EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices

Ziyu Ying
zjy5087@psu.edu
The Pennsylvania State University
State College, Pennsylvania, USA

Sandeepa Bhuyan
sxb392@psu.edu
The Pennsylvania State University
State College, Pennsylvania, USA

Yan Kang
ybk5166@psu.edu
The Pennsylvania State University
State College, Pennsylvania, USA

Yingtian Zhang
yjz5396@psu.edu
The Pennsylvania State University
State College, Pennsylvania, USA

Mahmut T. Kandemir
mtk2@psu.edu
The Pennsylvania State University
State College, Pennsylvania, USA

Chita R. Das
cxd12@psu.edu
The Pennsylvania State University
State College, Pennsylvania, USA

ABSTRACT

Recently, point cloud (PC) has gained popularity in modeling various 3D objects (including both synthetic and real-life) and has been extensively utilized in a wide range of applications such as AR/VR, 3D reconstruction, and autonomous driving. For such applications, it is critical to analyze/understand the surrounding scenes properly. To achieve this, deep learning based methods (e.g., convolutional neural networks (CNNs)) have been widely employed for higher accuracy. Unlike the deep learning on conventional 2D images/videos, where the feature computation (matrix multiplication) is the major bottleneck, in point cloud-based CNNs, the sample and neighbor search stages are the primary bottlenecks, and collectively contribute to 54% (up to 80%) of the overall execution latency on a typical edge device. While prior efforts have attempted to solve this issue by designing custom ASICs or pipelining the neighbor search with other stages, to our knowledge, none of them has tried to "structurize" the unstructured PC data for improving computational efficiency.

In this paper, we first explore the opportunities of structurizing PC data using Morton code (which is originally designed to map data from a high dimensional space to one dimension, while preserving spatial locality) and observe that there is a huge scope to "skip" the sample and neighbor search computation by operating on the "structurized" PC data. Based on this, we propose two approximation techniques for the sampling and neighbor search stages. We implemented our proposals on an NVIDIA Jetson AGX Xavier edge GPU board. The evaluation results collected on six different workloads show that our design can accelerate the sample and neighbor search stages by 3.68× (up to 5.21×) with minimal impact on inference accuracy. This acceleration in turn results in 1.55× speedup in the end-to-end execution latency and saves 33% of energy expenditure.

ISCA '23, June 17–21, 2023, Orlando, FL, USA.

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589113

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems; System on a chip**; • **Computing methodologies** → *Computer vision.*

KEYWORDS

Point Cloud, Deep Neural Network, Approximation, Energy-efficiency, Edge Device

ACM Reference Format:
Ziyu Ying, Sandeepa Bhuyan, Yan Kang, Yingtian Zhang, Mahmut T. Kandemir, and Chita R. Das. 2023. EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579371.3589113

1 INTRODUCTION

Point Cloud (PC), a representation of objects in 3D space using a huge collection of points, primarily used for terrain scanning by military and space agencies and various industrial applications (e.g., engineering, architecture, archaeology, etc.) has gained tremendous attraction over the past few years. Especially, the recent advent of various emerging applications requiring hyper-realistic visualizations of objects such as 360-degree or volumetric video streaming [32, 73] in Virtual Reality (VR), virtual object and real-world interaction in Augmented Reality (AR) [74], cultural heritage modeling and rendering [51], as well as applications demanding high-fidelity version of the physical world such as object detection in autonomous driving and robotics, have put PC in the forefront. Additionally, research advancements in point acquisition technologies (such as 3D scanning [61], photogrammetry [2], etc.) as well as the availability of affordable and convenient PC acquisition devices such as LiDAR, RGB-Depth cameras (e.g., Microsoft Kinect [41], Intel RealSense [25], etc.) and, more recently smartphones (e.g., iPhone 13 Pro[1], Samsung Galaxy S20 [53]) equipped with LiDAR Depth Camera have fuelled its widespread adoption. According to a Straits Research report, the PC market is forecasted to reach 6.93 Billion USD by 2030 [21].

The rising interest in PC has led to increase in PC data availability, thus making it feasible to leverage deep learning-based techniques such as classification, segmentation, and detection for PC data, thereby facilitating the machines with a better understanding of the scene to make more accurate decisions. Especially,

PC deep learning analytics can provide more useful inputs to autonomous cars, drones, robots, etc. by assisting with better visual perception to detect and measure the distance of objects precisely, failing which can result in devastating consequences. Moreover, the availability of LiDAR cameras and neural processing units (NPUs) on the commodity mobile/edge devices have made on-device PC inference an attractive option, thus eliminating the need for offloading computations to server which can potentially incur significant communication latency or energy consumption [4, 29, 38, 40].

However, unlike the deep learning analysis on 2D image data, which are structured (where neighboring pixels can be simply found by using indexes), deep learning on PC faces several challenges as the raw PC data are *irregular* (i.e., points are unevenly sampled across different regions) and *unstructured* (i.e., distances between neighboring points are not fixed). Due to these inherent properties of the PC data, two additional steps are employed on inferencing the raw PC data – *sample* and *neighbor search*. However, since the off-the-shelf deep learning acceleration works like [24, 54, 57, 69] are not tailored for such operations, the PC analytics applications do not run efficiently on existing hardwares. Our profiling data reveal that just the sample and neighbor search stages contribute to more than 50% of the overall PC inference latency on a typical edge device. So, it is imperative to optimize these two stages, and thereby decrease the latency and energy consumption of PC inference on the edge devices with limited resources and battery life.

While most prior works along this direction primarily focus on improving the PC inference accuracy [23, 28, 36, 63], only few efforts to reduce the inference latency and energy consumption [18, 35] on edge devices. Mesorasi [18] accelerated the feature computation stage and pipelined the neighbor search stage to some extent, but did not address the sample stage. Albeit, PointAcc[35] developed custom hardware accelerator for PC inference, but did not consider leveraging potential software-level optimization opportunities by taking into consideration the unique characteristics of the PC data. We want to emphasize that the primary bottlenecks in PC inference – sample and neighbor search stages – are stemmed from the irregular and unstructured nature of the PC data. Hence, driven by insights from 2D image data, where sampling and neighbor search is faster due to its structured representation, "*structurizing*" or "*re-ordering*" the unstructured 3D PC data can provide potential opportunities for accelerating PC inferences.

Morton Code [27], a geometrical data representation method for mapping a multi-dimensional data to one-dimension while keeping the locality among data in tact, can be a potential candidate to achieve such structured representation. However, it is not straightforward to employ Morton Code to speedup the PC inference due to following **challenges**: 1) *how to* ***effectively structurize*** *the PC data using Morton code ?*; 2) *how to* ***efficiently utilize*** *the Morton reordered data ?*; and 3) *what are the* ***trade offs*** *between performance benefits, memory overheads as well as the inference accuracy?*

Towards this, in this work, we present and evaluate *EdgePC*, a framework to accelerate the PC inference pipeline as well as improve its energy efficiency on edge devices, with minimal accuracy loss. The key idea is to *leverage* Morton Code to re-arrange the raw points (which are irregular and unstructured) and eventually *maximize* the "structuredness" of point cloud frame. In an ideal scenario, the point cloud frames can be as structured as 2D images. As the re-ordered point cloud frame is more structured, the neighborhood property of the points can be directly inferred by the indexes of each point. In other words, we can treat the re-arranged 3D point cloud similar to a 2D image when performing the CNN inference. Therefore, instead of optimizing the SOTA sampler and neighbor searcher to reduce the execution latency (which have been the focus of prior works [17, 35, 71]), in this paper, we decide to intelligently "skip" these two stages by **approximating** the sampled points as well as their neighbors by simply picking the points with proper indexes from the "structured" point cloud frame, with *minimal* computation. Note however that, such approximation will yield sub-optimal samples and false neighbors (more details in Sec. 5), resulting in the accuracy drop in CNN inference. Therefore, to avoid such accuracy loss, we integrate the aforementioned Morton Code-based approximations into the CNN models and retrain the networks.

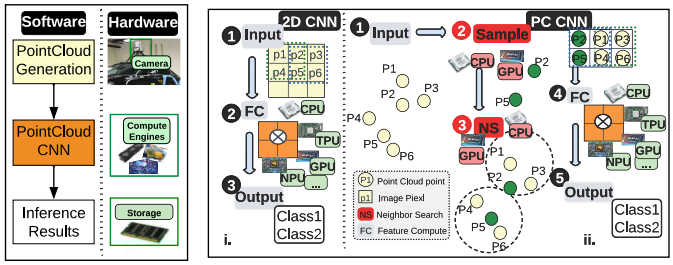
In summary, our major **contributions** are listed below:

- We first perform a detailed "end-to-end" (E2E) characterization of the typical PC inference application on the state-of-the-art (SOTA) PC analytics workloads and identify the *sample* and *neighbor search* stages to be the primary bottlenecks, due to their inefficient execution on the existing hardware. Furthermore, we demonstrate that this inefficiency stems from the inherent *irregular* and *unstructured* nature of the PC data.
- To address such inefficiencies, we propose a mechanism for using the *Morton Code* to *structurize* the raw PC data. Both our qualitative and quantitative analysis show the advantages of using Morton code in structural data representation. Next, we propose two complementary approaches to efficiently utilize this structured PC data to speedup inference: 1) *approximate the sample stage* by uniform sampling on structured PC data, and 2) an *index-based neighbor searcher* to approximate the SOTA neighbor searchers. Finally, to minimize the impact of these approximations on the CNN model precision, we conduct a thorough design space exploration to find an optimal design point to strike the right balance among inference accuracy, performance improvement and memory consumption, which is particularly critical for the limited compute-energy budget edge devices. We also incorporate our approximations for sample and neighbor search stages into the PC CNN models and retrain the weight matrices to minimize the inference accuracy drop.
- We implement our design on an edge GPU development board [43] and evaluate it on six different PC workloads. Our experimental results show that, *EdgePC* can speedup the sample and neighbor search stages by 3.68× which in turn translates to 1.55× speedup of the end-to-end PC inference latency. It also saves 33% of the original overall inference energy consumption with a negligible impact on inference accuracy (compared to the baseline setup).

