An Adjustable Farthest Point Sampling Method for
Approximately-sorted Point Cloud Data

Jingtao Li†, Jian Zhou†, Yan Xiong*, Xing Chen*, Chaitali Chakrabarti*
†School of Electrical Computer and Energy Engineering, Arizona State University, Tempe, AZ, 85287
‡ASML-HMI, San Jose, CA, 95134
*{jingtao1, yxiong35, xchen382, chaitali}@asu.edu; †{zhoujian1991}@gmail.com

Abstract—Sampling is an essential part of raw point cloud data processing, such as in the popular PointNet++ scheme. Farthest Point Sampling (FPS), which iteratively samples the farthest point and performs distance updating, is one of the most popular sampling schemes. Unfortunately, it suffers from low efficiency and can become the bottleneck of point cloud applications. We propose adjustable FPS (AFPS) to aggressively reduce the complexity of FPS without compromising the sampling performance. AFPS, parameterized by $M$, divides the original point cloud into $M$ small point clouds and samples $M$ points simultaneously. It exploits the dimensional locality of an approximately sorted point cloud data to minimize its performance degradation. On a multi-core platform, AFPS method with $M = 32$ can achieve 30× speedup over original FPS. Furthermore, we propose the nearest-point-distance-updating (NPDU) method to limit the number of distance updates to a constant number. On the ShapeNet part segmentation task, using the NPDU method on AFPS on a point cloud with 2K-32K points helps achieve a 34-280× speedup with 0.8490 instance average mIoU (mean Intersection of Union), which is only 0.0035 lower than that of the original FPS.

Index Terms—LiDAR Sensor, 3D Point Cloud, Farthest Point Sampling, Multi-core Hardware

I. INTRODUCTION

With the wide application of LiDAR (Light Detection and Ranging) technology in autonomous driving and self-navigation robots, point cloud data has become a popular data source. However, processing large volume of point cloud data efficiently is a difficult problem especially when it is generated at a fast rate as in LiDAR. Recently, PointNet++ [1] was shown to achieve excellent performance in processing raw point cloud data with low latency compared to traditional processing schemes that depend on converting point clouds into other formats. PointNet++ is based on a hierarchical set abstraction layer that consists of sampling and grouping operations, where sampling obtains representative points and grouping derives groups of neighboring points of each sampled point.

There are several popular sampling schemes. Of these, Random Point Sampling (RPS) achieves low latency but has poor performance for a sparse point cloud since it can miss sampling points in sparse regions. Farthest Point Sampling (FPS), which iteratively samples the farthest point and performs distance updating, is a popular sampling scheme [1]–[3]. However, FPS suffers from poor hardware efficiency and is the latency bottleneck of PointNet++ model when executed on a Graphics Processing Unit (GPU).

To improve the sampling efficiency, alternative approaches have been investigated. Learning-based methods are used in [4], [5] while an efficient Grid-GCN sampling method is used in [6]. These approaches either require re-design of the network or an expensive pre-processing step, leading to extra design effort and implementation overhead.

The computational complexity of FPS is $O(NC)$, where $N$ is the number of points in the point cloud and $C$ is the number of sampled points. One way of reducing the time complexity is by dividing the original point cloud data into multiple sectors and performing local FPS on each one of them. The sampling performance of such a method can be quite poor if the original (unsorted) point cloud data is divided arbitrarily. However, we observe performance degradation can be avoided if the point cloud data is sorted or approximately sorted.

In this paper we propose adjustable FPS (AFPS) which divides the original point cloud into multiple point sectors along the (sorted) dimension, and performs FPS on data in each sector. AFPS exploits the locality of data in the sorted dimension and is able to maintain algorithmic performance comparable to FPS while achieving significant reduction in the time complexity. We also propose the nearest-point-distance-updating (NPDU) heuristic to further improve the efficiency of performing local FPS in each small sector. By only performing distance updates for points close to the newly sampled point, the number of distance update operations in each iteration is reduced from $O(N)$ to a constant number. Use of the NPDU method on AFPS achieves even better efficiency with minimal performance loss in most cases. We evaluate the performance of our algorithms on Transmuter, a multicore platform [7]. Our evaluation of three down-stream tasks shows the massive speedup of proposed NPDU-AFPS with minor degradation in task performance. In summary, we make the following contributions:

