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

A Supplementary Materials for L-FPS

A.1 Farthest Point Sampling Algorithm

Algorithm 1 FPS Algorithm

```
Input P: input point set, n: number of output points, seed: index of the seed point
Output fps_idx: list of sampled points indices
/* Initialization */
fps_idx ← [seed]
min_dist[$i$] ← $inf$ where $i = 0, ..., N - 1$
/* Sampling */
for $i \leftarrow 1$ to $n - 1$ do    // Cannot be parallelized
    for $j \leftarrow 0$ to $N - 1$ do    // Can be parallelized
        D ← distance($P$[fps_idx[$i - 1$]], $P$[$j$])
        if min_dists[$j$] > D then
            min_dists[$j$] ← D
    fps_idx.append(argmax(min_dists))
```

We describe the Farthest Point Sampling (FPS) algorithm, which samples $n = N/stride$ ($stride$ = downsampling rate) output points from given N input points in a way that maximizes the distances of points in order to preserve the boundary of the 3D point cloud. We explain this process in detail with Algorithm 1.

Given the input point set P of N input points and the index of the seed point *seed*, the final goal of this algorithm is to generate the indices of n sampled points (i.e., **fps_idx**). The initial length of fps_idx is 1 since FPS starts sampling with the seed point (Line 2). As the process continues, it will append $(n - 1)$ more sampled points to the list. Also, we define min_dist of length N initialized with *inf* values (Line 3). The usage of min_dist[i] is to keep track of the minimum distance between an input point $P[i]$ and a group of sampled output points so far.

The outer loop iterates $(n - 1)$ times and each iteration produces a sampled output point (Line 5-10). To sample an output point, the inner loop iterates N

input points and determines which point to sample as an output point (Line 6-9). An inner loop computes the distance between the current input point (i.e., $P[j]$) and the most recently sampled output points (i.e., $P[fps\_idx[i-1]]$) (Line 7). If this distance is smaller than the min_dist[j], it updates min_dist[j] with this distance (Line 8-9). In this way, min_dist[j] stores the minimum distance among the distances between input point j and i sampled points so far. Once the inner loop finishes iterating N input points, we retrieve the index from min_dist that has the maximum value and append this index to fps_idx (Line 10). This process maximizes the distance among the sampled points. As the outer loop finishes iteration, fps_idx finally gets all the output points.

A.2 Time and Space Overhead of L-FPS

We analyze time and space overhead of L-FPS in Table 1. For the time overhead, we report the extra time cost caused by L-FPS and its relative ratio compared to the total baseline training time. The experimental results show that the time overhead of our proposed approach is negligible, accounting for no more than 3.03% of the total training time. Moreover, it is noteworthy that L-FPS results obtained using our approach can be reused across different models if the stride of the first layer are identical. All three models employed in our experiments use the same stride value of 4, enabling them to share the FPS results for the same training dataset. For the space overhead, we report the file size of L-FPS results. The disk space overhead is also relatively small, accounting for 1.3 and 12.4 gigabytes for S3DIS and ScanNet dataset, respectively.

To address space overhead issue that can arise with large datasets, we propose two solutions, each with its corresponding trade-offs:

- *On-the-fly filtering (space VS. time):* Performs L-FPS filtering stage every epoch instead of performing them in a batch before training. This reduces storage overhead but increases the training time since batching benefits are lost. This approach still yields a substantial speedup over FPS.
- *Sample reuse (accuracy VS. space):* Stores a certain amount of sampling results ($nepoch/N$) and reuse each sample N times during training. This reduces storage overhead by a factor of N, but may sacrifice the accuracy due to the decreased sampling randomness.

Model (Dataset)	Time Overhead (Time, Portion)	Space Overhead (File size)
PN++ (S3DIS)	357s (3.03%)	1.3GiB
MB-L (S3DIS)	357s (2.07%)	1.3GiB
MB-XL (S3DIS)	357s (1.09%)	1.3GiB
PN++ (ScanNet)	2732s (1.85%)	12.4GiB
MB-L (ScanNet)	2732s (1.63%)	12.4GiB
MB-XL (ScanNet)	2732s (1.26%)	12.4GiB

Table 1: Time and Space Overhead of L-FPS. PN++, MB stands for PointNet++, PointMetaBase.