2 BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Point Cloud and its Applications. Point cloud (PC), typically captured by the LiDAR cameras, consists of a set of unordered points, with each point associated with a 3D coordinate and its attributes like RGB colors, normals, reflectances, etc. Note that PC is being widely used to represent the 3D models due to its simplicity



(a) Example application. (b) 2D CNN Vs. PC CNN.

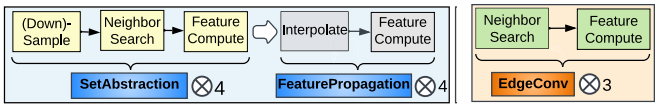
Figure 1: (a) PC CNN application example: autonomous driving car; (b) Comparison between 2D CNN and point cloud CNN.

(it does not require triangle mesh generation to construct surface) and high accuracy (it can preserve the original geometric information [30]). Such representation is essential for many applications like VR/AR [5, 49, 62], 3D reconstruction [8, 34, 37], autonomous driving [7, 9, 19], etc. Recently, deep learning has been utilized in these applications as it can achieve very high accuracy [10]. Fig. 1a shows an example to detect objects for autonomous vehicles using a PC-based CNN model. Specifically, the LiDAR camera first scans the surroundings and constructs the PC, following that the underlying compute engine (CPUs, GPUs or NPUs, etc.) performs the CNN inference on the captured PCs. Finally, the detected results (e.g., the cars, pedestrians, trees, etc.) are forwarded to the decision-making engine for the next steps. Such applications not only need high inference accuracy, but also execution efficiency in terms of faster execution for real-time updates, as well as low energy consumption when performed on edge devices. Therefore, the PC CNN inference needs to be carefully designed to meet the above requirements.

2.1.2 PC CNN Pipeline. Unlike CNN inference on 2D images, where the inputs are "well-structured" and each pixel and its neighbors can be easily located via "indexes", 3D PCs are essentially unstructured data, and thus, cannot be directly fed into the convolution layers. Given a PC frame, the first step in a PC-based CNN is sampling (obtain a global coverage for the PC frame) and followed by neighbor searching (find local neighbors for each sampled point). For example, as shown in Fig. 1b, first, points *P*2 and *P*5 (green color) are sampled from a PC frame. Then, their corresponding neighbors (the points within the dotted circle boundary) are determined. Next, these sampled points and their neighbors (*P*2,*P*1,*P*3; *P*5,*P*4,*P*6) form a 2D matrix (similar to the input shown in Fig. 1b for 2D CNN) and then the convolution layers are employed to extract the features. While the feature computations (FC) (which fundamentally perform matrix multiplications) can be accelerated using the off-the-shelf specialized accelerators like TPUs or NPUs, there are no such customized hardware for sample and neighbor search stages (primary bottlenecks). Thus, minimizing the execution time of these two critical stages and thereby reducing the overall PC CNN execution and energy consumption is the focus of this work.

2.2 Related Work

2.2.1 PC Analysis. Since Charles et al. proposed the PointNet [47] (the first/leading work that directly handles 3D PC data using deep learning CNNs), deep learning on PCs have been extensively studied in various domains, like 3D shape classification [64, 67, 70], object detection [33, 46, 56, 75] and tracking [20, 55, 59] or segmentation [11, 31, 47, 48]. While most of these prior works focus on improving the inference accuracy by designing better CNN architectures, there are also a few targeting to reduce the inference latency and/or improving the energy efficiency, as discussed next.



(a) PointNet++ (b) DGCNN

Figure 2: Architecture for (a) PointNet++(s) and (b) DGCNN(s).

2.2.2 Accelerating PC CNN. To accelerate PC CNN, Mesorasi [18] proposed a delayed-aggregation technique to minimize the feature computation (FC) latency and pipelined the FC and neighbor search stages, while PointAcc [35] proposed a specialized accelerator for mapping and memory management units. In [71], the authors minimized the data movement overheads by increasing the spatial locality of the PC data. Crescent [17] solved the irregular memory accesses in k-d tree-based neighbor search by splitting the tree into top- and bottom-trees. While these techniques can improve the performance of PC CNN to some extent, they have their own limitations. For example, [18] and [17] can only benefit the neighbor search step, but ignore the optimization opportunities for the sample stage, whereas [71] only targets graph-based CNNs, which has a limited application scope. Finally, PointAcc customizes specialized accelerators, which not only involves complex design and tuning cycles, but also misses the potential software-level optimizations, and therefore, could not fully exploit the off-the-shelf hardware such as edge GPUs.

Thus, to the best of our knowledge, there is no prior work investigating the optimization opportunities for the two critical stages in PC pipeline - sample and neighbor search - to minimize the execution time of PC inferences on *edge devices*.

3 MOTIVATION

We first study two widely-used PC CNN pipelines, and then breakdown their latencies to understand their inefficiencies.

3.1 PC CNN Latency Characterization

Fig. 2 shows the model architectures of two popular PC CNNs used for semantic segmentation tasks – PointNet++ [48] and DGCNN [72]. Primarily, PointNet++ consists of two basic modules, namely, *SetAbstraction* (*SA*) and *FeaturePropagation* (*FP*), with 4 consecutive *SA* modules followed by 4 consecutive *FP* modules. In each *SA* module, the input PC ($N \times C$ matrix, where N is the number of points and C is the feature dimension of each point) is first down-sampled into n points, which can serve as a good coverage of the original input PC. An efficient sampling algorithm for PC CNNs is the farthest point sampling (FPS) [16] , which iteratively selects the farthest point from a set of unsampled points until all the n points are sampled (more details in Sec. 5.1). Next, in the SA module, neighbors for each sampled point are searched. Two commonly-used neighbor

search methods for this step are ball query [45] and k-nearest neighbor (k-NN) [15]. Finally, the features for these sampled points and their neighbors are grouped into a feature matrix (of dimension $n \times S \times C$, where S is the number of neighbors for each sampled point) which is subsequently fed into the feature computation (FC) stage for feature extraction. The *FP* module can be viewed as *reverse-SA*, where the given n points are first up-sampled/interpolated into N points and then convolved with the convolutional kernels in the FC step. Similar to PointNet++, DGCNN consists of 3 successively connected basic modules (*EdgeConv* (*EC*)), as depicted in Fig. 2b. Since the number of points is fixed throughout this network, there is no sampling stage in *EC* (i.e., same number of input points are fed into its subsequent stages).

To better understand the performance implication of these stages, we plot the inference latency of these two CNN models on four datasets using a typical edge SoC (NVIDIA AGX Xavier [43]) in Fig. 3. Specifically, the execution latency is characterized into two main components: sample & neighbor search and feature compute. Overall, across the studied workloads, the sample and neighbor stage can take from 38% to 80% of the end-to-end inference latency. Also, as the number of points increases, these stages take even more time. For example, compared to the ModelNet [66] dataset (with 1024 points/PC), the sample and neighbor search execution latency with the ScanNet [13] dataset (with 8192 points/PC), increases to 80%, making these two stages the primary bottlenecks in the entire inference pipeline. Driven by these observations, we next investigate the reasons behind such inefficiencies, and further explore the potential opportunities for speeding up these stages.

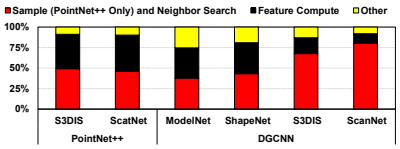


Figure 3: Latency breakdown for PointNet++ [48] and DGCNN [72].

3.2 Reasons for Inefficiencies and Potential Opportunities

As discussed earlier, the sampling and/or neighbor search stages are the main bottlenecks in the PC CNN inference pipeline. However, considering a 2D image as shown in Fig. 4ⓐ-**i**, sampling and neighbor searching can be easily achieved by selecting the pixels with proper indexes with negligible overheads (Fig. 4ⓐ-**ii**). Unfortunately, due to the inherent properties (i.e., unstructuredness and randomness) of PCs, simply performing uniform sampling for PC data will result in loss of information. For example, as shown in Fig. 4ⓑ-**i**, given a PC frame that contains 12 points (i.e., $\{P1, ..., P12\}$), by the uniform sampling approach, the selected points would be $\{P2, P4, ..., P10, P12\}$ (yellow colored points). As, these sampled points cover only half of the input PC, which is not a good representation of the original PC, this would hurt the CNN models' accuracy. Further, as shown in Fig. 4ⓐ-**iii**, for a given pixel (e.g., $p2$) in a 2D image, its neighbors can be found simply via indexing (e.g., $p1, p3, p6$). However, for the point $P2$ in PC (Fig. 4ⓑ-**iii**), simply choosing the points with its nearby indices $\{1, 3, 4\}$ would return 2 "false neighbors".