- We propose AFPS, a parametrizable sampling scheme that partitions the point cloud into $M$ point sectors and performs small-scale FPS on local data in parallel. We show if the point cloud is approximately sorted, AFPS brings minimal performance loss while achieving a huge reduction in time cost. The method is amenable to parallel implementation and easy-to-use in current FPS-based point cloud processing schemes.
• To further reduce the complexity, we propose a neighboring point distance updating scheme to reduce the number of distance updates per iteration to a constant, instead of $O(N)$ in the original FPS.
• Compared to original FPS, AFPS can achieve 30× speedup with $M = 32$, on the Transmuter platform. On the ShapeNet part-segmentation task, AFPS with NPDU can achieve 34× speedup with a slightly worse task performance of 0.8490 in instance average mean Intersection of Union (mIoU), compared to the original FPS of 0.8525.

The rest of the paper is organized as follows: Section II introduces point cloud processing and FPS. Section III presents details of our proposed AFPS method and NPDU heuristic. Section IV gives extensive evaluation on performance and efficiency of proposed methods, and Section V concludes this work.

II. BACKGROUND

A. Point Cloud Processing

PointNet++ [1] processes raw point cloud data directly and achieves good performance compared to traditional method based on 3D voxels or 2D views. Unlike schemes such as [8], [9] that use global max pooling of the point cloud and might miss detailed local features, PointNet++ uses set abstraction layer, which extracts local features and uses them to build features for larger regions. Set abstraction layer adopts sampling and grouping as the key operation for feature extraction, and FPS is adopted as the sampling operation. FPS has been widely used in modern point cloud processing schemes [1]–[3] even though it dominates the PointNet++ inference time. For example, on an RTX-2080ti GPU with 4352 CUDA cores, FPS takes about 53.7% of the total time.

B. Farthest Point Sampling (FPS)

While FPS was introduced in traditional image processing long back [10], it has gained popularity with PointNet++ [1]. FPS is described in Algorithm 1. It is a greedy algorithm that samples the point that is the farthest away to visited points in each iteration. The visualization of this process is shown in Fig. 1. After the first point is sampled randomly, all other points calculate their Euclidean distance to the sampled point, and store the distance in an array. Then, the point that has the largest distance is selected as the next point to sample (line 12), and the distance array is updated if the current distance being calculated is smaller than the old distance (line 8, 9).

We observe two important properties of FPS:

(P1): Points that are far from the newly sampled point have high priority to be sampled.

(P2): In each iteration, only one point is sampled, followed by $O(N)$ distance calculations.

Because of (P1), FPS is able to capture prominent point features and is robust in sampling point clouds regardless of the sparsity. However, because of (P2), FPS has poor hardware efficiency and lacks scalability.

The baseline FPS has a high computation complexity of $O(NC)$ and so [10], [11] propose to use a tree-based data structure to reduce its complexity. However, implementing tree-based optimizations are difficult for a multi-core system such as GPU, and implementing the original FPS is still the more favorable option [12]. This paper address the poor performance of FPS on a multi-core processor by proposing a heuristic version of FPS that is both light in computation and easy for parallel implementation.

There are other alternate solutions to replace FPS and we mention them here for completeness. They include DNN-based sampling methods [4], [13] that uses a differentiable DNN model to determine the sampling and thus can be optimized during training, DNN-assisted sampling methods [5], [14] that compensate random point sampling with a dedicated neural network module to achieve good scalability, etc. However, these learning-based sampling methods rely on a parameterized DNN module, which can be sensitive to overfitting. Another promising approach, Grid-GCN [6], adds

\begin{algorithm}
\caption{Farthest Point Sampling}
\label{alg:fps}
\begin{algorithmic}[1]
\algorithmicrequire{Point cloud $P$ of shape $(N, 3)$ denoted by number of points $N$ and point cloud dimension (3). The number of points to be sampled is $C$.}
\begin{algorithmic}
\State \textbf{// Initialize distance}
\State $\text{Distance}[N] = \{0\}$;
\State $\text{Sampled}[C] = \{-1\}$;
\State \textbf{// Sampling start}
\State $\text{sample\_idx} = \text{randint}(0, N)$;
\State $\text{Sampled}[0] = \text{sample\_idx}$;
\For{$i \leftarrow 0$ to $C - 1$}
\State \textbf{// Update distances}
\For{$j \leftarrow 0$ to $N - i$}
\State $\text{dist} = \text{geo\_dist}(P[j,:], P[\text{sample\_idx},:], D)$;
\If{$\text{dist} \leq \text{Distance}[j]$}
\State $\text{Distance}[j] = \text{dist}$;
\EndIf
\EndFor
\State \textbf{// Sample next}
\State $\text{sample\_idx} = \text{argmax}(\text{Distance})$;
\State $\text{Sampled}[i] = \text{sample\_idx}$;
\EndFor
\end{algorithmic}
\end{algorithmic}
\end{algorithm}

Fig. 1. Visualization of FPS process.
a voxelization pre-processing step to put raw point cloud data into a regular grid before doing a scalable sampling. But this pre-processing step is non-trivial (will be shown in later experiments) and lead to poor latency in some cases.

III. Method

A. Dimensional Locality

Most point cloud data is not unstructured and unsorted as previously assumed. In fact, point cloud data points stored in consecutive memory locations are found to be related; we refer to this characteristic of the data as dimensional locality. For example, point cloud data collected by 3D scanning LiDAR [15], and 3D point cloud data from 2D-LiDAR obtained by a 2D LiDAR sensor moving vertically [16], [17], have this property.

We model the two data collection methods in Fig. 2. The scanning 3D LiDAR senses the environment by rotating around the z-axis, according to Fig. 2(a). Thus the point cloud can be seen as approximately sorted along the x-axis, where the bin size of the approximate sorting algorithm is dependent on the Field of View (FoV) of the sensor and proximity of the object to the sensor. The 2D LiDAR is modeled in Fig. 2(b). It is equipped with a step motor and moves vertically so the point cloud data is sorted along the z-axis.

![Fig. 2. LiDAR point cloud data collection process. (a) Scanning 3D LiDAR, where point cloud data is approximately-sorted. (b) 2D LiDAR with step motor that moves the sensor vertically generates point cloud data that is exactly-sorted in the vertical direction.](image)

If dimensional locality is present in the point cloud, the proposed AFPS has minimal loss in sampling quality, as will be shown in later experiments. Dimensional locality can also be used to find approximate “point neighbors” for our NPDU heuristic. For cases where point cloud data does not have locality, some form of binning can be used – exact sorting is not required. However, in this paper, we focus on cases that naturally have dimensional locality and are free of sorting overhead as shown in Fig. 2.

B. Adjustable FPS

A direct approach to accelerating the original FPS is to divide the distance calculations of all points into many cores and processing them in parallel. However, such an approach does not scale well with the number of cores. Synchronization is needed to derive global farthest point when all cores finish updating their distances in each iteration and so as the number of cores increases, synchronization gradually dominates the time cost. Our approach is quite different. We divide the original point cloud $P$ into $M$ disjoint sectors according to their storage location and perform a local FPS on each sector. Since FPS is conducted on non-overlapping sectors, there is no need for synchronization. Our approach reduces the total number of computations by a factor of $M$ since to sample $C$ points using AFPS, only $C/M$ iterations are needed. Thus, the algorithm complexity is reduced to $NC/M$.

However, implementing AFPS on unsorted point cloud comes with loss in sampling quality. AFPS only samples $N/M$ points from each sector. In the extreme case, if an unsorted point cloud happens to have all the $N$ “representative points” in the same sector, sampling only $N/M$ points from that sector may result in missing most of the “representative” points. But if the point cloud data has dimensional locality, the “representative points” are likely to be distributed more uniformly in different sectors, resulting in fewer misses. Thus, while this method could suffer from some small performance degradation, it results in significant reduction in number of computations. We set $M$ as an adjustable parameter in AFPS.

In one extreme case, if $M$ is equal to the number of sampled points, the algorithm can finish in a single iteration and is equivalent to RPS. On the other, if $M$ is equal to one, AFPS is the same as the original FPS. We can choose the value of $M$ to achieve great latency reduction with little impact on task performance, as will be shown in Section IV.

C. NPDU Heuristic

From property $P_1$ of the original FPS, we see that for each newly sampled point, we need to exclude points that have small distances from the points that have been sampled and should encourage points that have large distances since they are better potential sampling candidates. We want to keep this property while limiting the number of distance updates to further reduce the computational complexity.

Towards this goal, we propose the nearest points distance updating (NPDU) method. For a point cloud that has dimensional locality, we only update the distances of points that are stored in the nearest neighboring locations of the newly sampled point. As shown in Fig. 3(a), the horizontal bar represents a point cloud that is stored in the memory. With NPDU, only $k$ neighboring points’ distances will be updated in each iteration. Notice that NPDU can be easily applied on top of the AFPS as shown in Fig. 3(b). It limits the number of distance updates of each smaller-scale FPS and further reduces the computational complexity. NPDU heuristic preserves property $P_1$ of the original FPS in the following manner: due to dimensional locality, it is sufficient to choose points that are approximate neighbors, that is, close in one or more dimensions. This because points that are close in one dimension are more likely to be neighbors in 3D space. Second, once distance updating of neighboring points is done, sampling of neighboring points is less likely to happen in future iterations. By applying NPDU, we can limit distance updating in each iteration to a constant $k$ per sector, and effectively reduce the time cost since distance updating is the most expensive operation. Note that NPDU does not reduce
algorithm complexity since the max operation in each iteration still has \(O(N)\) complexity.

![Image](image.png)

Fig. 3. Illustration of NPDU heuristics. (a) Apply NPDU on original FPS with \(k = 8\); distance updates are done only for points that are located near the sampled points. (b) Apply NPDU on AFPS with \(M = 2\) and \(k = 4\); point-cloud is divided into sectors, and NPDU is applied on each sector.

IV. EXPERIMENTAL RESULTS

A. Experiment Settings

We implement the proposed methods on top of a Pytorch repository of PointNet++ [18], which includes classification task on Modelnet40 [19], Part Segmentation task on ShapeNet [20] and Semantic Segmentation task on S3DIS [21]. The details of the three tasks are shown in Table I. The input size is represented by \((B, N, D)\), where \(B\) is the batch size, \(N\) is the number of points, and \(D\) is the number of dimensions.

We use “task performance” as an accuracy metric of the point cloud sampling method. For the evaluations presented here, we apply sampling methods in a pre-trained PointNet++ model, and perform testing on down-stream tasks. For instance, we use class accuracy for object classification task and semantic segmentation task, and instance average mIoU (mean Intersection of Union) for part segmentation task \(^1\). We take the average of 10 evaluations for each data point to eliminate randomness.

| Task                  | Input Size | Evaluation Metric          |
|-----------------------|------------|----------------------------|
| Object Classification | (24, 1024,3) | Class Accuracy             |
| Part Segmentation     | (24, 2048,3) | Instance Average mIoU     |
| Semantic Segmentation | (24, 4096,3) | Class Accuracy             |

For evaluation, we use Transmuter [7] (TM) as the testbed. TM is a multi-core architecture that optimizes both programmability and efficiency. We use a configuration with 4 tiles, where each tile consists of one LCP (local control processor) and 16 Arm cores. Cache memory consists of shared cache of 4KB L1 cache bank per core (shared across cores) and 4KB L2 cache bank per tile (shared across tiles). We use Gem-5 framework to get latency and cache statistics.

\(^1\)For detailed definition of each metric, please refer to PointNet_Pointnet2_pytorch

B. AFPS evaluation

We test the speedup of the proposed AFPS sampling method on the part segmentation task. As is shown in Fig. 4, compared to original FPS, AFPS reduces time cost significantly with increasing \(M\). It saturates at \(M = 16\) when GPE to LCP communication starts dominating the time cost. At \(M = 32\), the time cost is \(5.7 \times 10^{-4}\) second, a \(29.8\times\) speedup compared to the original FPS which takes \(0.017\) second. Thus, setting \(M = 32\) (or even lower) for this TM configuration is sufficient for addressing the bottleneck issue of this application. Note that choosing a large \(M\) provides marginal improvement on FPS performance and does not translate to the overall latency reduction as other components become the bottleneck.

![Image](image.png)

Fig. 4. Time cost of the proposed AFPS on part segmentation task measured on TM testbed for different values of \(M\). The input size is \((24, 2048, 3)\) and the sample size is \(N=512\).

Next, we investigate the task performance of the proposed AFPS on the part segmentation task, where we compare its performance on unsorted, approximately sorted, and exactly sorted point clouds. The results are shown in Table II. The baseline FPS achieves an average mIoU of 0.8522 on the part segmentation task. If the point cloud is unsorted, applying AFPS suffers a large performance drop; when \(M = 32\), it only achieves mIoU of 0.8474. However, if the point cloud is approximately sorted along dimension \(x\), the performance drop is much smaller. For approximately sorted data, we set the bin size\(^2\) at 128 points. For \(M\) equal to 2 or 8, AFPS on approximately-sorted point cloud has almost no drop in task performance compared to the baseline. We also have a similar observation on a point cloud that is exactly sorted. For 2D LiDAR with stepping motor where the point cloud is exactly sorted along \(z\) axis, the performance even increases to 0.8527 mIoU for \(M = 32\), which is even higher than that of the original FPS.

C. NPDU on original FPS

Next, we test the proposed NPDU heuristic in terms of hardware efficiency. The number of points of the input point cloud is varied from 1K to 16K and \(k\) is set equal to 8. The TM time cost and cache hit rate performance is shown in Fig. 5. When the point cloud size is small at 1K/2K, applying NPDU achieves a \(2\times\) speedup but when the input size increases to 8K, a peak speedup of around 11x can be achieved. The reason

\(^2\)every point in a bin with a larger index is greater than points in a bin with a smaller index, but points in a bin are unsorted.
for the large gain is that while the cache hit rate of FPS drops significantly, the cache hit rate of NPDU remains the same. However, when the point cloud size data increases to 16K, the difference in the cache hit rates reduces and the time cost difference between the two methods is not as spectacular.

Fig. 5. Time cost of FPS (baseline) and NPDU for different input sizes while keeping sample size to be 512. NPDU has good performance even for larger input data sizes.

We investigate the task performance and efficiency tradeoff of NPDU on the part segmentation task for different values of $k$ (4 to 64). As shown in Fig. 6, choosing a small $k$ brings the lowest time cost but results in a noticeable degradation in task performance, compared to original FPS that is represented by the red line. We observe choosing $k$ equal to 8 can achieve optimal mIoU performance with a lower time cost compared to larger $k$. However, the optimal point differs from case to case, especially when using NPDU with AFPS.

**D. NPDU on AFPS**

Next, we study a combination of applying NPDU on AFPS, which further reduces the time cost. In Fig. 7, the time cost of the baseline AFPS with different values of $M$ are shown by three gray lines. We can see a clear difference in time cost between different values of $M$. The time cost of NPDU with AFPS is shown by the blue bars. We see that NPDU with AFPS achieves 30% to 50% time cost reduction compared to the AFPS baseline. We also observe that for small $M$, using a small $k$ can impact the average mIoU task performance a lot, while other cases do not present obvious trend. To achieve the optimal tradeoff point, $k$ needs to be tuned differently for different choices of $M$ through experimentation. For the part segmentation task, for $M = 8$, the optimal $k$ is 8, and for $M = 16$, the optimal $k$ is 16.

Fig. 7. Time cost on TM and average mIoU for different values of $M$ and $k$ of NPDU AFPS. The input size 2048 and the sample size is 512. The three gray rails are AFPS w/o NPDU.

We also observe that AFPS in conjunction with NPDU is suitable for multi-core hardware such as TM. For a 4 tile TM architecture, for low $k$, when the number of cores per tile increases, NPDU has lower core utilization which results in speedup degradation. When the number of cores equals to 16, the speedup over a single core is only 13.72× (instead of 16×), as shown in Table III. This is because the workload of only $k$ distance calculations per iteration is small for distribution among 16 cores. When combined with AFPS, the total workload becomes $k \times M$, and the speedup is steady at around 2× when number of cores doubles.

Table III shows the speedup over 1-core-per-tile transmuter for different choices of $k$ and $M$.

The proposed NPDU AFPS can easily scale up to a larger problem. As shown in Table IV, NPDU AFPS with $M$ set to 32 achieves much better scalability. Using all 16 cores, AFPS can sample a 32K large point cloud in just 0.0022s and achieve a massive speedup of 280× compared to the original FPS.

### Table I

**Task Performance of Proposed AFPS on Point Clouds with Different Characteristics. (Baseline: 0.8522)**

| Characteristics | Task Performance (mIoU) |
|-----------------|------------------------|
|                 | $M = 2$ | $M = 8$ | $M = 32$ |
| unsorted        | 0.8503  | 0.8468  | 0.8447  |
| approx.-sorted  | 0.8526  | 0.8518  | 0.8488  |
| exactly-sorted  | 0.8519  | 0.8515  | 0.8489  |
| exactly-sorted  | 0.8493  | 0.8523  | 0.8527  |

![Table I](image)

### Table II

**Average mIoU**

| Method          | Number of cores per tile |
|-----------------|---------------------------|
| FPS             | 2 | 4 | 8 | 16 |
| NPDU FPS (k=8)  | 13.72x |
| NPDU AFPS (M=16, k=4) | 8.79x |
| NPDU AFPS (M=32, k=16) | 16.78x |

![Table II](image)

### Table III

**SPEED OVER 1-CORE-PER-TILE TRANSMUTER**

| Method          | Point Cloud Input Sizes |
|-----------------|-------------------------|
| FPS             | 2k | 4k | 8k | 16k |
| NPDU AFPS (M=32) | 0.0005 | 0.0007 | 0.0011 | 0.0022 | 0.0044 |

![Table III](image)
E. Comparisons

We compare both the time cost and task performance of other sampling methods with our proposed method. Comparison with Grid-GCN is done by implementing both voxelization that partitions point cloud into 3D grids with 40 grids in each dimension, and the random voxel sampling method used in the original paper [6] on the TM platform. We also include RPS and original FPS in our evaluations. For task performance, we include all three tasks described in Table I. Fig. 8 shows time cost in logarithmic scale in (a) and performance degradation compared to original FPS in (b). For time cost comparison, we can see that RPS achieves the lowest time cost while FPS and Grid-GCN have similar high time cost. The gray bar that represents our proposed NPDU AFPS achieves the second-lowest time cost. Grid-GCN has poor time cost performance because of the time-consuming voxelization step. It costs even more than the original FPS for part segmentation and object classification because of smaller number of points. For task performance, RPS has the highest degradation, whereas proposed NPDU FPS and AFPS achieve the smallest degradation. NPDU AFPS is slightly worse than AFPS, but it is consistently better than Grid-GCN or RPS for all three tasks.

V. CONCLUSION

In this work, we propose AFPS method that separates point cloud into multiple small sectors along a sorted dimension, and NPDU heuristic that limits the number of distance updates. We exploit the approximately-sorted point cloud pattern in the data collection step and use it to our advantage to compensate for the task performance degradation. Experimental results show the improvement on latency and scalability over the original FPS method. NPDU in combination with AFPS can achieve a 34× speedup on a 2K point cloud and 280× speedup on a 32K point cloud, compared to the original FPS. For task performance, the proposed NPDU AFPS has consistently better performance than RPS or Grid-GCN.

ACKNOWLEDGMENT

This paper was supported in part by a grant from AFRL and DARPA under agreement number FA8650-18-2-7864.

REFERENCES

[1] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, “PointNet++: Deep hierarchical feature learning on point sets in a metric space,” Advances in Neural Information Processing Systems, vol. 30, 2017.
[2] S. Shi, X. Wang, and H. Li, “PointRCNN: 3D object proposal generation and detection from point cloud,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 770–779.
[3] M.-H. Guo, J.-X. Cai, Z.-N. Liu, T.-J. Mu, R. R. Martin, and S.-M. Hu, “PCT: Point cloud transformer,” Computational Visual Media, vol. 7, no. 2, pp. 187–199, 2021.
[4] I. Lang, A. Manor, and S. Avidan, “SampleNet: Differentiable point cloud sampling,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 7578–7588.
[5] Q. Hu, B. Yang, L. Xie, S. Rosa, Y. Guo, Z. Wang, N. Trigoni, and A. Markham, “RandLA-Net: Efficient semantic segmentation of large-scale point clouds,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 11 108–1117.
[6] Q. Xu, X. Sun, C.-Y. Wu, P. Wang, and U. Neumann, “Grid-GCN for fast and scalable point cloud learning,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 5661–5670.
[7] S. Pal, S. Feng, D. Park, S. Kim, A. Amarnath, C. Yang, X. He, J. Beaumont, K. May, Y. Xiong, K. Kaysky, J. M. Morton, J. Sun, M. D. T. O'Boyle, M. Cole, C. Chakrabarti, D. T. Blaauw, H. Kim, T. N. Mudge, and R. G. Dreslinski, “Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration,” in PACT ’20, October 3-7, 2020. ACM, 2020, pp. 175–190.
[8] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “PointNet: Deep learning on point sets for 3D classification and segmentation,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 652–660.
[9] S. Ravanbakhsh, J. Schneider, and B. Poczos, “Deep learning with sets,” arXiv preprint arXiv:1611.04500, 2016.
[10] Y. Eldar, M. Lindenbaum, M. Porat, and Y. Y. Zeevi, “The farthest point strategy for progressive image sampling,” IEEE Transactions on Image Processing, vol. 6, no. 9, pp. 1305–1315, 1997.
[11] R.-T. Liu, C. Chang, Z.-Q. Man, and Z. Wang, “The farthest neighbor queries based on R-trees,” in 2015 International Conference on Machine Learning and Cybernetics (ICMLC), vol. 1, 2015, pp. 235–238.
[12] Z. Wang, A. Kazhdan, and M. Alexa, “Object-ness: A principled approach to object detection from point clouds,” in Computer Vision and Pattern Recognition, 2015, pp. 2465–2472.
[13] X. Yan, C. Zheng, Z. Li, S. Wang, and S. Cui, “PointASNL: Robust point clouds processing using nonlocal neural networks with adaptive sampling,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 5580–5589.
[14] W. Han, Z. Zhang, B. Caine, B. Yang, C. Sprunk, O. Alsharif, J. Ngiam, V. Vasudevan, J. Shlens, and Z. Chen, “Streaming object detection for 3D point clouds,” in European Conference on Computer Vision. Springer, 2020, pp. 423–441.
[15] Z. Fang, S. Zhao, S. Wen, and Y. Zhang, “A real-time 3D perception and reconstruction system based on a 2D laser scanner,” Journal of Sensors, vol. 2018, 2018.
[16] H. F. Murcia, M. F. Monroy, and L. F. Mora, “3D scene reconstruction based on a 2D moving LiDAR,” in International Conference on Applied Informatics. Springer, 2018, pp. 295–308.
[17] X. Yan, “Pointnet_pointnet2_pytorch,” https://github.com/yanx27/Pointnet_Pointnet2_pytorch, 2021.
[18] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, “3D ShapeNets: A deep representation for volumetric shapes,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1912–1920.
[19] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su et al., “ShapeNet: An information-rich 3D model repository,” arXiv preprint arXiv:1512.03385, 2015.
[20] 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, 2016, pp. 1534–1543.