.

TABLE IV: Number of Memory Copies

Data Copy Counts	PointNet++	**Ours**
Semantic Segmentation	242,682	4,302
Classification	152,324	6,027
Part Segmentation	107,422	16,215

TABLE V: Amount of Data Movement

Tasks	Overall Data Movement (MB)	
	PointNet++	**Ours**
Semantic Segmentation	286,145	97,140
Classification	25,123	1,169
Part Segmentation	112,684	84,842

Table V reveals a noteworthy reduction in the total amount of data movement during end-to-end inference, particularly in the classification task. Notably, the data movement in the classification task decreases by 21× (from 25123 MB to 1169 MB). These results highlight the significant improvements our approach brings in terms of minimizing memory copies and optimizing data movement.

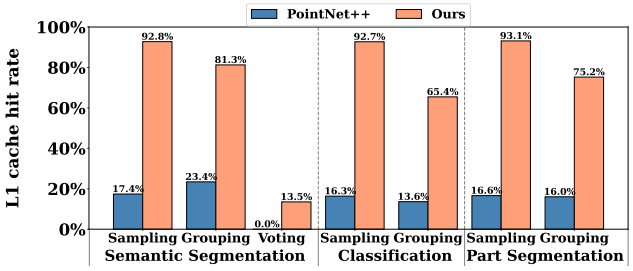


Fig. 10: L1 Cache Hit Rate

Cache Hit Rate. To evaluate memory efficiency, we profile the cache hit rate of our method compared to the baseline throughout various stages. Figure 10 demonstrates that our method significantly improves the L1 cache hit rate in the sampling, grouping, and voting stages compared to the baseline. For instance, in the sampling stage of the semantic segmentation task, the L1 cache hit rate increases from 17.4% to 92.8%. The significant improvement is primarily attributed to the implementation of our proposed kernel design. Our kernel design fuses multiple kernels while minimizing irregular memory accesses, enhancing temporal locality and improving cache efficiency.

Accuracy and IoU Results. Besides performance, we have conducted an evaluation on the accuracy of each task. In terms of accuracy evaluation, we conducted a comprehensive analysis using consistent metrics to compare our method with the baseline across all tasks. As shown in Table VI, the differences in accuracy and IoU between our method and the baseline remain consistently within a narrow margin of 1% for all three tasks. Specifically, for the semantic segmentation task,

TABLE VI: Model Accuracy Comparison

(a) Semantic Segmentation

Accuracy Metric	PointNet++	**Ours**
Overall Accuracy	83.4%	83.0%
Class Average IoU	54.4%	53.8%

(b) Classification

Accuracy Metric	PointNet++	**Ours**
Instance Accuracy	92.7%	92.6%
Class Accuracy	88.5%	88.4%

(c) Part Segmentation

Accuracy Metric	PointNet++	**Ours**
Class Average Accuracy	87.7%	86.9%
Class Average IoU	82.9%	82.1%
Instance Average IoU	85.5%	85.1%

our method exhibits a decrease in accuracy of only 0.4% and a reduction in IoU of 0.6% compared to the baseline. In the classification task, our method demonstrates a minimal drop in accuracy of merely 0.1% when compared to the baseline. Lastly, for the part segmentation task, our method showcases a decrease in accuracy and IoU of just 0.8% in comparison to the baseline.

This experiment demonstrates that our method achieves significant performance improvements compared to the baseline, while maintaining almost the same level of accuracy.

VII. Related Work

This section summarizes the related works on high-performance point cloud analytics on edge devices.

Point Cloud Compression. Processing acquired point cloud data on edge devices is often costly due to the excessive memory footprint. To mitigate this, existing work compresses point cloud data and processes them in compressed formats. Ying et al. [14] propose intra-frame and inter-frame point cloud compression methods specifically designed for edge devices with limited computing and power resources. EdgePC [5] also enhances *sampling*, and *grouping* stages, but requires encoding the point cloud data into Morton codes compressed format. Chen et al. [15] leverage sibling context and surface priors to improve point cloud compression, achieving better compression rates and higher quality reconstructions than existing methods. K-D Bonsai [16] exploits value similarity in the data structure that holds the point cloud to compress the data in memory. Orthogonal to these works, our approach focuses on optimizing parallelism, memory efficiency, and GPU kernel launch overhead for operators used in point cloud analytics.

Model Serving on Edge Devices. Edge devices are often equipped with multiple accelerators, including CPUs, GPUs, and NPUs. Existing research has primarily focused on enhancing throughput by leveraging these heterogeneous accelerators.

For example, Dagli and Belviranli [17] propose a scheduling method to map DNN layers to various accelerators, aiming to reduce latency and improve throughput. PointSplit [18] addresses inference latency for 3D object detection on edge devices through compute kernel redesign and contention-aware scheduling. Unlike these works, our approach identifies the underutilization of edge GPUs as the key bottleneck, and hence we focus on optimizing GPU performance to enhance overall efficiency.