Hence, due to the unstructured nature of PC data, we cannot directly employ the sample and neighbor search techniques used

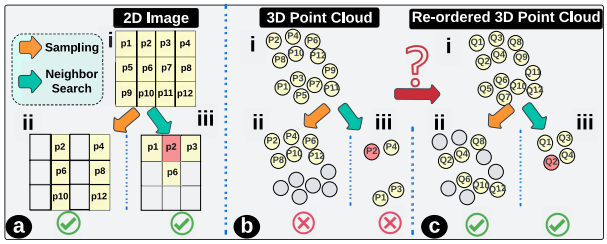


Figure 4: Sample and neighbor search for a 2D image (e.g., pixel p2) in (a); 3D PC (e.g., point P2) via indexing in (b); Re-organized 3D PC (e.g., point Q2) via indexing in (c).

in 2D images (e.g., index-based approaches) for raw PC data. Therefore, 3D PC data uses FPS (for down-sampling), ball query or k-NN (for neighbor search) in order to avoid the loss of important information, at the expense of doubling the execution latency (Fig. 3). However, if we can make the PC data more "structured" (a best case would be similar to a 2D image, i.e., 100% structured), then the 2D image sampling and neighbor search techniques can be applied for 3D PC analysis with negligible overheads (e.g., comparing to FPS, ball query or k-NN, etc.), while maintaining an reasonable accuracy. Fig. 4ⓒ-**i** shows such an example, specifically, after the PCs are rearranged into a more structured shape, simply performing uniform sampling can return a good coverage for the input PCs (Fig. 4ⓒ-**ii**). Similarly, the index-based neighbor search accurately determines the neighbors on this structured PC (Fig. 4ⓒ-**iii**). Compared to the original PC data, this re-organized data is more structured, thus providing the opportunity to utilize the index-based sampling and neighbor search techniques. Note however that, it is *not* trivial to achieve such conversion. Specifically, this conversion technique should meet three requirements: 1) *low complexity* to minimize the overheads; 2) *high parallelism* to fully exploit the existing parallel hardware platforms like GPUs; and 3) *high accuracy* to avoid CNN inference accuracy drop. Fortunately, Morton code, which describes the geometrical relationship between points and can map multi-dimensional data to one dimension while preserving the spatial locality of the data points, is a perfect candidate to achieve such conversion [68].

In fact, Morton code has been applied to optimize the neighbor search for 3D data in prior works or software libraries. For instance, RTNN [76] has proposed to sort the query points spatially by using the Morton codes (i.e., group the points that are spatially close together), in order to reduce the control flow divergence and better utilize the hardware. In comparison, works like [22, 26, 39, 50] have proposed/implemented grid-based solution strategies for neighbor searching and utilize Morton code to reduce the search space by skipping the searching process for the grids which are far away from the query point. We want to emphasize however that these works focus on "non-approximate" neighborhood search and/or have limited applicability scope. For example, [26] only targets at the ball query-like neighbor search. [12] proposes a $(1 + \epsilon)$-approximate nearest neighbor search technique on Morton-sorted points, however, additional computations are required to achieve the specified error bound (i.e., ϵ). On the other hand, as demonstrated later in Sec. 5.2.3, our design is an approximation-based solution with no restrictions on the type of neighbor searcher and even further reduces

search space/computations and can trade off accuracy for better performance/latency. In other words, while these prior efforts aim to enhance the efficiency for neighbor search with negligible to no accuracy loss, our work prioritizes achieving higher performance with acceptable searching errors, which is more applicable/suitable to PC CNN inference in the context of edge devices.

4 STRUCTURIZING POINT CLOUD WITH MORTON CODE

In this section, we first introduce the Morton code and explain how to generate and utilize Morton code for the PC analysis (Sec. 4.1). Next, we demonstrate the effectiveness of applying Morton code for structurizing the PCs using both the qualitative (Sec. 4.2) and quantitative (Sec. 4.3) results.

4.1 What is Morton code?

Morton code (also known as Z-curve [27]), maps data from an n-dimensional space to one dimensional by performing bitwise interleaving on n-d integer coordinates. For example, a point with coordinate $(2,\ 3,\ 4) = (010,\ 011,\ 100)_b$ translates to Morton code $282 = 100, 011, 010_b$ due to bitwise interleaving. Such mapping can preserve the spatial locality of data points and has been applied in many operations like texture mapping[14], N-body problem[52] and matrix multiplication[6]. However, considering our application (PC), where each point is stored as 3 floating-point coordinates, one critical question is, *how to generate Morton code for non-integer data points?* Simply quantizing the coordinates to the nearest integers might result in information loss. For example, all the points within the $[0, 0.5)\times[0, 0.5)\times[0, 0.5)$ bounds would be quantized as $(0, 0, 0)$. To avoid this, we first divide/voxelize the PC data space (the cuboid of dimension $L\times W\times H$) into several smaller cubes (SC)/voxels of dimension $r\times r\times r$, where $r<\min(L, W, H)$ is the predefined grid_size. Next, each SC/voxel can be indexed by 3 integers – (i, j, k) where $0\leq i<L/r$, $0\leq j<W/r$ and $0\leq k<H/r$. Given a point, its coordinates can be voxelized into 3 integers depending on which SC/voxel it belongs to, and then the Morton code can be generated by bitwise interleaving. Such transformation from 3-D to 1-D space (3 x, y, z-indexes to 1 Morton code) can simplify the subsequent computations. Due to the spatial locality preservation, the points nearby in space will have similar Morton codes. Thus, to "structurize" the input PC, we *re-order* the points as their Morton codes. Especially, if the original indexes for input points are $I=\{0,..., N-1\}$ where N is the number of points, after sorting, the new indexes will become $I'=\{i_0,..., i_{N-1}\}$, where i_0 and i_{N-1} are the indexes of the points with minimum and maximum Morton code values, respectively. The re-arranged points are more structured (e.g., like the pixels in 2D image), and thus, can enable index-based sampling and neighbor searching.

4.2 Sampling Structurized Point Cloud

To demonstrate the effectiveness of using "structurized" PC for sampling, in Fig. 5, we plot the (down-)sampling results on different PC data (raw PC [58] and "structurized" PC) using two sampling approaches: farthest point sampling (FPS) and uniform sampling. In general, both the FPS on raw data (Fig. 5ⓐ) and the uniform sampling on "structurized" PC (Fig. 5ⓒ) provide a good coverage for input PC (the sampled points are uniformly distributed across the PC model). While the distribution of the uniformly sampled points on raw PC (Fig. 5ⓑ) is either too dense (almost become a continuous line) or too sparse (very few points in certain regions). This "uneven distribution" of points becomes more apparent on further zooming into the PC. On the other hand, although FPS can achieve very good sampling quality, its overhead is too high. Based on our profiling on an edge device (Nvidia AGX Xavier board), sampling 1024 points from the Bunny model [58] (which contains 40256 points) with FPS takes $\approx 81.7ms$, while the uniform sampling consumes only $\approx 1ms$. Clearly, such performance gap between FPS and uniform sampling is expected to further widen when employing larger models (with more points). Fortunately, with the Morton code serving as the bridge to "structurize" the PC data, we can *directly* perform uniform sampling on the "re-ordered" PC data with minimal sampling overhead, while still maintaining a very good quality (similar to the FPS sampling results), as depicted in Fig. 5ⓒ.

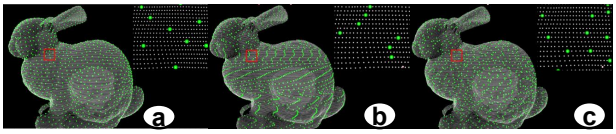


Figure 5: Sampled Bunny [58] model via (a) FPS on raw PC; (b) uniform sampling on raw PC; and (c) uniform sampling on sorted/structurized PC data with Morton code.

4.3 Neighbor Search for Structurized Point Cloud

Apart from sampling, the other time-consuming stage in PC CNN inference pipeline is the neighbor search stage. This is mainly because of the randomness of PC data, due to which we have to iterate through the whole point set when searching the neighbors for any given point. In this section, we discuss the potential opportunities of "skipping" the neighbor search stage by utilizing the "structured" PC data. For example, given the re-arranged points as well as the new indexes ($I' = \{i_0, ..., i_{N-1}\}$ as discussed in Sec. 4.1), we do not need to use complex state-of-the-art (SOTA) neighbor search algorithms (e.g., ball query or k-nearest neighbor (k-NN) which have been widely used in current SOTA PC CNNs). Instead, we can *bypass* the search process by directly selecting k consecutive points near the target data point, where k is the number of neighbors. This means, the k neighbors for the point with index i_p would be the points with indexes $\{i_{p-k/2}, ..., i_p, ..., i_{p+k/2}\}$. By doing so, we can deploy a neighbor search technique similar to the ones used with 2D images (e.g., index-based approach), for 3D PC data as well.

To show the effectiveness of the index-based neighbor searching, Fig. 6 plots the *false* neighbor ratio (defined by the ratio of the "neighbors" picked by our above-mentioned scheme but are not identified as neighbors by the

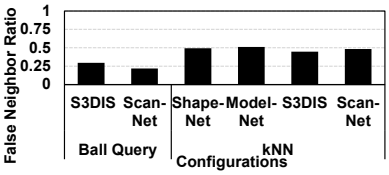


Figure 6: False neighbor ratio on different datasets.

SOTA techniques) , where the x-axis is the configurations (i.e., applying different SOTA neighbor search algorithms on different datasets), while the y-axis shows the false neighbor ratio. As we can observe, the false neighbor ratio can be as low as 23%. Furthermore, if we increase the search window (that is, search k neighbors from $\{i_{p-W/2}, ..., i_p, ..., i_{p+W/2}\}$ instead of directly selecting the $\{i_{p-k/2}, ..., i_p, ..., i_{p+k/2}\}$, where W is the search window size and $W>k$), the false neighbor ratio can be further decreased to 5% (shown later in Sec. 6.3).

Takeaways: While such index-based solutions are efficient and fast, they often lead to suboptimal samples and false neighbors. Therefore, simply utilizing the pre-trained PC CNN models will result in decreased inference accuracy; instead, it is required to include the approximations when retraining the models.

5 MORTON CODE-BASED SAMPLER AND NEIGHBOR SEARCH DESIGN

We observed in Sec. 4 that Morton code can be employed to *structurize* the PC data, thus providing the opportunities to use index-based sampling and neighbor searching (the technique used in 2D image). Thus, unlike prior works [17, 35] which try to optimize the sample and/or neighbor search stages by customized accelerators or by choosing specific data structures (e.g., k-d tree, where the tree construction process itself brings non-negligible overheads), in this work, instead of optimizing these two stages, we choose to "*skip*" them by **approximating** the *complex computations on raw PC data* (performed by the SOTA PC CNNs) with the *simple index-based methods on "structured" PC data.*

5.1 Morton-code-based Sampler

We first present the SOTA down-sampling technique, farthest point sampling (FPS) [16] and its inefficiencies. We then introduce our proposed design utilizing the Morton code to "structurize" the PC data and our design considerations.

5.1.1 Inefficiencies of the SOTA Sampler. To understand the SOTA sampling technique (FPS) for PC data, we illustrate its processing steps in Fig. 7 and Fig. 8(a). Given a PC data containing N points ($P = \{p_1, ..., p_N\}$) and the desired number of sampled points (n) as inputs, the FPS first initializes the un-sampled(S') and sampled(S) sets as input PC and empty set, respectively, while the elements in array D (which stores the distances between un-sampled points and sampled set) are initialized to be inf. Next, the first sampled point is randomly picked (e.g., s_0), and then both the un-sampled set S' and the sampled set S, as well as the distance array D are updated accordingly . For the next $n-1$ points, each time a new point is sampled, all the un-sampled points are traversed and the point with maximum distance to the sampled set is picked . Based on the distance array D, the next sampled point can be decided. Note that each time a new point s_i is sampled, the distance array is updated . The time complexity for such update is $O(N)$, and considering that we need to sample n points in total, the time complexity becomes $O(nN) \approx O(N^2)$ (where $n=O(N)$ in practical scenarios). Moreover, all the points in S are inserted sequentially, meaning that there is limited parallelism in FPS, which further adds to the inefficiencies.

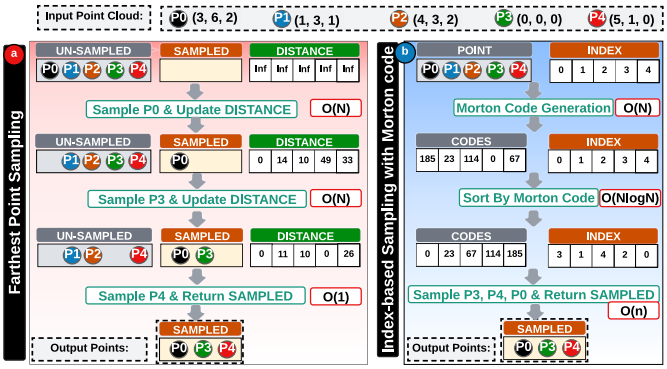


Figure 7: Farthest point sampling (SOTA).

Figure 8: Examples: (a) Farthest point sampling on raw PC data; (b) Index-based sampling on Morton code structured PC data.

Fig. 8(a) shows an example using the FPS to sample 3 out of 5 points. First, the sampled and un-sampled sets and the distance array are initialized. Then, $P0$ is sampled, and thus, the (squared) distance array becomes $\{0, 14, 10, 49, 33\}$. Next, the point $P3$ is sampled as its distance to the sampled set is maximum (49). The new distance array is $\{0, 11, 10, 0, 26\}$ and finally the point $P4$ with maximum distance value (26) is sampled.

Takeaway: Although the FPS can yield a good coverage for the input PC data, its compute complexity is too high: $O(N^2)$ for N input points, where N is in orders of 10^6 considering a PC frame with millions of points. Also, due to the data dependency in the sampled set (S) when sampling a new point, all the points must be sampled one-by-one, which limits the performance boosting scope through parallelism. To improve performance, we explore the opportunities by *directly operating on "structured" PC data* (not considered in prior works), and accelerate both the down-sampling and interpolation/up-sampling stages using approximation opportunities.

5.1.2 Optimizing the Sampling Stage.

Optimizing Down-sampling: Since Morton code can make the PC data more structured, we can perform uniform sampling (with negligible overhead compared to FPS), while still obtain a near-optimal coverage of the input PC data (see Fig. 5). Driven by this observation, we design the Morton code-based PC (down-)sampler as shown in Algo. 1 and Fig. 8(b). Overall, our proposal consists of 3 steps, 1) Morton code generation; 2) Morton code sorting; and 3) uniform sampling of the re-ordered/re-organized points. For the Morton codes generation (shown in Algo. 1), we need two more inputs, i.e., r for the predefined grid_size (discussed in Sec. 4.1) and $\{x_{min}, y_{min}, z_{min}\}$ array for storing the minimum bounds of PC data. The Morton code generation process can run in a fully-parallel manner, where for each point $p_i=(x_i, y_i, z_i)$, we first calculate which voxel it belongs to (line#4 in Algo. 1), then the Morton code can be

obtained by (bitwise) interleaving the indexes for that SC (line#5 in Algo. 1). Due to the high parallelism, generating MC for 8192 points using an edge GPU only takes 0.1ms (0.07% of end-to-end latency). The next step sorts the generated Morton code and outputs the new indexes $I'=[i_0, ..., i_{N-1}]$ in Morton order (line#10 in Algo. 1), where p_{i_0} has minimum the Morton code while $p_{i_{N-1}}$ has the maximum Morton code. The final step uniformly samples the points with the new indexes with a step size of N/n (line#11-13 in Algo. 1). Given a PC frame with N points, the time-complexity of this algorithm is $O(N\log N)$ (due to the sorting stage). Note that, for most cases we have $n{=}O(N) \gg \log N$.

Therefore, with our proposal, we can decrease the compute-complexity from quadratic to logarithmic-time. Further, as observed from line#11 in Algo. 1, all the points can be sampled in parallel, which resolves the data dependency issue observed in the FPS technique, as discussed in Sec. 5.1.1. Similar to Fig. 8(a), in Fig. 8(b), we sample 3 points from the input PC (consisting of 5 points). Specifically, given these 5 input points in this example where $\{x_{min}, y_{min}, z_{min}\}=\{0, 0, 0\}$, with a predefined grid_size r=1, we can obtain the Morton codes ($\{185, 23, 114, 0, 67\}$) using the Morton code generation algorithm described in Algo. 1. Sorting these Morton codes outputs the new index array ($\{3, 1, 4, 2, 0\}$), and we can simply perform uniform sampling and pick points P_3, P_4 and P_0, which are exactly the same points (when using the FPS algorithm). However, if we increase the grid_size r, the sampled results might differ from the baseline (FPS) results. For example, if the grid_size is defined as r=4, then the Morton codes would become $\{2, 0, 1, 0, 1\}$, for which the sorted indexes are $\{1, 3, 2, 4, 0\}$ and finally the sampled points are $\{1, 2, 0\}$. In such cases, simply approximating the FPS results with the results of the Morton code-based sampler might degrade the inference accuracy. Thus, we have to carefully decide the grid_size when using the Morton code-based sampler to achieve a good balance between the inference accuracy and the memory overheads for storing the Morton codes (note that a higher grid_size value means having fewer small cubes, and therefore would need fewer bits to represent the indexes for these SCs; more details in Sec. 5.1.3).

Algorithm 1: Proposed Morton Code-based Sampler

```
Input  : P={p_0, ..., p_{N-1}}:input points; {x_min, y_min, z_min}
Input  : n: number of sampling points; r: preset resolution
Output : S = {s_0, ..., s_{n-1}}: sampled points
1  procedure MC_Gen(P,r,{x_min, y_min, z_min})
2      Init: MC= [0] × N
3      for p_i in P do                          // fully parallel
4          x=(x_i-x_min)/r; y=(y_i-y_min)/r; z=(z_i-z_min)/r
5          MC[i]=bitwise_interleave(x, y, z)
6      return MC
7  procedure MC Sampler(P,n,r,{x_min, y_min, z_min})
8      Init: S=∅
9      MC=MC_Gen(P,r,{x_min, y_min, z_min})
10     [i_0, i_1, ..., i_{N-1}] = merge_sort(MC)
11     for k in [0, ..., n − 1] do              // fully parallel
12         index = k × N/n; S = S ∪ {p_{i_index}}
13     return S
```

Optimizing Up-sampling: As discussed in Sec. 3.1, the interpolation/ up-sampling stage in PC CNN (e.g., PointNet++) can be viewed as the "reverse" of the down-sampling stage. Thus, the Morton code-based down-sampling proposal is also applicable for the up-sampling stage. Specifically, the goal of interpolation stage is to "recover" the feature of the original points ($\{f(p_0), ..., f(p_{N-1})\}$) from the sampled points ($\{f(s_0), ..., f(s_{n-1})\}$). For example, the feature for point p_t can be obtained by $f(p_t){=}g[f(s_i), f(s_j), f(s_k)]$, where s_i, s_j, and s_k are the 3 closest points to p_t in space where $g[]$ is a predefined function like weighted average. As all the points in S are uniformly sampled from the "structured" PC by picking the points $\{p_{i_0}, p_{i_{step}}, p_{i_{2step}}, ..., p_{i_{(n-1)step}}\}$, where $step$ is the step size (N/n). Therefore, for any given point with index i_j (p_{i_j}) where $j \in \{0, ..., N-1\}$, its 4 (approximately) closest points from S should be $\{p_{i_{j'-2step}}, p_{i_{j'-step}}, p_{i_{j'+step}}, p_{i_{j'+2step}}\}$ where $j'{=}j\text{-}j\%step$. With this knowledge, we only need to pick 3 closest points out of these 4 points to perform the interpolation, instead of searching through the entire sampled point set S. Therefore, with our Morton code-based up-sampler, the compute-complexity can be decreased by $n/4 = O(n)$.

Takeaway: Our Morton code-based sampler's output can serve as a good approximation for the sampling results obtained from the SOTA, and thus enables fast and efficient uniform sampling for PC data. However, due to such approximation, the sampled points are *sub-optimal* (especially for a large predefined grid_size value), which may degrade the precision of CNN models. Also, as this work targets edge devices with limited resources, we need to carefully configure our designs such that we can achieve a good balance between *performance improvement*, *memory usage*, and *inference accuracy*.

5.1.3 Design Considerations. In this subsection, we discuss how our design can consider tradeoffs between the performance improvement, inference accuracy and memory overheads.

Performance improvement vs. inference accuracy: As the proposed Morton code-based sampler can boost the PC CNN performance by enabling uniform sampling and the PC CNNs always have multiple sampling layers (recall from Sec. 3.1, PointNet++ [48] consists of 4 down-sampling and 4 up-sampling layers), one may consider to apply such optimization to *all* the sampling layers. However, as observed in Sec. 5.1.2, due to the *sub-optimal* sampled results, the precision of CNN model might be degraded. So, instead of applying our Morton code-based approximation to all the sampling layers, we only optimize the critical sampling layer(s), i.e., the ones contributing most to the execution latency. Next, we use PointNet++ on ScanNet dataset as an example and discuss the details of our design. Fig. 9 (black bars) plots the execution latency for all the 8 sampling layers in PointNet++. As can be observed, the down sampling layer in the first SA module and the up-sampling layer in the

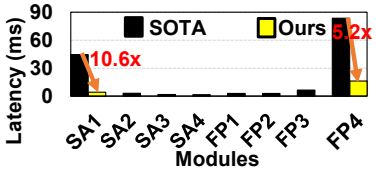


Figure 9: Down-sample (in SA modules) and up-sample (in FP modules) latency in PointNet++(s).

last FP module are the most time-consuming sampling layers. Therefore, we choose to apply the Morton code-based sampling for these two layers and use the the state-of-the-art samplers for the rest sampling layers. As shown in Fig. 9 (yellow bars), we can accelerate the down-sampling layer by 10.6× and the up-sampling/interpolation layer by 5.2×.

Memory overheads vs. inference accuracy: As depicted in Algo. 1, in order to utilize the index-based sampler, first we need to generate the Morton code for each point to structurize the PC data. However, to store these Morton codes, we need to allocate extra space. Considering a PC frame which contains N points, with each point corresponding to a a-bits Morton codes, $Na/8$ Bytes more space will be used. On the other hand, as the PC data contains the 3D coordinate information, these a bits need to be further split into 3 parts ($\lfloor a/3 \rfloor$ bits for each dimension), meaning that the entire space can be divided into $2^{\lfloor a/3 \rfloor} \times 2^{\lfloor a/3 \rfloor} \times 2^{\lfloor a/3 \rfloor}$ small cubes (SCs), which will further translate to a grid_size value of $r{=}D/2^{\lfloor a/3 \rfloor}$, where D is the dimension of the PC's bounding box. Intuitively, the inference accuracy of the CNN models can benefit from a larger a (corresponding to a smaller r), at the expense of higher memory overheads, and vice verse. In this work, we choose a=32 to achieve a good balance between the memory overhead and the inference accuracy degradation.

5.2 Morton Code-based Neighbor Search

In the previous section, we have explored the opportunities of using Morton code to structurize the raw PC data and thus boost the performance for sample stages. Recall from Sec. 4.3 that, the neighbor search stage can also benefit from the "structured" PC data. Before diving into the details of our proposed optimizations, let us first understand the state-of-the-art neighbor search approaches and their inefficiencies.

5.2.1 Inefficiencies of the SOTA Neighbor Search. Ball-Query (BQ) [45] and k-NN [15] are two of the most commonly used neighbor searchers. Specifically, given a point p_i and the number of neighbors to be searched (e.g., k), the BQ algorithm searches through the candidate points set (e.g., $P{=}\{p_0, ..., p_{N-1}\}$) and returns the points inside the ball with center p_i and a predefined radius R. To achieve this, the distance between p_i and any other points ($dist(p_i, p_j)$) for any j in {0, ..., N-1} needs to be computed in $O(N)$ time. Thus, searching the neighbors for all the N points would take $O(N^2)$ time. Fig. 10(a) shows an example using the BQ algorithm to search 3 neighbors for $P2$, which takes the same PC data in Fig. 8 as input, and defines the radius (R) as 11. Following the above-mentioned flow, first, it computes the distance between $P2$ and the other points. Next, $P0$, $P1$, and $P4$ are picked as their distance to $P2$ is less than R. Similarly for k-NN, the first step is also obtaining the distance matrix (with a time complexity of $O(N^2)$[1]; next, the 3 points with green background are selected as they have minimum distances.

Takeaway: As the PC data contains up to millions of points, any algorithm with $O(N^2)$ complexity would be too costly for edge devices. So, an efficient neighbor searcher needs to be designed to reduce the computation complexity, while still maintaining a high parallelism and inference accuracy.

[1]Although prior ball-query or k-NN implementations – like the kd-tree-based approaches, have lower complexity ($O(NlogN)$), both tree construction and traversal have limited parallelism, and hence, are inefficient.

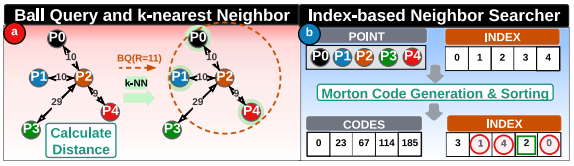


Figure 10: Examples for (a) ball-query and kNN; (b) index-based neighbor search with Morton code structured PC data.

5.2.2 Optimizing the Neighbor Search Stage. Motivated by the index-based neighbor searcher used for 2D images and by the observation that Morton code can be used to structurize the PC data from Sec. 4 and Sec. 5.1.2, in this section, we propose to *approximate* the SOTA neighbor search process by simply selecting/searching the points whose indexes are near the target point (the point for which we search the neighbors) in the "structured" PC data. For example, after we obtain the new index array ($I'{=}\{i_0, ..., i_{N-1}\}$, by sorting the Morton codes, as discussed in Sec. 5.1.2), for any given point p_{i_j}, instead of searching its neighbors from the entire space (which is the case in SOTA works), now we only need to search through the points with indexes of $\{i_{j-W/2}, ..., i_{j+W/2}\}$ ($k{\leq}W{\leq}N$ is a predefined *search window size*), by which the time complexity is reduced from $O(N)$ to $O(W)$. Note that, W is normally set to be much smaller than N, meaning that the performance improvement brought by our proposal is expected to be high. In the example shown in Fig. 10(b), W is defined as k+1 (4=3+1), within which $P1$, $P4$, and $P0$ are selected.

Takeaway: Our index-based neighbor searcher on "structured" PC data can reduce the time complexity of the neighbor search stage by $O(N/W) \approx O(N)$, thus improving the performance. However, similar to the issue of the proposed sampler discussed before, since the the Morton code cannot make the PC data 100% structured, the searched neighbors using this proposal is also *sub-optimal* and might contain "false neighbors". This can potentially lead to quality degradation of CNN models. Therefore, in next section, we explain our design considerations on *how to maximize the performance improvement, while minimizing the impact (caused by the proposed approximations) on CNN models.*

5.2.3 Design Considerations. Similarly, the first question we want to ask here is: *for which layer(s) should we apply our approximations?* Recall that, PointNet++ [48] has 4 neighbor search layers. We plot the performance improvement and the ratio of false neighbors for each layer in Fig. 11 to show the impact of our proposal on these layers. As observed, layer1 has the most significant speedup and the least false neighbor ratio. As a result, it is a

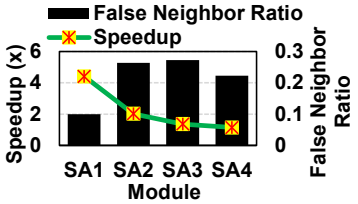


Figure 11: Our neighbor searcher speedup vs. false neighbor ratio for 4 modules in PointNet++(s).

perfect candidate for applying our optimizations. Moreover, since we also apply the Morton code-based approximations for the first sample layer (Sec. 5.1.3), and the neighbor search is right after the sample stage, we can simply reuse the Morton code for our neighbor searcher without any extra overhead. However, for the rest layers, we can only obtain limited performance boost at the expense of much larger false neighbor ratios, which may further result in CNN model degradation. Also, generating the Morton codes for these layers would introduce extra overheads and further decrease the benefits of our proposal.

Therefore, similar to the design for sampler discussed in Sec. 5.1.2, we only apply the optimization for the first neighbor search layer. As opposed to PointNet++ where all the modules use the x, y, z coordinates of input PC for neighbor searching, in DGCNN [72], only first module uses the geometry coordinates for determining the nearest neighbors. For the next several modules, the distance between points are measured using the features (i.e., $dist(p_i, p_j)$=$dist(f_i, f_j)$ where f_i is the feature vector of p_i and always has higher dimensionality (e.g., 64)). Due to this, our Morton code-based optimization can only be applied for the first module as it cannot process higher-dimensional data. For the following neighbor search layers, instead of performing the SOTA computations, we decide to interleave the "reuse" and "compute". For example, with the reuse distance of 1, we reuse the neighbor search results from layer1 for layer2, and employ the SOTA algorithm for layer3. The intuition behind this is that, during the propagation the the CNN model, the neighborhood of points would not vary much across consecutive layers.

Note that, both the reused Morton codes and search results are stored in the GPU memory. As the PC data is processed in batches (shown later in Table. 1), the per-batch size of the Morton code and reused search data are only up to 32KB and 160KB, respectively, leading to reduced memory access overheads. Further, our optimization reduces the neighbor search computations, resulting in fewer GPU memory accesses and also reducing the overall data request stalls by 4-5%.

After deciding the layer where approximation will be applied, the next question is: *how to decide the search window size*, which *is the key parameter to trade off between execution latency and model precision?* In fact, with our proposal, the user can adaptively select proper search window size to accommodate the application requirement. We further investigate how different search window sizes shape the behavior of our approach in a sensitivity study in Sec. 6.3.

5.3 What are the Pros and Cons of Approximation?

As discussed in Sec. 5.1.2 and 5.2.2, compared to the SOTA sampling and neighbor search techniques, our proposal first uses the Morton codes to structurize the unordered PC data, thereby enabling the index-based operations. This is expected to be much faster. In fact, we show later in Sec. 6.2, with our proposal, the sampling and neighbor search can achieve up to 5.2× speedup. However, as discussed earlier, due to the suboptimality of the sampled points and the existence of false neighbors as well as the reuse of neighbor indexes across CNN modules, the inference precision of CNN models is expected to be degraded. To address this issue, we must retrain the CNN models taking into consideration the Morton code-based approximation (discussed further in Sec. 6.2).

5.4 Architectural Insights – Optimizing the Shifted Bottlenecks

Our Morton code-based approximation techniques can accelerate the sample and neighbor search stages by 3.68x(shown in Sec. 6.2). In this section, we explore further opportunities to improve performance/energy efficiency and provide some architectural insights as future research directions.

5.4.1 Increase Tensor Core Utilization for Feature Computation. Recall from the execution latency breakdown in Fig. 3 that, after our optimizations, now the main bottleneck for PC inference shifts to the feature compute (convolution) stage. Fortunately, as recent edge devices are equipped with the tensor cores [44], which are customized for accelerating matrix multiplications, as shown later in Sec. 6.2, by employing the tensor cores, the inference can be further accelerated by 27%. However, we observe that the tensor core utilization is basically *zero* for several layers. This is mainly because the channel dimension of the input matrix is below a certain threshold, and the tensor cores are not invoked. In fact, based on our profiling, with an input matrix of dimension 32×1000×12×32 and the weight matrix of dimension of 12×64×1×1, the convolutional layer takes 40.4ms to execute with no tensor core utilization. However, if we resize the input matrix into 32×100×120×32 shape and convolve it with a 120×64×1×1 dimensional weight matrix, although the compute complexity remains the same, the execution latency reduces to 18.3ms, with a 40% tensor core utilization. Driven by this observation, one optimization direction might be *extending the input feature dimension by gathering the features of several nearby points.* For example, given an input matrix of NxkxC where k is the number of neighbors for each of the N points, and C is the original feature dimension, instead of performing the feature computation for each point P_i, we can first merge the features of several (t) nearby points (like $[f_{it},...,f_{(i+1)t-1}]$, by which the feature dimension is increased to Ct) and then perform the FC. Finally, we can split the convolution result back for these t points (e.g., by averaging). With our Morton code based reordering, the points which are nearby in the input matrix are also close to each other in the space. Therefore, such approximation (merge and split) is not expected to degrade the model quality much. To make this work, the data layout and the approximation techniques have to be carefully designed to avoid any memory access overheads and maintain the CNN model precision.

5.4.2 Decreasing the Data Movement Overheads in Grouping. Another time-consuming operation in the optimized PC CNN pipeline is the *grouping*, which gathers the features of each sampled point and that of their neighbors to form a new feature matrix. For example, given an input feature map of shape NxC and a neighbor index matrix I_M of shape nxk, where N and n are the number of points and the number of sampled points, respectively, while C is feature dimension of each point, after the grouping stage, the dimension of new feature matrix is nkxC. In practice, nk is larger than N (e.g., for PointNet++, nk=8N), assuming that each GPU thread gathers the feature for one index (out of nk in total), there must be some

threads reading *exactly* the same data from memory. Driven by this observation, we want to ask, *can we properly schedule the GPU threads such that there are data sharing/reuse opportunities across them?* In fact, based on our profiling, with simply sorting the index matrix (after which the indexes in each row of I_M is in ascending order), the amount of data transferred/read from L2 cache and system memory can be decreased by 53.9% and 25.7%, respectively. These initial results are encouraging to pursue Morton code-based architectural design space exploration in future.

6 EXPERIMENTAL EVALUATION

We implement and evaluate our proposals against the SOTA PC inference across three metrics critical for PC CNN application: execution latency (for both sampling and neighbor search stages as well as the E2E inference pipeline), energy consumption, and model precision (accuracy). We first describe our experimental platform, PC CNN workloads (the CNN models and the datasets), and the design configurations (Sec. 6.1). Next, we analyze the results (Sec. 6.2) and vary the design points to present the sensitivity study (Sec. 6.3). Finally, we compare our proposal to prior works (Sec. 6.4).

6.1 Methodology

6.1.1 Experimental Platform. We use NVIDIA Jetson AGX Xavier [43], an edge development board (Fig. 12), which consists of a 512-core Volta GPU with 64 Tensor cores, a 8-core ARMv8 64-bit CPU, and 16GB LPDDR4x memory.

6.1.2 Workloads. To demonstrate the effectiveness of our proposals, we use two popular PC CNNs, namely, PointNet++ [48] and DGCNN [72] and four datasets as listed in Table. 1. Specifically, both S3DIS [3] and ScanNet [13] are indoor scene datasets, on which the PointNet++(s) [48] and DGCNN(s) [72] are employed to perform the semantic segmentation. ModelNet40 [66] and ShapeNet [65] are two synthetic datasets. We use DGCNN(c) and DGCNN(p) to perform classification and part segmentation on these two datasets, respectively. All PC frames in these datasets are preprocessed into several mini-batches, with each batch having a fixed number of points, as shown in Table 1.

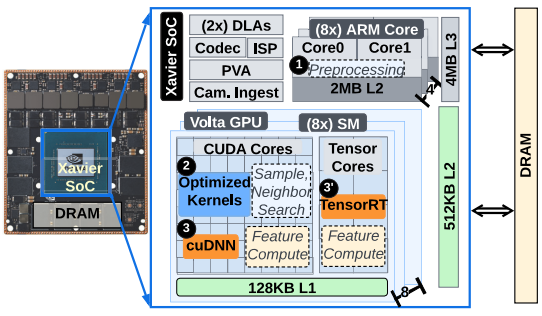


Figure 12: Experimental setup.

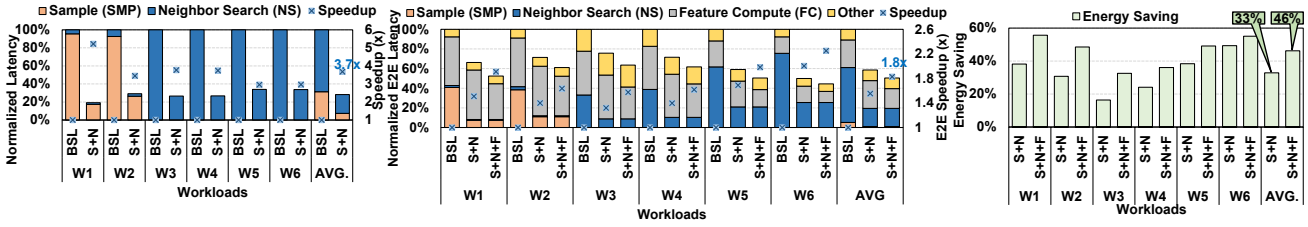
6.1.3 Configurations.
Baseline: We evaluate the baseline inference on the Volta edge GPU and implement the CNN inference using PyTorch [60] with cuDNN (for feature compute stage). The sampling and neighbor search stages are optimized using CUDA kernels.
S+N: We implement the proposed Morton code-based approximation techniques with custom CUDA kernels and apply them to both **S**ample and **N**eighbor search stages (step-❷ in Fig. 12). We use 32 bits for the Morton code because based on our sensitivity study, as the number of bits required to store Morton code increase, the false neighbor percentage (in neighbor search stage) reduces till 32 bits and further increasing the bits does not yield much benefit.
S+N+F: To further boost the end-to-end inference pipeline, we deploy the **F**eature compute (FC) stage to Tensor cores available on the Volta GPU (step-❸' in Fig. 12).

Table 1: Workloads used in this work.

Workload	Model	Dataset	#Points/Batch	Task
W1	PointNet++(s)[48]	S3DIS[3]	8192	Semantic Segmentation
W2		ScanNet[13]	8192	
W3	DGCNN(c)[72]	ModelNet40[66]	1024	Classification
W4	DGCNN(p)[72]	ShapeNet[65]	2048	Part Segmentation
W5	DGCNN(s)[72]	S3DIS[3]	4096	Semantic Segmentation
W6		ScanNet[13]	8192	

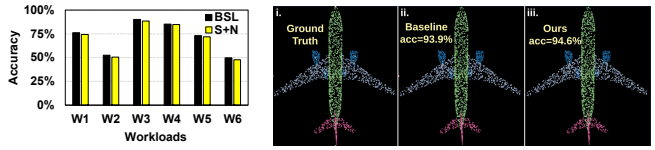
6.2 Performance Results

We present and compare the normalized execution latency (left axis) and speedup (right axis)of our proposals w.r.t. to baseline and energy consumption (via the built-in *tegrastats*[42] tool on the Jetson board) for each workload, as well as the accuracy to evaluate our proposals' effectiveness, and plot the experimental results in Fig. 13 and observe the following:
Execution Latency: Overall, our optimizations can accelerate the sampling (SMP) and neighbor search (NS) stages by 3.7×, on average, w.r.t. the baseline, as shown in Fig. 13a. Across the workloads using the PointNet++ model, W1 can benefit more from our proposals compared to W2 (e.g., 5.21× vs. 3.44× speedup). This is mainly because the inference is performed at batch-level as mentioned in Sec. 6.1.2, and the batch size of W1 is fixed (32) and is usually larger than that of W2 (ranging from 4 to 41 depending on the PC frame, with an average batch size of 14). Recall that, as mentioned in Sec. 5, the baseline sampler and neighbor searcher have quadratic-time compute complexity, as a result, each batch is processed sequentially due to lack of available compute resources on the resource-constrained edge device. As a result, the execution time for the SMP and NS stages increases from about 33ms/batch for ScanNet to 76ms/batch for S3DIS. On the other hand, as our proposal can decrease the computational complexity of the down- and up-sampling stages by $O(N/logN)$ and $O(N)$, respectively, and thus enabling to process multiple batches in parallel, the gap between the execution latency for the SMP and NS stages on ScanNet and S3DIS reduces to only 4.9ms/batch (it takes 9.7ms/batch for ScanNet and 14.6ms/batch for S3DIS), which is the main reason for the performance gains observed in W1 and W2. The speedup for the rest of the workloads with DGCNN are ≈3 or 4. Specifically, for both W3 and W4, our proposal can achieve around 3.7× speedup. Such acceleration comes from two sources: 1) due to the approximation of the SOTA neighbor searcher with our proposal for first



(a) Speedup (Sample and Neighbor Search) (b) Speedup (end-to-end, E2E) (c) Energy Saving

Figure 13: Performance of six workloads w.r.t. the baseline: (a) sample and neighbor search, and (b) E2E speedup; (c) energy saving.



(a) Accuracy comparison (b) Ground truth vs baseline vs ours

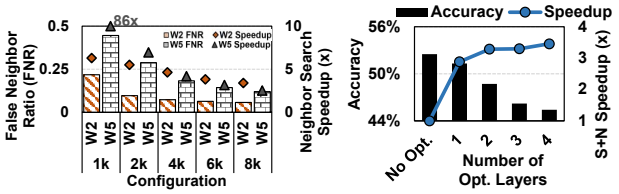
Figure 14: (a). Accuracy; (b). Demo: part segmentation with the baseline inference and our proposal.

EC module (described in Sec. 3), the NS stage can get 29× (W3) and 14.2× (W4) speedup (due to the reduction in time complexity as explained in Sec. 5.2.3); 2) owing to the reuse of the neighbor indexes, the NS computation can be skipped for the second and fourth *EC* modules. A similar trend can be observed in W5 and W6 as well. The performance improvement for the SMP and NS stages can further translate to a 1.55× average speedup in terms of the end-to-end (E2E) latency as shown in Fig. 13b. Moreover, when the tensor cores are employed for accelerating the FC stage, the E2E inference can be further accelerated by up to 2.25× (W6).

Energy Consumption: Furthermore, the computation reduction for the SMP and NS stages when using our Morton code-based schemes can reduce the energy consumption as well. As shown in Fig. 13c, with our design, the energy consumption for inferencing one PC frame is decreased by 33% on average, and 13% more energy can be saved by deploying the feature compute stage (comprising of matrix multiplication operations) into the tensor cores (instead of just using the CUDA cores). For W1 and W2 which employ the PointNet++, our proposal can achieve 38% and 31% energy savings, respectively. Apart from the reduced execution latency as discussed above, another reason for such savings come from a lower power consumption when applying our approximation techniques (e.g., power consumption decreased from 4.5W to 4.2W for W1). For the remaining four workloads, however, the energy savings are slightly lower compared to their corresponding execution latency reduction. For example, the 1.32× speedup observed with W3 (Fig. 13b), only translates to 16% energy savings (Fig. 13c). This is due to our reuse proposal which induces an increased memory pressure when the neighbor index array from the previous *EC* module is cached/stored for later reuse. Based on our measurements, the power consumption of the memory increases from 1.35W for the baseline setting to 1.63W in our proposal.

Accuracy: Due to the suboptimality of our Morton code-based sampler and neighbor searcher, directly using the pretrained CNN models to perform the inference can cause an accuracy drop. So, to maintain the accuracy, we need to retrain the CNN models with our proposed approximations. Fig. 14a shows the accuracy comparison of our retrained models with the baseline. As observed, the accuracy drop is within 2%, thereby having a minimal impact on the inference quality, as shown in Fig. 14b. Moreover, for the accuracy-sensitive applications, we can opt to trade the performance for accuracy. For example, for workload W3, with a larger search window size, our accuracy drop can be as low as 0.7% while simultaneously achieving 4.2× speedup. This reflects the "flexibility" of our proposal in adapting to different application-level requirements and execution environment constraints (accuracy- vs latency-sensitive).

6.3 Sensitivity Study



(a) False neighbor ratio vs search window size vs neighbor search speedup. (b) Accuracy vs number of optimization layers vs speedup.

Figure 15: Sensitivity study.

We study the sensitivity of our proposal by varying the design points discussed in Sec. 5. Fig. 15a shows the tradeoffs between false neighbor ratio (FNR) and speedup for the neighbor search (NS) stage when varying the search window size (where k is the number of neighbors), thus providing insights into efficiently choosing the proper search window size. For example, the accuracy-sensitive applications can use a larger search window for preserving the accuracy, while the high throughput demanding applications can prefer a smaller search window to boost the performance. Another design consideration is the number of layers/modules to apply our MC-based optimization. As shown in Fig. 15b, with only the first SA module (and the corresponding FP module) being optimized, the sampling and neighbor search stages can be sped up

by 2.9× with only 1.2% accuracy drop. However, when we apply the MC-based approximation to more layers, the performance only improves slightly with the significant accuracy drop. This demonstrates the effectiveness of our proposal. To summarize, given new workloads, the developer can first perform the characterization (like the one in Sec. 3) to identify the bottleneck layer(s), for which the MC-based optimizations will be applied and the parameters (e.g., search window size) can be adaptively chosen to accommodate the application's requirement.

6.4 Comparison against Prior Works

We compare our proposal with the prior works from both the quantitative and qualitative perspectives. We first implemented the delay-aggregation (DA) technique in [18] for PointNet++ [48] and tested on S3DIS [3]. The feature compute (FC) stage can be accelerated by 2.1× with DA (88.2ms to 42.2ms per batch). However, as also depicted in [18], since the feature dimension usually increases after convolutional layers, the aggregation/feature grouping stage will become the primary bottleneck. For example, after applying DA, the latency for feature grouping stage increases by 2.73×. Further, as it has no optimization for sampling stage, DA can only achieve 1.12× speedup for the E2E inference latency. PointAcc [35] is another work which optimize the PC CNN by customizing specific accelerators (e.g., mapping unit and memory management unit). Note however that, *our work is orthogonal to PointAcc.* Especially, the key module in the mapping unit in [35] is the distance calculation (with $O(N^2)$ time complexity). With our work, we only need to calculate the Morton codes for each point (with $O(N)$ time complexity), meaning that, by deploying our proposal in the mapping unit in PointAcc [35], the performance of PC CNN inference can be further boosted. Point-X [71] increases the energy efficiency for graph-based PC CNNs by extracting the spatial locality via a SBFS graph traversal algorithm. However, we want to emphasize that, compared to graph traversal, Morton code is a better candidate for capturing the spatial locality of PC data due to its simplicity and has also been proven to be efficient in a recent work [68]. Furthermore, Point-X [71] has very limited application scope and lacks generality as shown in Table. 2 since it is only targets the graph-based PC CNNs. Recently, [17] addresses the irregular memory access issue in neighbor search stage by splitting the k-d tree into top- and bottom-trees. This work, however, overlooks the sampling stage. Finally, all these prior works introduce extra design overheads for hardware customization, while EdgePC favors the commercially available hardwares and can boost the performance without any overheads.

Table 2: Qualitative comparisons.

	Accuracy	Generality	Design Overhead
Crescent [17]	✓	✓	✗
PointAcc [35]	✓	✓	✗
Point-X [71]	✓	✗	✗
EdgePC	✓	✓	✓

7 CONCLUSION

Point Cloud (PC) has recently gained huge attention with the increasing availability of low-cost PC acquisition devices like smartphones with LiDAR cameras. In particular, the availability of specialized accelerators (e.g, NPU, TPU) on smartphones/handhelds have made PC inference on edge devices an attractive option. In 3D PC deep learning analytics, sampling and neighbor search stages contribute to more than 50% of the end-to-end inference latency, thus are the primary bottlenecks in the entire PC inference pipeline. Although few prior works have either designed custom PC accelerators in hardware or proposed software optimizations, they have missed opportunities to optimize these two critical stages, especially considering the unique characteristics of the PC data (irregular and unstructured). In this paper, we present a novel technique to utilize *Morton code* to "structurize" the raw PC data and minimize the sampling and neighbor search latencies by intelligently skipping computations on the structured PC data. Towards this, we design *EdgePC*, which is an edge-friendly framework with two complementary approximation techniques for the sampling and neighbor search stages in order to accelerate the PC inference as well as improve its energy efficiency. Our evaluations of six different PC workloads on an NVIDIA Jetson Xavier edge board demonstrate that the proposed design does not only achieve an average speedup of 3.68× for the sampling and neighbor search stages (which translates to 1.55× speedup of the end-to-end PC inference latency) with minimal accuracy loss, but it also saves 33% of the overall PC inference energy.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback and suggestions towards improving the paper content. This research is supported in part by NSF grants #1931531, #2116962, #2122155, #2211018, #1763681 and #2028929.

REFERENCES

[1] Apple Inc. 2022. iPhone 13 Pro. "https://www.apple.com/am/iphone-13-pro/".
[2] Alan Walford. 2017. What is Photogrammetry? "https://www.photogrammetry.com/".
[3] Iro Armeni, Ozan Sener, Amir R Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, and Silvio Savarese. 2016. 3D Semantic Parsing of Large-Scale Indoor Spaces. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 1534–1543.
[4] Sandeepa Bhuyan, Shulin Zhao, Ziyu Ying, Mahmut T. Kandemir, and Chita R. Das. 2022. End-to-End Characterization of Game Streaming Applications on Mobile Platforms. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1 (2022), 25 pages.
[5] Thomas Blanc, Mohamed El Beheiry, Clément Caporal, Jean-Baptiste Masson, and Bassam Hajj. 2020. Genuage: visualize and analyze multidimensional single-molecule point cloud data in virtual reality. *Nature Methods* 17, 11 (2020), 1100–1102.
[6] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, USA, 233–244.
[7] ByteBridge. 2021. How 3D Point Cloud Annotation Service Fuels the Field of Automatic Driving? "https://medium.com/nerd-for-tech/application-of-3d-point-cloud-in-the-field-of-automatic-driving-723ec9544a6c".
[8] Jingdao Chen, Zsolt Kira, and Yong K Cho. 2019. Deep learning approach to point cloud scene understanding for automated scan to 3D reconstruction. *Journal of Computing in Civil Engineering* 33, 4 (2019), 04019027.
[9] Siheng Chen, Baoan Liu, Chen Feng, Carlos Vallespi-Gonzalez, and Carl Wellington. 2020. 3D Point Cloud Processing and Learning for Autonomous Driving: Impacting Map Creation, Localization, and Perception. *IEEE Audio and Electroacoustics Newsletter* 38, 1 (2020), 68–86.
[10] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. 2017. Multi-view 3D Object Detection Network for Autonomous Driving. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 6526–6534.
[11] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *2019*

IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). IEEE Computer Society, Los Alamitos, CA, USA, 3070–3079.

[12] Michael F Connor. 2007. *Simple, Thread-Safe, Approximate Nearest Neighbor Algorithm*. Master's thesis.

[13] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. 2017. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2432–2443.

[14] DISCOVER three.js. 2022. A Brief Introduction to Texture Mapping. "https://discoverthreejs.com/book/first-steps/textures-intro/".

[15] Richard O Duda, Peter E Hart, and David G Stork. 1973. *Pattern classification and scene analysis*. Wiley New York.

[16] Yuval Eldar, Michael Lindenbaum, Moshe Porat, and Yehoshua Y Zeevi. 1997. The farthest point strategy for progressive image sampling. *IEEE Transactions on Image Processing* 6, 9 (1997), 1305–1315.

[17] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022. Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 962–977.

[18] Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. 2020. Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 1037–1050.

[19] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, USA, 3354–3361.

[20] Silvio Giancola, Jesus Zarzar, and Bernard Ghanem. 2019. Leveraging Shape Completion for 3D Siamese Tracking. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 1359–1368.

[21] GlobeNewswire, Inc. 2022. Straits Research. "https://www.globenewswire.com/en/news-release/2022/09/14/2516327/0/en/LIDAR-Market-Size-is-projected-to-reach-USD-6-93-Billion-by-2030-growing-at-a-CAGR-of-19-27-Straits-Research.html".

[22] Julian Gross, Marcel Köster, and Antonio Krüger. 2019. Fast and Efficient Nearest Neighbor Search for Particle Simulations.. In *CGVC*. 55–63.

[23] Taisuke Hashimoto and Masaki Saito. 2019. Normal Estimation for Accurate 3D Mesh Reconstruction with Point Cloud Model Incorporating Spatial Structure. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 54–63.

[24] Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. 2018. RAPIDNN: In-Memory Deep Neural Network Acceleration Framework. *CoRR* abs/1806.05794 (2018).

[25] Intel Corporation. 2022. Intel RealSense Depth and Tracking cameras . "https://www.intelrealsense.com/".

[26] Jan Bender, Weiler Marcel and Stephan Seitz. 2022. cuNSearch. https://github.com/InteractiveComputerGraphics/cuNSearch.

[27] Jeroen Baert. 2013. Morton encoding/decoding through bit interleaving: Implementations. "https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/".

[28] Artem Komarichev, Zichun Zhong, and Jing Hua. 2019. A-CNN: Annularly Convolutional Neural Networks on Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 7413–7422.

[29] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. 2017. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. Association for Computing Machinery, New York, NY, USA, 409–421.

[30] Shiyi Lan, Ruichi Yu, Gang Yu, and Larry S Davis. 2019. Modeling Local Geometric Structure of 3D Point Clouds Using Geo-CNN. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 998–1008.

[31] Loic Landrieu and Martin Simonovsky. 2018. Large-Scale Point Cloud Semantic Segmentation with Superpoint Graphs. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 4558–4567.

[32] Kyungjin Lee, Juheon Yi, Youngki Lee, Sunghyun Choi, and Young Min Kim. 2020. GROOT: A Real-Time Streaming System of High-Fidelity Volumetric Videos. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. Association for Computing Machinery, New York, NY, USA, 14 pages.

[33] Bo Li. 2017. 3D Fully Convolutional Network for Vehicle Detection in Point Cloud. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE Press, 1513–1518.

[34] Chen-Hsuan Lin, Chen Kong, and Simon Lucey. 2018. Learning Efficient Point Cloud Generation for Dense 3D Object Reconstruction. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 8 pages.

[35] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. 2021. PointAcc: Efficient Point Cloud Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 449–461.

[36] Xingyu Liu, Charles R Qi, and Leonidas J Guibas. 2019. FlowNet3D: Learning Scene Flow in 3D Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 529–537.

[37] Xiangjun Liu, Wenfeng Zheng, Yuanyuan Mou, Yulin Li, and Lirong Yin. 2021. Microscopic 3D reconstruction based on point cloud data generated using defocused images. *Measurement and Control* 54, 9-10 (2021), 1309–1318.

[38] Zhi Liu, