Frugal 3D Point Cloud Model Training via Progressive Near Point Filtering and Fused Aggregation

Donghyun Lee[1], Yejin Lee[2], Jae W. Lee[1], and Hongil Yoon[3]

[1] Seoul National University
[2] Meta
[3] Google
{eudh1206, yejinlee, jaewlee}@snu.ac.kr, hongilyoon@google.com

Abstract. The increasing demand on higher accuracy and the rapid growth of 3D point cloud datasets have led to significantly higher training costs for 3D point cloud models in terms of both computation and memory bandwidth. Despite this, research on reducing this cost is relatively sparse. This paper identifies inefficiencies of unique operations in the 3D point cloud training pipeline: farthest point sampling (FPS) and forward and backward aggregation passes. To address the inefficiencies, we propose novel training optimizations that reduce redundant computation and memory accesses resulting from the operations. Firstly, we introduce Lightweight FPS (L-FPS), which employs progressive near point filtering to eliminate the redundant distance calculations inherent in the original farthest point sampling. Secondly, we introduce the fused aggregation technique, which utilizes kernel fusion to reduce redundant memory accesses during the forward and backward aggregation passes. We apply these techniques to state-of-the-art PointNet-based models and evaluate their performance on NVIDIA RTX 3090 GPU. Our experimental results demonstrate 2.25× training time reduction on average with no accuracy drop.

Keywords: 3D Point Cloud · Efficient Vision · DNN Training

1 Introduction

3D point cloud is a set of points representing scenes or objects with 3D geometry information from sensors (e.g., LiDAR and RGB-D cameras). Deep neural networks (DNNs) [3, 7, 18, 20, 21, 27, 29, 31, 36] have been recently employed for 3D point cloud tasks such as 3D semantic segmentation, object detection, and others. The capacity of DNNs and the size of 3D point cloud datasets have rapidly increased and this trend has led to higher training and inference costs, making real-world deployment challenging. While there have been proposals to optimize inference latency [8, 9, 37, 38], accelerating the training phase has been less explored likely due to the relatively short history of 3D point cloud models and their unique pipeline structure.

Accelerating the training phase of 3D point cloud models is crucial for both research and production. Researchers often repeat training phases exhaustively to experiment with different model architectures and hyperparameters. In practical scenarios, models are frequently retrained to adapt to new patterns and information extracted from an ever-increasing volume of data. These requirements escalate the training cost considerably [6,30]. There are existing techniques to accelerate the training phase in other application domains, such as natural language processing (NLP) and graph neural networks (GNNs) [5, 10, 32]. However, these techniques are not directly compatible with 3D point cloud models. Furthermore, no prior work has thoroughly explored and addressed the unique challenges of 3D point cloud model training.

We have conducted an in-depth empirical analysis on 3D point cloud model training pipelines to identify that the main performance bottleneck results from two *unique* operations differentiating the training of 3D point cloud model from others: (1) *Farthest Point Sampling (FPS)* and (2) *forward and backward aggregation* operations. The FPS operation samples the output point cloud coordinates from the given input point cloud coordinates. It is repeated every epoch, yielding a significant amount of redundant computations. The aggregation operation aggregates related input point features for each output point through max-reduction. This results in a significant amount of redundant reads and writes to GPU global memory, which slows down the training pipeline.

To address these bottlenecks, we introduce two novel optimization techniques specifically designed for the 3D point cloud training pipeline.

- **Lightweight-FPS (L-FPS)**: Preserves randomness and sampling quality while significantly reducing redundant FPS distance calculations. This algorithm leverages two observations: sampling points that are separated by at least a minimum pairwise distance among the sampled points is sufficient to achieve a comparable sampling quality as FPS, and this minimum distance can be predetermined prior to training.
- **Fused aggregation**: Significantly reduces redundant global memory accesses by fusing kernels for forward and backward aggregation passes.

We comprehensively evaluate the effectiveness of these techniques using state-of-the-art PointNet-based models. Evaluations on NVIDIA RTX 3090 GPU demonstrates 2.25× training time reduction on average with no loss in accuracy.

2 Related Work

2.1 Deep Neural Networks for 3D Point Cloud

Sparse Convolution Based Approaches. Sparse convolution based models use sparse convolution on voxelized point cloud data instead of raw point cloud data. They have a network architecture similar to traditional 2D CNNs, which allows many best practices used in 2D CNNs to be applied to them. This has led to the development of many sparse convolution based models [3, 11, 22, 39], demonstrating high performance in various tasks. Nevertheless, one of the limitations of the sparse convolution based models is that they can only operate on the voxelized point cloud data, which inevitably results in the loss of information.

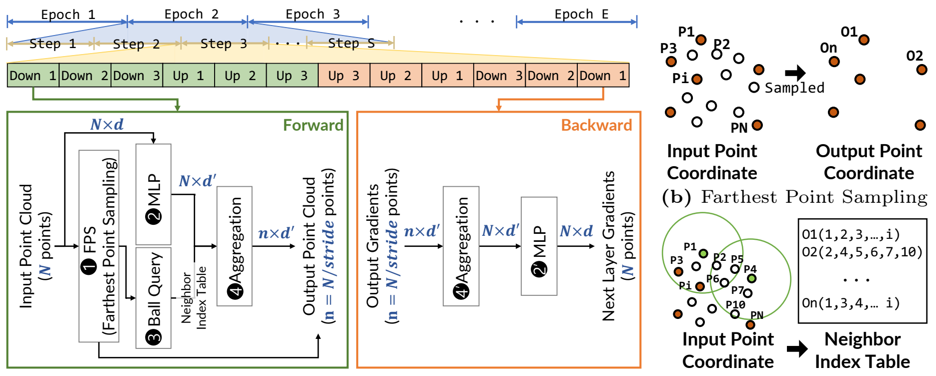
PointNet Based Approaches. PointNet [26] and PointNet++ [27] are the first to apply DNN on raw 3D point cloud without preprocessing. Following the success of PointNet++, numerous other variations [17–19, 25, 28, 29, 33] have emerged, and they have continuously enhanced the performance and computational efficiency of the model. PointNeXt [29] achieves a significant performance improvement through various data augmentation, optimizations, and model scaling. Mesorasi [9] significantly reduces floating-point operations (FLOPs) by adopting the technique called *delayed aggregation*, changing the order of grouping and MLP; Section 3 describes the details of this technique. Very recently, PointMetaBase [18] has introduced explicit positional embedding to further improve model performance. It outperforms state-of-the-art voxel based methods on the S3DIS [2] dataset. They thoroughly explore the optimal PointNet based model architecture in terms of accuracy and FLOPs, providing standard building blocks for future PointNet based models. In this paper, we aim to accelerate the training phase of PointNet based models, which are one of the most commonly used network architectures in 3D point cloud processing.

2.2 Accelerating PointNet Based Models

Efficient Sampling and Neighbor Search. EdgePC [37] accelerates farthest point sampling and neighbor search by structuring point cloud data with Morton codes. QuickFPS [1, 12] proposes a hardware-assisted solution with k-d tree based algorithm to speed up farthest point sampling. While these works reduce the inference latency to some extent, EdgePC suffers from considerable accuracy loss and QuickFPS is limited by its reliance on the specialized hardware support. However, our L-FPS technique accelerates sampling in the training pipeline without sacrificing accuracy and additional hardware changes.

Several proposals have been made to address the inefficiency of the FPS, including random sampling [13], adjustable FPS [16], and grid sampling [14, 35]. However, none of these methods have consistently shown better performance than FPS, mostly resulting in model performance degradation as demonstrated by adjustable FPS [16]. Moreover, these approaches mainly target inference use cases, while we focus on accelerating the training of FPS-based models by exploiting the unique opportunities presented by FPS in the training pipeline.

Reducing Memory Overhead. LPN [15] reduces memory footprint in the training phase by freeing the memory allocated for intermediate values in the forward pass. These values are reconstructed later in the backward pass. While this helps reduce peak memory usage during the training phase, it does not reduce DRAM accesses, thereby failing to improve training throughput. Our fused aggregation technique removes both memory footprint and DRAM accesses to these intermediate values through kernel fusion. Applying the fused aggregation to the LPN not only accelerates training speed but also reduces the peak memory usage.



(a) Overall Training Pipeline of PointNet-based Models **(b)** Farthest Point Sampling **(c)** Ball Query

Fig. 1: General Training Flow of PointNet-based Models

3 Background: Overview of Training Pipeline for PointNet Based Models

This section overviews the training pipeline of PointNet based models. The model architecture is based on PointMetaBase [18], which demonstrates state-of-the-art performance in both accuracy and computational efficiency by adopting delayed aggregation and positional embeddings.

The PointNet based models are mainly composed of downsampling layers. Upsampling layers are used in segmentation models but we mainly focus on describing the downsampling layers as they are the key performance bottlenecks (refer to Section 4.1). Figure 1a demonstrates the forward and backward pass of a single downsampling layer.

The forward pass requires the following steps. ❶ First, the input point coordinates are used to sample $n = N/stride$ output points where N is the number of input points and $stride$ is the downsampling rate. To preserve the shape and information of the input points, Farthest Point Sampling (FPS) [27] is often utilized to sample output points from an unordered set of input points (Figure 1b). ❷ Then, input point features, each represented by a d-dimensional vector, are processed through a Multi-Layer Perceptron (MLP) network. This results in the transformation of the input point feature matrix dimension from $N \times d$ to $N \times d'$. ❸ A ball query is performed on each output point as a centroid to identify the neighbor input points of each output point. The $n_{neigh.}$ neighbor indices of each output point are stored in the neighbor index table (Figure 1c). ❹ Finally, the features of the neighbor points for each output point are grouped according to the neighbor index table and then max-reduced to aggregate the local features of the neighbor points. Several works [17, 18] use positional embeddings in the aggregation stage to encode the relative position information between the neighbors and the centroid. We explain more details about this process in Section 5.2.

The backward pass of the downsampling layer is only performed for the operations that handle the feature vectors, i.e., aggregation and MLP operations.

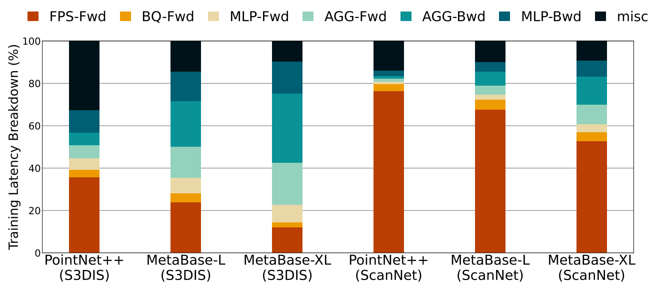


Fig. 2: Training Time Breakdown of PointNet-based 3D point cloud models. FPS, BQ, MLP, and AGG stand for Farthest Point Sampling, Ball Query, Multi Layer Perceptron, and Aggregation, respectively. Upsampling layers are included in misc.

4' For the backward pass of aggregation, gradients of size $n \times d'$ are scattered and then sum-reduced to the matrix of size $N \times d'$. More details about the backward pass of aggregation will be explained in Section 4.2 as well. 2' Finally, using the sum-reduced gradients, the weights for the MLP layers are updated.

4 Analysis of Training Pipeline

4.1 Bottleneck Analysis

Figure 2 shows the latency breakdown of the training phases of 3D point cloud neural network models [18, 27]. Due to the delayed aggregation technique [9], which significantly reduces the overhead of the MLP operations, *FPS* and *aggregation* operations arise as new performance bottlenecks in the training pipeline. These two most time-consuming operations take 47-79% of the training time. First, the *FPS* is a sampling method whose cost quadratically increases as the size of input point cloud grows. Since the cost is heavily dependent on the input point cloud size, the portion of the *FPS* execution time significantly differs across datasets. The high cost of the algorithm results from its sequential nature. Each iteration depends on the results from previous iterations, and there is redundant computation caused by repeated sampling overhead across epochs. Second, the *aggregation* for forward and backward passes also takes an average 22.84% of the overall training time. We observe that this results from redundant accesses to GPU global memory of the aggregation kernel. Section 4.2 presents a detailed analysis of the redundancy in these most time-consuming operations.

4.2 Key Observations

Redundant Computations from FPS. FPS has inherent inefficiency that makes it challenging to parallelize on GPUs. The purpose of the FPS operation is to sample $N/stride$ output points from N input points that are as far apart from each other as possible to preserve the boundary shape of the 3D point cloud. The

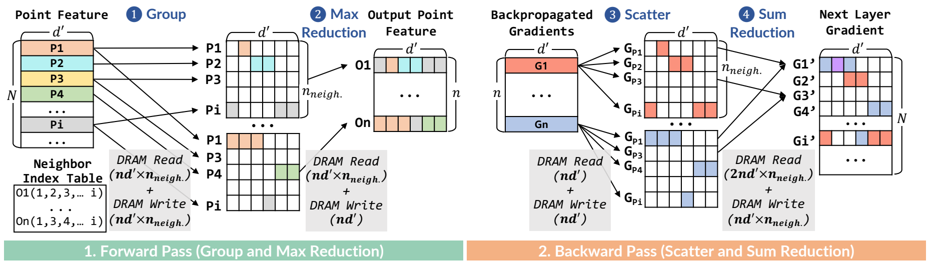

Point Feature
1 Group
2 Max Reduction
Output Point Feature
Neighbor Index Table
O1(1,2,3,… i)
On(1,3,4,… i)
DRAM Read ($nd' \times n_{neigh.}$) + DRAM Write ($nd' \times n_{neigh.}$)
DRAM Read ($nd' \times n_{neigh.}$) + DRAM Write (nd')
Backpropagated Gradients
3 Scatter
4 Sum Reduction
Next Layer Gradient
DRAM Read (nd') + DRAM Write (nd')
DRAM Read ($2nd' \times n_{neigh.}$) + DRAM Write ($nd' \times n_{neigh.}$)
1. Forward Pass (Group and Max Reduction)
2. Backward Pass (Scatter and Sum Reduction)

Fig. 3: Standard Aggregation

process begins with selecting a seed point that is used throughout all layers in the model. Then, the process continues by selecting a point that has the farthest distance from the group of previously selected points. The distance between a point p and a group of points is defined as the minimum of the distances from each point in the group to the point p. This process is iterated until the number of selected points reaches $N/stride$. Only a single point is selected for every iteration, making the entire process sequential. The detailed process is elaborated in Appendix A.1. By examining FPS from a training perspective, we make the following two key observations:

- **Observation 1: Farthest point sampling in the training process incurs a significant number of redundant distance calculations.** The xyz coordinates of the input points do not change throughout the training process, leading to the redundant distance calculations being recomputed for the same input scene every epoch. This motivates us to eliminate inefficiencies arising from these redundant operations in the training pipeline.
- **Observation 2: The key factor in achieving high-quality sampling is to ensure a minimum spacing among the sampled points, and this information can be obtained in advance, prior to training.** As a result of employing FPS, the pairwise distance among all sampled points exceeds a certain minimum *threshold*. Figure 6a demonstrates the distribution of minimum spacing between the points sampled by FPS. If we can identify this threshold value in advance before the sampling phase, simply selecting points whose spacing is larger than the threshold can yield sampling quality nearly identical to that achieved by FPS. Actually, we target the FPS operation in the training process, we can easily calculate the threshold value in advance by simply performing the FPS once before training.

Redundant Memory Accesses from Aggregation. Figure 3 shows the standard GPU implementation [24,29] of the aggregation for forward and backward passes in the training pipeline. The forward pass starts with given N input feature vectors, each with dimension d' and the neighbor index table that stores $n_{neigh.}$ indices of the neighbor points chosen by the ball query operation for each output point (i.e., 3 from Figure 1a). 1 First, $n_{neigh.}$ features are gathered for each output point, generating a matrix of size $n \times n_{neigh.} \times d'$ when the number of output points is n. ❷ Then, column-wise max reduction is applied to grouped $n_{neigh.}$ vectors, reducing the size of the matrix to $n \times d'$. To represent the elements that are selected through max pooling, each element in a column is colored with the same color as the point feature to which it belongs. The grouping operation requires $nd' \times n_{neigh.}$ DRAM reads and writes, while the max reduction operation takes $nd' \times n_{neigh.}$ DRAM reads and nd' DRAM writes.

The backward pass simply updates the corresponding gradients in inverse order. ❸ Given the backpropagated gradients from the previous layer, gradients are scattered to the exactly matching positions where the corresponding output features have originated. This requires a highly sparse $nd' \times n_{neigh.}$ sized scattered gradient matrix, with a sparsity of $1/n_{neigh.}$. ❹ Then, they are reduced by performing atomic, element-wise summation to the position where the corresponding input features have originated. The elements in the next layer's gradient are colored with the same color as the summed gradients. (The violet-colored element of $G1^{'}$ indicates that the element is obtained by summing the second elements of $G1$ and Gn). The scatter operation requires nd' DRAM reads and writes, while the sum reduction takes $2nd' \times n_{neigh.}$ DRAM reads and $nd' \times n_{neigh.}$ DRAM writes. Through the analysis of the aggregation passes, we make the following key observations:

- **Observation 3: There are redundant memory accesses to intermediate values in both forward and backward passes.** The forward and backward passes lead to a significant amount of redundant memory reads and writes to the $nd' \times n_{neigh.}$ sized intermediate values in the forward pass and scattered gradients in the backward pass.
- **Observation 4: Ineffectual computations are performed in the backward pass.** When performing sum reduction in the backward pass, some summations are performed on zeros due to the high sparsity of the scattered gradients matrix. This does not affect the output. For example, we observe that two G_{P3} gradient vectors are gathered and summed, even though this is an ineffectual computation with zero vectors.

5 Our Proposal

We propose two optimization techniques based on the observations from the previous section. First, we introduce L-FPS, a *Lightweight FPS via progressive near point filtering*, which is a technique that produces multiple high-quality sampling results without redundant distance calculations. Second, we propose *fused aggregation* that reduces the redundant memory accesses and computations by fusing memory operations during the aggregation.

5.1 Lightweight FPS via Progressive Near Point Filtering

There are two key requirements for sampling within the training pipeline. First, the sampling approach should provide good sampling quality. Second, it also

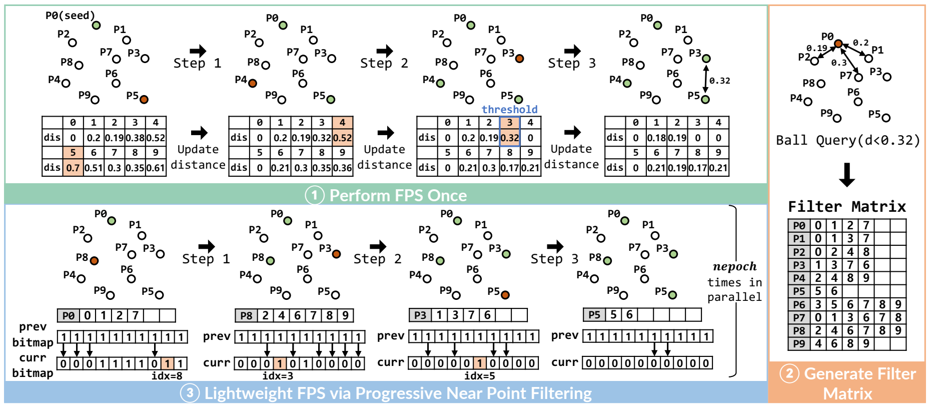


Fig. 4: Lightweight FPS via progressive near point filtering

should provide enough randomness among samples across epochs. The original FPS meets the requirements by selecting the farthest samples and utilizing different seed points every epoch. Appendix A.3 elaborates on the importance of randomness in FPS.

We propose Lightweight-FPS (L-FPS), a sampling technique that maintains both randomness and sampling quality while eliminating redundant distance calculations across epochs. Figure 4 illustrates the process using an example of sampling four points out of ten input points. Note that all steps are performed offline before training. ❶ First, we perform FPS once and find the minimum spacing between the farthest point sampled points, which is the distance between the last selected point and the second-to-last selected point in this example. This value, referred to as the threshold (0.32 in the figure), serves as a key parameter. ❷ Then, we perform a ball query with each input point as a centroid to identify points within the specified threshold proximity. The indices of the identified points are then saved to the filter matrix. In the illustrated example, indices 0, 1, 2, and 7 are stored in the 0th entry, meaning that the distances between point P0 and points P0, P1, P2, and P7 are all smaller than the threshold. ❸ Finally, we perform lightweight FPS using the filter matrix. We use a bitmap to indicate the points that can be sampled. At first, all bitmap entries are initialized to 1, indicating that all points can be sampled. In each step, the bitmap is updated by setting the corresponding entries to zeros for the point indices from the filter matrix. This operation filters out points that are sufficiently close ($< threshold$) to the previously sampled points. After filtering, we randomly select a point index that is still set to 1 from the bitmap. By repeating this step, we can get the sampling results. This process is performed for *nepoch* (number of epochs) times in parallel on the GPU, and *nepoch* versions of sampling results are saved to disk. Different versions of sampling results are then loaded for each epoch during training, ensuring the randomness of samples across the epochs.

The biggest advantage of L-FPS is that it does not require any redundant distance calculations when generating *nepoch* sampling results at the filtering phase. We perform distance calculations only once when performing FPS and generating filter matrix. However, the cost is negligible compared to performing FPS for *nepoch* times in the original strategy. Actually, L-FPS not only reduces the amount of computation, but also minimizes memory accesses and memory footprint through the use of lightweight filter matrix. This further improves the performance of training. A naive approach to avoid redundant computations is to store all the distances between points in a lookup table. This leads to a tremendous amount of memory access and footprint overhead of $O(N^2)$, where N is the number of points. With the proposed near point filtering approach, all the information we need for sampling can be concisely presented in the filter matrix, which has $O(kN)$ complexity, where k is the column dimension of the filter matrix. We empirically observe that setting k to 16 is sufficient to cover the number of points to be filtered.

The key distinction between L-FPS and FPS lies in allowing the ability to sample any points that exceed a predetermined threshold for spacing, rather than selecting a single point farthest from the previously selected point set. The experimental results corroborate this claim. In Section 6.2, we compare the minimum spacing distribution and visualization outcomes between L-FPS and FPS. Additionally, we demonstrate through experiments that L-FPS has no negative impact on accuracy.

Compatibility with Data Augmentation Techniques. In practice, PointNet based models adopt various data augmentation techniques [18, 29] to achieve higher model performance. Popular transformations including scaling, rotation, jittering, and shifting are all compatible with L-FPS with minor impacts on sampling quality because they do not significantly change the relative distances between points. Random cropping, which adjusts the size of an input scene by randomly cropping a particular section, is also compatible with L-FPS with minor code changes in the dataloader. (Appendix C.2 and C.3).

Time and Space Overhead. Saving sampling results on disk for all scenes in the training dataset results in time and space overhead. However, L-FPS incurs a one-time cost, and thus the overall cost is negligible when compared to the entire training time. The latency accounts for at most 3% of the baseline training time. The space overhead is also relatively marginal, which is around 1 to 10 gigabytes (refer to Appendix A.2 for the details). Furthermore, it does not incur any additional GPU memory usage as we don't store the sampled indices for all training data samples in memory during training. Only the necessary indices are loaded and released in each iteration.

5.2 Fused Aggregation

To eliminate the memory and computational inefficiencies in aggregation, we propose fusing the two operations: grouping and max-reduction. Figure 5 illustrates fused aggregation for the forward and backward passes. The fused forward

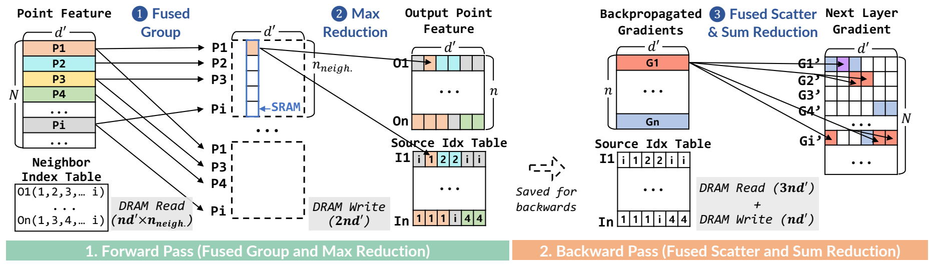

Point Feature
1 Fused Group
2 Max Reduction
Output Point Feature
Neighbor Index Table
DRAM Read ($nd' \times n_{neigh.}$)
SRAM
DRAM Write ($2nd'$)
Source Idx Table
Saved for backwards
Backpropagated Gradients
3 Fused Scatter & Sum Reduction
Next Layer Gradient
DRAM Read ($3nd'$) + DRAM Write (nd')
1. Forward Pass (Fused Group and Max Reduction)
2. Backward Pass (Fused Scatter and Sum Reduction)

Fig. 5: Overview of Fused Aggregation

pass works as follows: ❶ target columns that are to be max-reduced are read from DRAM and temporarily written to SRAM (i.e., shared memory in GPU). ❷ Max-reduction is then performed on the fly in SRAM and the maximum element is written to DRAM. Note that this does not incur any additional SRAM usage compared to standard aggregation. The large intermediate value matrix ($n \times n_{neigh.} \times d'$) does not need to be present in SRAM at the same time, which is the same case in standard aggregation. The total DRAM access is reduced to $nd' \times n_{neigh.}$ reads and nd' writes by eliminating redundant DRAM accesses. However, once the two kernels are fused in the forward pass, we cannot perform backpropagation without additional information about where to scatter the gradients. To address this, we save extra information that tracks corresponding sources of input point features for the selected maximum elements. This source index table is later used in the backpropagation. This incurs an additional overhead of nd' writes to DRAM, which makes the total DRAM write count $2nd'$.

Given the source index from the forward pass, the backward pass can be easily fused. ❸ All we have to do is to scatter the gradients according to the source index table. For example, given the source indexes of the gradient vector $G1$, i.e., $(i, 1, 2, 2, i, i)$, the corresponding gradient values from the first, fifth, and sixth columns are scattered to the i^{th} gradient, the corresponding value from the second column is scattered to the 1^{st} gradient, and the values from the third and fourth columns are scattered to the 2^{nd} gradient. Since the memory accesses to the intermediate gradient matrix are eliminated, the fused version of backpropagation only incurs $3nd'$ DRAM reads and nd' DRAM writes.

To sum up, the total amount of DRAM accesses is reduced from $(3nd' \times n_{neigh.} + nd')$ to $(nd' \times n_{neigh.} + 2nd')$ in the forward pass and from $(3nd' \times n_{neigh.} + 2nd')$ to $4nd'$ in the backward pass. Considering that $n_{neigh.}$ is usually 32 in most conventions [18, 29], fused aggregation requires about 5.13× fewer DRAM accesses than the baseline.

Support for Explicit Positional Embedding. Several recent works [17, 18] use explicit positional embedding to encode the relative position information between the centroid and its neighbors. Since they are added to each of the grouped feature vectors before max pooling, we have made some minor changes to our fused kernel. Detailed descriptions of the algorithm can be found in Appendix B.1.

Dataset	Model	N_{point}	N_{layer}	Epochs	Batch size	Accuracy (Stdev.)	
						Baseline	L-FPS
S3DIS [2]	PN++	24000	4	100	8	63.19 (0.54)	63.39 (0.34)
	MB-L	24000	15	100	8	69.82 (0.40)	69.76 (0.40)
	MB-XL	24000	20	100	8	70.67 (0.37)	70.74 (0.43)
ScanNet [4]	PN++	64000	4	100	2	59.42 (0.26)	59.54 (0.57)
	MB-L	64000	15	100	2	70.52 (0.27)	70.54 (0.31)
	MB-XL	64000	20	100	2	71.78 (0.28)	71.74 (0.44)

Table 1: Dataset, Model Training Parameters, and Model Performance. PN++, MB stands for PointNet++, PointMetaBase.

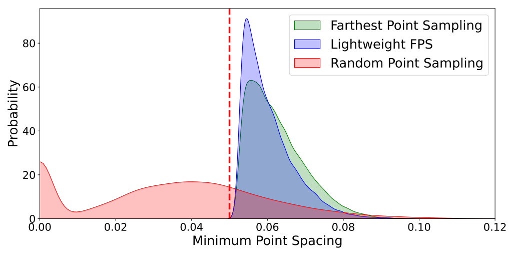
Time and Space overhead. As explained above, saving the source index table incurs an additional time overhead for DRAM writes (nd') and the same amount of space overhead (nd') for DRAM memory capacity. However, this is almost negligible compared to the saved DRAM access and footprints ($2nd' \times n_{neigh.}$), considering that the common value of $n_{neigh.}$ is 32 [18, 29].

6 Experiments

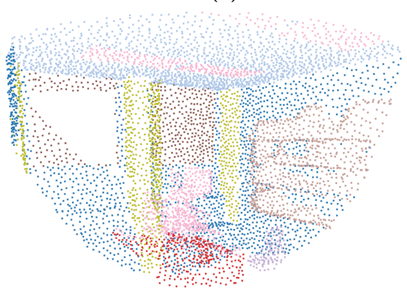
6.1 Methodology

Dataset, Models, and Metrics. We evaluate our proposals based on six PointNet-based models [18, 27] for various 3D point cloud tasks [2, 4]. Table 1 shows target models, datasets, and training parameters. Note that N_{point} denotes the size of point cloud scene and N_{layer} refers to the total building blocks in PointNet++ and PointMetaBase. We use two versions of PointMetaBase (L and XL) [18] that recently achieved state-of-the-art performance in S3DIS datasets. For PointNet++ [27], we have augmented the model architecture with delayed aggregation [9] and positional embedding [18] to make the baseline stronger by boosting the performance in both accuracy and efficiency as described in Appendix B.2. Furthermore, we use the same training techniques and evaluation methodology used in PointMetaBase [18] and PointNeXt [29]. Appendix C.1 describes more information about data augmentations. We use mIoU (mean Intersection over Union) as an accuracy metric in 3D semantic segmentation. We measure the training throughput by running three epochs, multiplying the average throughput by the number of epochs and adding preprocessing time required for generating sampling results using our L-FPS strategy. We evaluate the accuracy impact with validation set because the test labels are not publicly available.

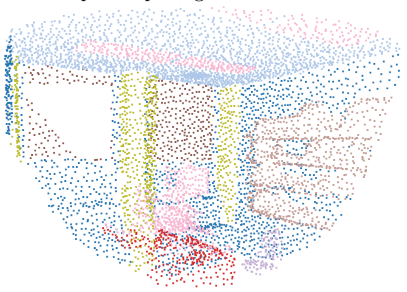
Implementations. We have implemented our proposals on OpenPoints library [29], which is a highly optimized framework that supports various PointNet-based models. We have applied a minor change to the dataloader and implemented a new custom GPU kernel in CUDA to support L-FPS and fused aggregation. L-FPS can be easily applied by adding a few lines of code to the dataloader



(a) Distribution of minimum point spacing



(b) Farthest Point Sampling



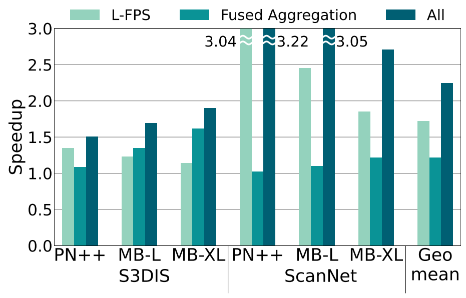
(c) Lightweight FPS(L-FPS)

Fig. 6: Comparison of sampling quality for S3DIS dataset

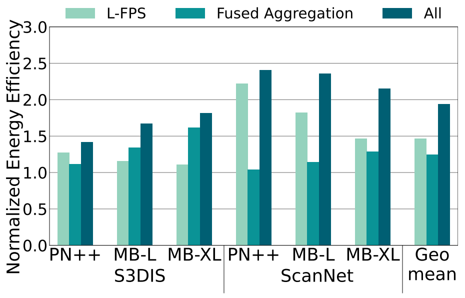
and our fused GPU kernel can be used just like other operations registered in the OpenPoints library. We only apply L-FPS to the first layer of each model since they are the major bottleneck in most cases. This is because as the number of points decreases due to downsampling, the execution time of FPS decreases quadratically. We use one NVIDIA RTX 3090 GPU [23] and 2×Intel(R) Xeon(R) Gold 6338 CPU with 32 core and 512GiB of DRAM. For software setup, we use PyTorch 1.10.1 with CUDA 11.2. All training is performed in FP32 data format.

6.2 Sampling Quality and Model Accuracy

This section analyzes the impact of L-FPS on sampling quality and end-to-end model accuracy. Note that fused aggregation does not introduce any algorithmic change to the training pipeline and thus does not impact accuracy. Figure 6b and 6c present the sampling quality of FPS and L-FPS, respectively, through visualization of the point cloud scene from S3DIS dataset sub-sampled with each strategy. We report visualization results for other datasets in Appendix D.2. The results indicate that L-FPS maintains sampling quality. Figure 6a demonstrates the distribution of minimum point spacing between the sampled points. L-FPS maintains near-identical point spacing to FPS, consistently exceeding the threshold. This is especially evident compared to random sampling. In contrast to FPS, which selects the farthest points from each other, L-FPS samples any point beyond a predefined threshold, leading to a point spacing distribution slightly skewed towards lower values. However, the accuracy analysis in Table 1 demonstrates that the L-FPS has a negligible impact on accuracy, as indicated



(a) Normalized end-to-end training throughput.



(b) Normalized GPU energy efficiency.

Fig. 7: Performance Improvements. PN++ and MB stands for PointNet++ and PointMetaBase.

by a maximum mIoU loss of 0.06% and a potential gain of 0.2%. We also demonstrate the superiority of L-FPS in Appendix D.1 through comparisons with other sampling algorithms. Additionally, the applicability of L-FPS to other models and datasets is described in Appendix D.5 and D.6.

6.3 Training Throughput Improvement with Ablation Study

Figure 7a shows normalized end-to-end training throughput improvement based on various models and datasets. With the L-FPS standalone, our proposal can improve the geomean end-to-end throughput by 1.72×. With the fused aggregation standalone, we can achieve 1.22× geomean end-to-end speedup. The portion of FPS is quadratically proportional to the number of points, while the portion of aggregation is proportional to the number of downsampling layers. Based on this fact, the model configurations shown in Table 1 explain the speedup results. The speedup of fused aggregation tends to increase as the model size scales, whereas the speedup of L-FPS shows the opposite trend. This trend can be well observed in S3DIS dataset. While the speedup of L-FPS dominates the overall speedup in lightweight PointNet++ model, the trend is reversed as the model scales to PointMetaBase-XL.

On the other hand, the speedup of L-FPS far exceeds that of fused aggregation as the dataset size (number of input points) increases. This trend can be observed in the case of ScanNet dataset, which has the most number of points per scene. Our proposal achieves 2.25× geomean end-to-end speedup when both techniques are applied. L-FPS tends to contribute more to the overall speedup than fused aggregation because the portion of the FPS operations is usually larger than that of the aggregation as shown in Figure 2. Actually, the portions vary depending on model configurations. We can expect that these orthogonal approaches can have an additive effect. The experimental results substantiate that both are essential to achieve speedups across various models and datasets.

The detailed training time numbers are presented in Appendix D.3. The speedup specifically for the aggregation part is reported in Appendix D.4.

6.4 Energy Efficiency Improvement

We monitor the power consumption of GPU by running `nvidia-smi` command at 1ms intervals, and multiply average power consumption and training time to obtain energy consumption. Figure 7b demonstrates the normalized GPU energy efficiency. Our proposal achieves geomean 1.94× energy efficiency improvements when both techniques are applied. Smaller energy efficiency improvements than the speedup implies that the power consumption increases when our techniques are applied. This is because during the baseline training, GPU becomes almost idle while performing inefficient FPS, resulting in low power consumption.

6.5 Sensitivity Study

Batch Size. We evaluated three training batch sizes: default batch size for main experiments, halved, and doubled batch size. Figure 8 shows that our proposal generally performs better in small batch sizes because FPS becomes more computationally inefficient when the batch size decreases.

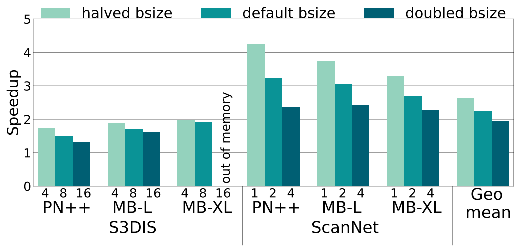


Fig. 8: Sensitivity to batch size.

Number of Points. In Section 6.2, we employed large-scale datasets with tens of thousands of points per scene. To assess the sensitivity to the number of points, we tested our proposal on PointNet++ model with ModelNet40 [34] dataset, which uses only 1024 points per scene. With fused aggregation, whose speedup is not sensitive to the number of points, we achieved a 1.22× speedup, whereas L-FPS did not show any speedup and even resulted in a 1.16× slowdown. The latency of FPS is so minimal that the preprocessing time of L-FPS and the overhead of loading sampling results from the dataloader take more than the actual time required to perform FPS. Therefore, we recommend employing only fused aggregation when dealing with an extremely small number of input points.

7 Conclusion

PointNet-based models are widely used for 3D point cloud tasks. However, the training costs have escalated rapidly due to the increase in model capacity and datasets. This work presents a comprehensive analysis of the training pipeline for state-of-the-art PointNet-based models, identifying unique optimization opportunities. To capitalize on these opportunities, we propose two techniques: *Lightweight FPS via progressive near point filtering* and *fused aggregation*. Our proposal significantly reduces redundant computations and memory accesses. The experimental results demonstrate that our techniques can substantially reduce the training time without compromising accuracy. We believe these findings are valuable for researchers and practitioners who seek to efficiently train and deploy their 3D point cloud applications in real-world scenarios.

Acknowledgements

This work was supported by a research grant from Samsung Advanced Institute of Technology (SAIT) and the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (RS-2024-00340008). Additionally, this work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program (IITP-2023-RS-2023-00256081) and an IITP grant (No. 2021-0-02068, Artificial Intelligence Innovation Hub), both funded by the Korea Government (MSIT). The source code is available at `https://github.com/SNU-ARC/Frugal_PN_Training.git`. This work was conducted while Yejin Lee was with Seoul National University. Jae W. Lee and Hongil Yoon are the corresponding authors.

References

1. Quickfps. `http://github.com/hanm2019/bucket-based_farthest-point-sampling_GPU`
2. Armeni, I., Sener, O., Zamir, A.R., Jiang, H., Brilakis, I., Fischer, M., Savarese, S.: 3d semantic parsing of large-scale indoor spaces. In: CVPR (2016)
3. Choy, C., Gwak, J., Savarese, S.: 4d spatio-temporal convnets: Minkowski convolutional neural networks. In: CVPR (2019)
4. Dai, A., Chang, A.X., Savva, M., Halber, M., Funkhouser, T., Nießner, M.: Scannet: Richly-annotated 3d reconstructions of indoor scenes. In: CVPR (2017)
5. Dao, T., Fu, D.Y., Ermon, S., Rudra, A., Ré, C.: FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In: NeurIPS (2022)
6. Evci, U., Gale, T., Menick, J., Castro, P.S., Elsen, E.: Rigging the lottery: Making all tickets winners. In: Proceedings of the 37th International Conference on Machine Learning (ICML) (2020)
7. Fan, L., Pang, Z., Zhang, T., Wang, Y.X., Zhao, H., Wang, F., Wang, N., Zhang, Z.: Embracing Single Stride 3D Object Detector with Sparse Transformer. In: CVPR (2022)
8. Feng, Y., Hammonds, G., Gan, Y., Zhu, Y.: Crescent: Taming memory irregularities for accelerating deep point cloud analytics. In: Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA) (2022)
9. Feng, Y., Tian, B., Xu, T., Whatmough, P., Zhu, Y.: Mesorasi: Architecture support for point cloud analytics via delayed-aggregation. In: Proceedings of the 53th International Symposium on Microarchitecture (MICRO) (2020)
10. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
11. Graham, B., Engelcke, M., van der Maaten, L.: 3d semantic segmentation with submanifold sparse convolutional networks. In: CVPR (2018)
12. Han, M., et al.: Quickfps: Architecture and algorithm co-design for farthest point sampling in large-scale point clouds. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023)
13. Hu, Q., Yang, B., Xie, L., Rosa, S., Guo, Y., Wang, Z., Trigoni, N., Markham, A.: Randla-net: Efficient semantic segmentation of large-scale point clouds. In: CVPR (June 2020)

14. Junyuan Ouyang, Xiao Liu, H.C.: Hierarchical adaptive voxel-guided sampling for real-time applications in large-scale point clouds. arXiv preprint arXiv:2305.14306 (2023)
15. Le, E.T., Kokkinos, I., Mitra, N.J.: Going deeper with lean point networks. In: CVPR (2020)
16. Li, J., Zhou, J., Xiong, Y., Chen, X., Chakrabarti, C.: An adjustable farthest point sampling method for approximately-sorted point cloud data. In: 2022 IEEE Workshop on Signal Processing Systems (SiPS) (2022)
17. Li, Y., Bu, R., Sun, M., Wu, W., Di, X., Chen, B.: Pointcnn: Convolution on x-transformed points. In: NeurIPS (2018)
18. Lin, H., Zheng, X., Li, L., Chao, F., Wang, S., Wang, Y., Tian, Y., Ji, R.: Meta architecture for point cloud analysis. In: CVPR (2023)
19. Liu, Y., Fan, B., Meng, G., Lu, J., Xiang, S., Pan, C.: Densepoint: Learning densely contextual representation for efficient point cloud processing. In: ICCV (2019)
20. Liu, Z., Tang, H., Lin, Y., Han, S.: Point-voxel cnn for efficient 3d deep learning. In: NeurIPS (2019)
21. Liu, Z., Yang, X., Tang, H., Yang, S., Han, S.: Flatformer: Flattened window attention for efficient point cloud transformer. In: CVPR (2023)
22. Nekrasov, A., Schult, J., Litany, O., Leibe, B., Engelmann, F.: Mix3D: Out-of-Context Data Augmentation for 3D Scenes. In: International Conference on 3D Vision (3DV) (2021)
23. Nvidia geforce RTX 3090. `https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090/` (2020)
24. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: NeurIPS (2019)
25. Qi, C.R., Liu, W., Wu, C., Su, H., Guibas, L.J.: Frustum pointnets for 3d object detection from rgb-d data. In: CVPR (2018)
26. Qi, C.R., Su, H., Mo, K., Guibas, L.J.: Pointnet: Deep learning on point sets for 3d classification and segmentation. arXiv preprint arXiv:1612.00593 (2016)
27. Qi, C.R., Yi, L., Su, H., Guibas, L.J.: Pointnet++: Deep hierarchical feature learning on point sets in a metric space. arXiv preprint arXiv:1706.02413 (2017)
28. Qian, G., Hammoud, H., Li, G., Thabet, A., Ghanem, B.: Assanet: An anisotropical separable set abstraction for efficient point cloud representation learning. In: NeurIPS (2021)
29. Qian, G., Li, Y., Peng, H., Mai, J., Hammoud, H., Elhoseiny, M., Ghanem, B.: Pointnext: Revisiting pointnet++ with improved training and scaling strategies. In: NeurIPS (2022)
30. Rebuffi, S.A., Kolesnikov, A., Sperl, G., Lampert, C.H.: icarl: Incremental classifier and representation learning. In: CVPR (2017)
31. Tang, H., Liu, Z., Zhao, S., Lin, Y., Lin, J., Wang, H., Han, S.: Searching efficient 3d architectures with sparse point-voxel convolution. In: ECCV (2020)
32. Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)
33. Wang, Y., Sun, Y., Liu, Z., Sarma, S.E., Bronstein, M.M., Solomon, J.M.: Dynamic graph cnn for learning on point clouds. ACM Transactions on Graphics (TOG) (2019)

34. Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., Xiao, J.: 3d shapenets: A deep representation for volumetric shapes. In: CVPR (2015)
35. Xu, Q., Sun, X., Wu, C.Y., Wang, P., Neumann, U.: Grid-gcn for fast and scalable point cloud learning (2020)
36. Yang, Y.Q., Guo, Y.X., Xiong, J.Y., Liu, Y., Pan, H., Wang, P.S., Tong, X., Guo, B.: Swin3d: A pretrained transformer backbone for 3d indoor scene understanding. arXiv preprint arXiv:2304.06906 (2023)
37. Ying, Z., Bhuyan, S., Kang, Y., Zhang, Y., Kandemir, M.T., Das, C.R.: Edgepc: Efficient deep learning analytics for point clouds on edge devices. In: Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA) (2023)
38. Zhang, J.F., Zhang, Z.: Point-x: A spatial-locality-aware architecture for energy-efficient graph-based point-cloud deep learning. In: Proceedings of the 54th International Symposium on Microarchitecture (MICRO) (2021)
39. Zhu, X., Zhou, H., Wang, T., Hong, F., Ma, Y., Li, W., Li, H., Lin, D.: Cylindrical and asymmetrical 3d convolution networks for lidar segmentation. arXiv preprint arXiv:2011.10033 (2020)