Accelerators. Many domain-specific accelerators have been proposed for efficient and low-latency point cloud analytics. Mesorasi [19] co-designs architecture and algorithm by proposing delayed-aggregation, an approximation technique, to improve performance and energy efficiency while retaining accuracy. PointAcc [20] explores various kernel mappings and caching techniques to improve chip utilization and memory efficiency. Crescent [21] designs an architecture with two new approximation techniques for memory efficiency and proposes a new training approach for point cloud analytics neural networks to better match the architecture, thus retaining accuracy. Chen et al. [22] propose an accelerator that leverages geometric similarity to reduce redundancy. MARS [23] adopts a filtering approach to remove unnecessary off-chip memory accesses and uses a flexible architecture to improve core utilization. In contrast to these accelerators, our work addresses the memory efficiency and core utilization problems by redesigning the compute kernels of the latency-critical subroutines.

VIII. Conclusions

This paper addresses the challenges of deploying PointNet++ on edge devices with limited resources. By identifying and tackling the bottlenecks of excessive kernel launches and irregular memory accesses, we introduce parallel sampling and voting to enhance GPU utilization and fused subroutines in the grouping stage to improve data locality.

Our optimizations significantly reduce inference latency and improve performance across various point cloud workloads on edge devices. Experimental results demonstrate that our approach not only significantly improves performance but also maintains comparable inference accuracy as the backbone PointNet++ across all three point cloud tasks.

Acknowledgement

We sincerely thank the anonymous reviewers for their detailed feedback. This material is based upon work supported by the National Natural Science Foundation of China (#62302416), Guangzhou Municipal Science and Technology Project (#2024A04J4228), the Guangzhou-HKUST(GZ) Joint Funding Program (#2023A03J0138), and a startup grant from The Hong Kong University of Science and Technology (Guangzhou). Corresponding author: Hongyuan Liu.

References

[1] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[2] S. Zhao, H. Zhang, C. S. Mishra, S. Bhuyan, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. Das, "HoloAR: On-the-fly Optimization of 3D Holographic Processing for Augmented Reality," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[3] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic Graph CNN for Learning on Point Clouds," *ACM Trans. Graph. (ToG)*, oct 2019.

[4] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space," in *Advances in Neural Information Processing Systems*, 2017.

[5] Z. Ying, S. Bhuyan, Y. Kang, Y. Zhang, M. T. Kandemir, and C. R. Das, "EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[6] Y. Eldar, M. Lindenbaum, M. Porat, and Y. Zeevi, "The Farthest Point Strategy for Progressive Image Sampling," *IEEE Transactions on Image Processing*, 1997.

[7] NVIDIA Corporation, "NVIDIA Jetson Orin," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2024.

[8] ——, "NVIDIA Nsight Systems," https://docs.nvidia.com/nsight-systems/UserGuide/index.html, 2024.

[9] ——, "NVIDIA Nsight Compute," https://developer.nvidia.com/tools-overview/nsight-compute/get-started, 2024.

[10] ——, "tegrastats Utility," https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html, 2019.

[11] I. Armeni, O. Sener, A. R. Zamir, H. Jiang, I. Brilakis, M. Fischer, and S. Savarese, "3D Semantic Parsing of Large-Scale Indoor Spaces," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*, 2016.

[12] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "Princeton Model Net. "https://modelnet.cs.princeton.edu/"," 2022.

[13] ——, "3D ShapeNets: A deep representation for volumetric shapes," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*, 2015.

[14] Z. Ying, S. Zhao, S. Bhuyan, C. S. Mishra, M. T. Kandemir, and C. R. Das, "Pushing Point Cloud Compression to the Edge," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[15] Z. Chen, Z. Qian, S. Wang, and Q. Chen, "Point Cloud Compression with Sibling Context and Surface Priors," in *Proceedings of the 17th European Conference of Computer Vision (ECCV)*, 2022.

[16] P. H. E. Becker, J.-M. Arnau, and A. González, "K-D Bonsai: ISA-Extensions to Compress K-D Trees for Autonomous Driving Tasks," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[17] I. Dagli and M. E. Belviranli, "Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024.

[18] K. Park, Y. R. Choi, I. Lee, and H.-S. Kim, "PointSplit: Towards On-device 3D Object Detection with Heterogeneous Low-power Accelerators," in *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks (ISPN)*, 2023.

[19] Y. Feng, B. Tian, T. Xu, P. Whatmough, and Y. Zhu, "Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[20] Y. Lin, Z. Zhang, H. Tang, H. Wang, and S. Han, "PointAcc: Efficient Point Cloud Accelerator," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[21] Y. Feng, G. Hammonds, Y. Gan, and Y. Zhu, "Crescent: taming memory irregularities for accelerating deep point cloud analytics," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[22] C. Chen, X. Zou, H. Shao, Y. Li, and K. Li, "Point Cloud Acceleration by Exploiting Geometric Similarity," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[23] X. Yang, T. Fu, G. Dai, S. Zeng, K. Zhong, K. Hong, and Y. Wang, "An efficient accelerator for point-based and voxel-based point cloud neural networks," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2023.