A.3 Necessity of randomness in Sampling

Training with fixed subsampled points can lead to the model performance degradation and overfitting. We quantify this by using the metric $d_{overfit}$*(train mIoU - val mIoU)*, which indicates the degree of overfitting. Table 2 shows that using fixed subsampled points in training results in overfitting and performance degradation relative to the baseline, while L-FPS does not.

	MB-L (ScanNet)			MB-XL (ScanNet)		
	Fixed	Base	L-FPS	Fixed	Base	L-FPS
mIoU (%)	70.06	70.52	70.54	71.56	71.78	71.74
$d_{overfit}$	20.99	19.84	19.55	22.14	21.27	21.10

Table 2: Model performance and degree of overfitting ($d_{overfit}$) measured on PointMetaBase-L and XL model for ScanNet dataset.

B Supplementary Materials for Fused Aggregation

B.1 Algorithmic Extension to Support Explicit Positional Embedding

Explicit positional embedding [3] is a technique used to encode the relative position information between the output point and its neighbors, and positional embeddings are obtained with a small sized MLP that takes point coordinates as input. Adding positional embeddings to the neighbor features allows output points to aggregate richer information, leading to higher model accuracy. Since these embedding vectors are added to each of the grouped neighbor feature vectors before max pooling, minor modifications are required to our kernel to support them. Algorithm 2 and 3 demonstrate our fused aggregation process and the highlighted lines indicate the changes required to support explicit positional embedding.

In Algorithm 2 the original forward pass of fused aggregation includes loading the target columns to SRAM (Line 5), performing max-reduction on the fly in SRAM (Line 8), and saving source index for backwards (Line 10). To support positional embedding, two major changes are made. First, element-wise addition of positional embedding vector to input point feature vector is performed (Line 6-7). This operation is fused into our kernel, operating on the fly in SRAM so that it does not incur unnecessary accesses to DRAM. Second, we need to save an additional source index table *source_pe* to gradients of positional embedding (Line 11) as we did for gradients of feature vectors (Line 10). These indices are used later in backward pass to scatter gradients for the positional embedding. The gradients of positional embedding are needed since they are used to update the weights of the small-sized MLP, which is used for generating positional embedding.

Algorithm 2 Forward Pass of Fused Aggregation

```
Input $P_{in} \in R^{N \times d'}$, $pos\_emb \in R^{n \times n_{neigh.} \times d'}$, $I_{neigh.} \in I^{n \times n_{neigh.}}$
Output $P_{out} \in R^{n \times d'}$
 1: Define array $source \in I^{n\times n_{neigh.}}$
 2: Define array $source\_pe \in I^{n\times n_{neigh.}}$
 3: for $i \leftarrow 0$ to $i \leftarrow n$ do    // Fully parallelized by GPU
 4:     for $j \leftarrow 0$ to $j \leftarrow d'$ do    // Fully parallelized by GPU
 5:         Load $P_{in}[I_{neigh.}[i]][j]$ from DRAM to SRAM    // 1. Fused Group
 6:         Load $pos\_emb[i][:][j]$ from DRAM to SRAM
 7:         On SRAM, $temp \leftarrow P_{in}[I_{neigh.}[i]][j] + pos\_emb[i][:][j]$
 8:         // 2. Max Reduction
 9:         On SRAM, $max, max\_idx$, $max\_idx\_pe \leftarrow$ max_reduce($temp$)
10:         Write $P_{out}[i][j] \leftarrow max$ to DRAM
11:         Write $source[i][j] \leftarrow max\_idx$ to DRAM    // Saved for backwards
12:         Write $source\_pe[i][j] \leftarrow max\_idx\_pe$ to DRAM    // Saved for backwards
```

Algorithm 3 Backward Pass of Fused Aggregation

```
Input $G_{in} \in R^{n \times d'}$, $source \in I^{n\times n_{neigh.}}$, $source\_pe \in I^{n\times n_{neigh.}}$
Output $G_{out} \in R^{N \times d'}$, $G_{emb} \in R^{n \times n_{neigh.} \times d'}$
 1: Initialize $G_{out}$ with zeros.
 2: for $i \leftarrow 0$ to $i \leftarrow n$ do    // Fully parallelized by GPU
 3:     for $j \leftarrow 0$ to $j \leftarrow d'$ do    // Fully parallelized by GPU
 4:         // 3. Fused Scatter & Sum Reduction
 5:         Load $G_{in}[i][j]$ from DRAM to SRAM
 6:         Load $G_{out}[source[i][j]][j]$ from DRAM to SRAM
 7:         On SRAM, $temp \leftarrow G_{in}[i][j] + G_{out}[source[i][j]][j]$ // Line 7 and 8 are
 8:         Write $G_{out}[source[i][j]][j] \leftarrow temp$ to DRAM    // processed atomically.
 9: for $i \leftarrow 0$ to $i \leftarrow n$ do    // Fully parallelized by GPU
10:     for $j \leftarrow 0$ to $j \leftarrow d'$ do    // Fully parallelized by GPU
11:         Write $G_{emb}[i][source\_pe[i][j]][j] \leftarrow G_{in}[i][j]$ to DRAM
```

In Algorithm 3 the original backward pass of fused aggregation was to scatter and sum the gradients to the place where the corresponding features are originated (Line 5-8). To support positional embedding, the process of generating gradients for positional embedding is added. Input gradients G_{in} are scattered according to the indices in *source_pe* and generate the $n \times n_{neigh.} \times d'$ sized gradient matrix G_{emb}, which is the gradient of positional embedding (Line 11). This process makes it inevitable to generate the $n \times n_{neigh.} \times d'$ sized intermediate values. This incurs extra DRAM accesses, leading to a decrease in the amount of savings in memory accesses with the fused aggregation technique. However, its impact on the speedup of fused aggregation is minimal since the atomic sum-reduce operation used to generate gradient G_{out} accounts for most of the backward latency. Our fused aggregation technique supporting positional embedding still eliminates $n \times n_{neigh.} \times d'$ sized sum-reduce operations and this allows us to achieve significant speedups. The experiment result (Section 5.3) in the main paper uses this extended fused aggregation technique.

B.2 Performance of PointNet++ with Delayed Aggregation and Positional Embedding

We have used a strengthened baseline in our experiments by augmenting PointNet++ [4] with delayed aggregation [2] and positional embedding [3] techniques. Delayed aggregation is a technique that changes the order of grouping and MLP to reduce the computation required for MLP. Original PointNet++ first groups the neighbors for each output point, applies MLP to the grouped neighbor vectors and then max-reduce to aggregate features. However, applying MLP to all the grouped neighbor vectors is inefficient since the grouped vector matrix is quite large. Mesorasi [2] solves this problem with delayed aggregation technique that first applies MLP to N input feature vectors and then groups the neighbor vectors. In this way, the input size of MLP reduces from $n \times n_{neigh.} \times d$ to $N \times d$, where n is the number of output points, $n_{neigh.}$ is the number of neighbors, and d is a feature dimension. Considering that stride is usually 2 or 4 and $n_{neigh.}$ is 32, $N << n \times n_{neigh.}$, and this results in a significant reduction in the computation for MLP.

Despite its computational efficiency, delayed aggregation causes accuracy loss, since it is hard to effectively encode the relative position information between output point and neighbors when delayed aggregation is applied. This is because MLP is applied before neighbors are grouped, making it challenging for the MLP to encode the positional relationship between output point and neighbors. PointMetaBase [3] addresses this issue by proposing explicit positional embedding. Combined with positional embedding, delayed aggregation substantially improves the model performance, even surpassing the baseline for some cases.

We apply delayed aggregation and positional embedding to the vanilla PointNet++ for our experiment to strengthen the efficiency and the accuracy of the baseline model. We report mIOU of the original PointNet++ and our strengthened version of PointNet++ in Table 3 to support our claim.

Dataset	Original	Augmented
S3DIS (mIOU)	60.65	63.19
ScanNet (mIOU)	53.20	59.42

Table 3: Performance Comparison between Original PointNet++ and Augmented PointNet++

C Data Augmentations

C.1 Data Augmentations Used in Experiments

We report the data augmentation techniques used for our experiments. We have followed the default setting of data augmentations used in PointMetaBase [3] repository. Table 4 lists data augmentation techniques used in each model and dataset. Random jittering adds independent noise to each point, while random scaling and rotating randomly scales and rotates the whole object or scene. Random dropping is a technique that randomly drops certain amount of points. Random shuffling is a technique that is used to give randomness to the sampling stage by randomly changing the seed point every epoch. Finally, random cropping is a technique that randomly crops the input scene to adjust the size of the scene to GPU memory capacity. We provide details about random cropping in Section C.2. We only report data augmentations that affect the *xyz* coordinates of the point cloud scene.

Model (Dataset)	Jitter	Scale	Rotate	Drop	Shuffle	Crop
PointNet++ (S3DIS)	✓	✓	✓	✗	✓	✓
PointMetaBase-L (S3DIS)	✓	✓	✓	✗	✓	✓
PointMetaBase-XL (S3DIS)	✓	✓	✓	✗	✓	✓
PointNet++ (ScanNet)	✗	✓	✓	✓	✓	✓
PointMetaBase-L (ScanNet)	✗	✓	✓	✓	✓	✓
PointMetaBase-XL (ScanNet)	✗	✓	✓	✓	✓	✓

Table 4: Data Augmentations used in Our Experiments

C.2 Random Cropping Support for L-FPS

Random cropping [5] is a method used to resize an input scene by randomly cropping a particular portion, enabling the input scene to fit within the GPU memory constraint. The uncertainty of the cropped region makes identifying sampled points prior to training become a nontrivial task.

L-FPS technique can integrate with the random cropping technique by incorporating minor modifications into the dataloader. Algorithm 4 outlines the extended L-FPS strategy for compatibility with the random cropping.

The key distinction in the offline phase lies in applying FPS to the entire point cloud scene before being cropped, instead of applying FPS to the cropped point set that is fed to the model. Given the entire point cloud scene P, we sample $|P|/stride$ points with L-FPS strategy and store the indices in the disk. For the same *stride* value (i.e. downsampling rate), we anticipate retrieving the desired number of points ($N_{crop}/stride$) from the $|P|/stride$ number of saved indices with a high probability after the scene P is cropped into N_{crop} points (note that

Algorithm 4 Random Cropping Support for L-FPS

Input *random_crop*: True **if** random crop **otherwise** False
Parameter *stride*: stride of the first layer, *epoch*: number of training epochs, $\mathbf{N}_{crop}$: number of the cropped points

```
 1: /* Offline Phase */
 2: fps_list ← list()
 3: for $P$ in dataloader do
 4:     fps_idx ← L-FPS($P$, $|P|/stride$)
 5:     fps_list.append(fps_idx)  // $epoch$ versions of sampling results
 6: Save fps_list to DISK
 7:
 8: /* Online Phase */
 9: Load fps_list from DISK to dataloader
10: for $e$ in range($epoch$) do
11:     for $P$, fps_idx[$e$] in dataloader do
12:         if $random\_crop$ is True then
13:             crop_idx ← random_crop($P$, $N_{crop}$)
14:             p_idx ← Append((fps_idx[$e$]&crop_idx), (crop_idx - fps_idx[$e$]))
15:         else
16:             p_idx ← range($|P|$)
17:             p_idx ← Append(fps_idx[$e$], (p_idx - fps_idx[$e$]))
18:         model($P$[p_idx])
```

$N_{crop} < |P|$). For example, we assume that $|P|$ is 48000, N_{crop} is 24000, and *stride* is 4. If we save $|P|/stride = 12000$ farthest points from P in the offline stage, we can stochastically retrieve approximately 6000 (i.e., $N_{crop}/stride$) valid points from 12000 saved indices after cropping.

The online phase also requires some minor modifications. When random cropping is not employed, we simply reorder the input scene P with p_idx, which is an index list with fps_idx is placed at the beginning and the rest of indices at the end (Line 17). By feeding the reordered input scene to the model, we can simply fetch the first $|P|/stride$ points to get the sampling results of the first layer while training. However, if random cropping is applied, minor modifications are required. Once the random crop function determines which points to crop (i.e., crop_idx) (Line 13), p_idx list is generated through a different process. We cannot directly use fps_idx as sampled indices since they might not exist in the cropped scene. We must first find the valid farthest point sampled indices from the crop_idx, which is done by applying an intersection operation between fps_idx and crop_idx (Line 14). Then, the valid sampled indices (fps_idx & crop_idx) are moved to the front and the rest of crop_idx is appended to the end, creating p_idx. Finally, we reorder P with p_idx and feed it to the model.

Adapting L-FPS to accommodate the random cropping inevitably introduces some minor approximations. We delve into this aspect in Section C.3.

C.3 Accuracy Impact of Random Cropping

In this section, we discuss the accuracy impact of the random cropping in L-FPS approach. There are two main reasons that random cropping might influence the sampling quality of L-FPS. First, the number of indices obtained through the intersection of fps_idx and crop_idx may be smaller than that of sampled points required in the first layer (i.e., $|fps\_idx \& crop\_idx| < |P|/stride$). This may allow some points that are not sampled through L-FPS to be used as a downsampled points. Second, the reused sampling results are from the entire point set, not from the cropped set. Sampling from the entire point set and subsequently cropping the result may not yield the equivalent sampling results compared to directly sampling from the cropped point set.

Despite these potential impacts, the empirical results gathered through performance evaluation in Section 5.2 confirms that these changes have a minimal impact on both sampling quality and model performance. To visually illustrate the (minor) impact on sampling quality, Section D.2 presents 3D visualizations of samples obtained using the original (vanilla) FPS, L-FPS, and random point sampling approaches, respectively.

D Additional Experiments

D.1 Sampling Algorithm Comparison

Alternative sampling algorithms In this section, we apply other representative sampling algorithms, i.e., random sampling and grid sampling, to the training pipeline and compare the model performance with L-FPS. For a fair comparison, we applied each sampling algorithm solely to the first layer, as we did for L-FPS. Additionally, the evaluation methodology remained consistent across all experiments. As shown in Table 5, L-FPS outperforms other sampling algorithms in most cases. Visualization results in Section D.2 further substantiate the superiority of L-FPS over other sampling algorithms.

Dataset	Model	Accuracy (diff.)		
		L-FPS	GS	RPS
S3DIS	PN++	63.39	62.87 (▼ 0.52)	62.40 (▼ 0.99)
	MB-L	69.76	69.25 (▼ 0.51)	68.75 (▼ 1.01)
	MB-XL	70.74	70.19 (▼ 0.55)	70.22 (▼ 0.52)
ScanNet	PN++	59.54	59.19 (▼ 0.35)	58.00 (▼ 1.54)
	MB-L	70.54	70.37 (▼ 0.17)	70.09 (▼ 0.45)
	MB-XL	71.74	71.93 (▲ 0.19)	71.66 (▼ 0.08)

Table 5: Model Performance of baseline and ours. GS, RPS stands for Grid Sampling and Random Point Sampling.

Faster algorithm for exact farthest point sampling QuickFPS is a co-designed software and hardware solution, proposing an accelerator equipped with a k-d tree based fast and exact farthest point sampling algorithm. Since QuickFPS is an accelerator, we use software version of QuickFPS [1] open sourced by the authors for fair comparison with L-FPS. QuickFPS does not incur any accuracy loss since no approximation is applied in their algorithm. However, they suffer from limited end-to-end speedup compared to L-FPS as demonstrated in Figure 1.

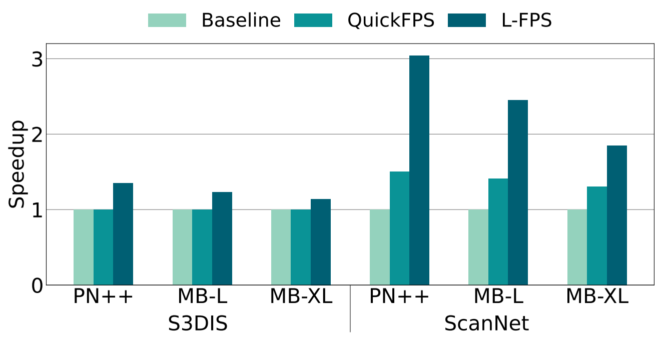
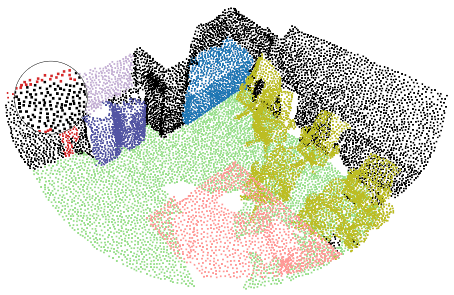


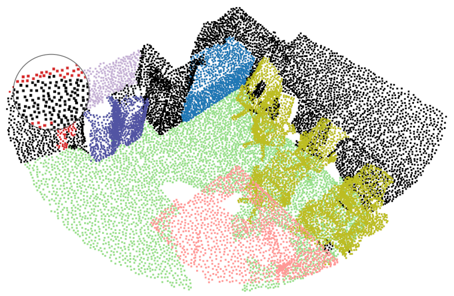
Fig. 1: Normalized training throughput.

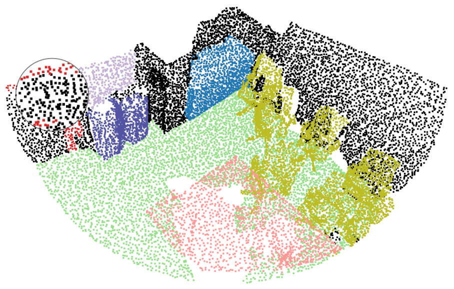
D.2 Sampling Quality Analysis via Visualization

We visualize the sampling quality of various methods: our proposed L-FPS, the original FPS, random sampling, and grid sampling. Figure 2 and 4 illustrate the results on ScanNet and S3DIS datasets, respectively. The sampling results demonstrate that both FPS and L-FPS approaches achieve high sampling quality, preserving the structure of objects and scenes across layers. However, grid and random sampling exhibit clustering and fail to preserve even spacing between each point.

Figure 3 and 5 show the distribution of minimum point spacing among the sampled points, which substantiates our observations. As explained in Section 5.2, L-FPS maintains near-identical point spacing compared to FPS and consistently exceeds the threshold for both the ScanNet and S3DIS dataset, while other representative sampling methods fall short.


(a) Farthest Point Sampling (FPS)


(b) Lightweight FPS (L-FPS)


(c) Grid Sampling

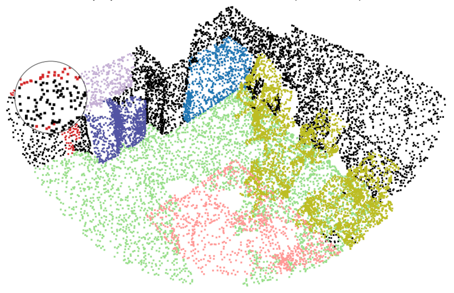

(d) Random Point Sampling (RPS)

Fig. 2: FPS, L-FPS , and Random Point Sampling Results for ScanNet Dataset.

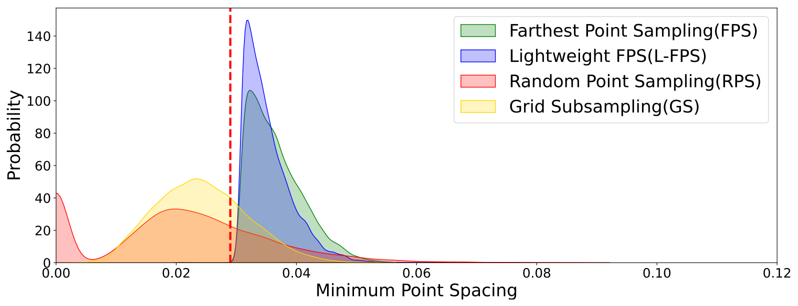


Fig. 3: Distribution of minimum point spacing for ScanNet Dataset.

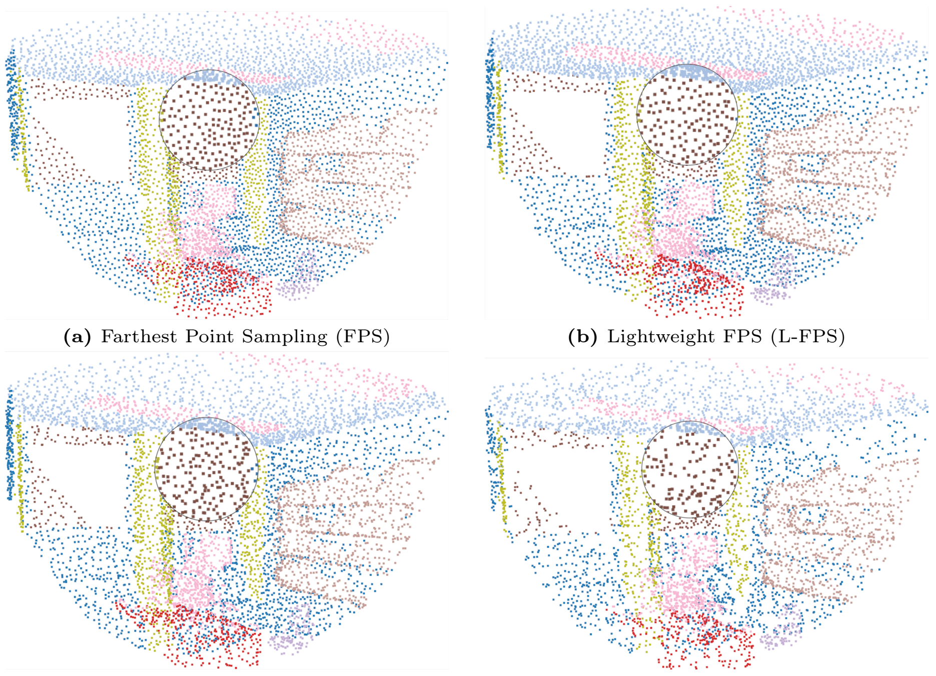

(a) Farthest Point Sampling (FPS)
(b) Lightweight FPS (L-FPS)
(c) Grid Sampling
(d) Random Point Sampling (RPS)

Fig. 4: FPS, L-FPS , and Random Point Sampling Results for S3DIS Dataset.

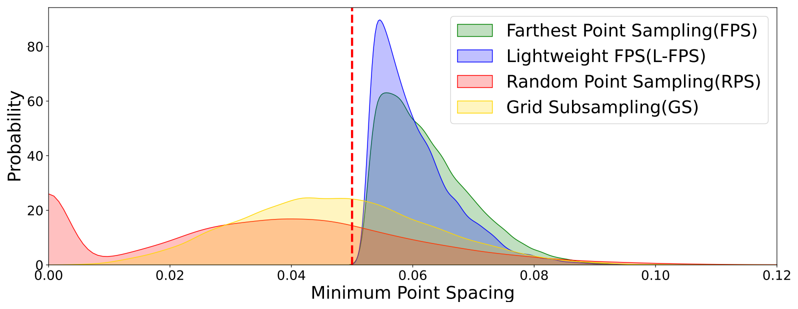


Fig. 5: Distribution of minimum point spacing for S3DIS Dataset.

D.3 Raw Training Time Numbers

We report the raw training time numbers, which we used to calculate the normalized end-to-end training throughput in the main paper.

Dataset	Model	Training Time (hours)	
		Baseline	Ours
S3DIS	PN++	3.27h	2.17h
	MB-L	4.78h	2.82h
	MB-XL	9.10h	4.79h
ScanNet	PN++	40.97h	12.70h
	MB-L	46.58h	15.25h
	MB-XL	60.16h	22.23h

Table 6: Raw training time of baseline and ours.

D.4 Speedup of Aggregation Only

In this section, we report the aggregation-only speedup. We achieve 4.36× geomean speedup on the aggregation operation with the fused aggregation technique. We also report forward and backward aggregation-only speedup. The speedup of backward aggregation is slightly higher (4.47×) than that of forward pass (4.28×), since the amount of memory reduction achieved through fused aggregation is larger in the backward aggregation. The speedup numbers are consistent across models and datasets. This is because all models use the same strides ($stride = 4$) and the same number of neighbors ($n_{neigh.} = 32$), which makes the amount of memory reduction comparable across all models and datasets.

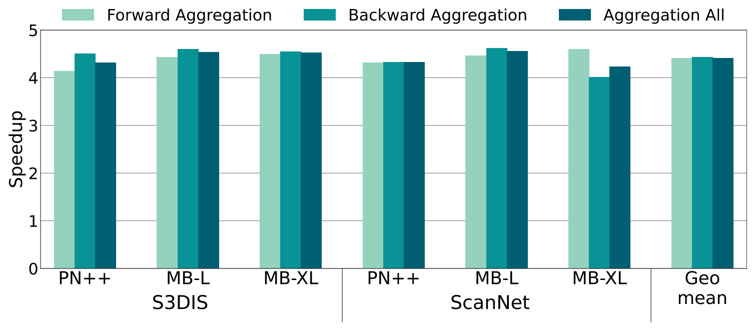


Fig. 6: Speedup of aggregation stage.

D.5 Applicability to other models

While fused aggregation is tailored to PointNet, L-FPS can be used for any point cloud model that utilizes FPS. To demonstrate this, we have applied L-FPS to Point Transformer (P-Trans) [6] to achieve a 1.47× training throughput improvement (using 4 GPUs) without sacrificing accuracy (refer to Table 7).

D.6 Applicability to other datasets

To demonstrate wide applicability of our proposal, we apply L-FPS and fused aggregation to two additional datasets: SemanticKITTI (outdoor semantic segmentation) and ModelNet40 (classification). Table 7 shows that our techniques achieve substantial training throughput improvements of 2.22× on SemanticKITTI (SK) and 1.05× on ModelNet40 (MN), all while maintaining accuracy. The relatively modest speedup on ModelNet40 results from the limited effectiveness of L-FPS on smaller point clouds (Section 6.5).

	MB-XL (SK)		PN++ (MN)		P-Trans (S3DIS)	
	Base	Ours	Base	Ours	Base	Ours
mIoU (%)	51.52	51.57	92.59	92.52	70.24	70.11
T_{train}(hours)	85.83	38.75	0.43	0.41	14.7	10.0

Table 7: Model performance and training time results across various models and datasets. OA is used for classification.

References

1. Quickfps. `http://github.com/hanm2019/bucket-based_farthest-point-sampling_GPU`
2. 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)
3. 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)
4. 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)
5. 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)
6. Zhao, H., et al.: Point transformer. In: ICCV (2021)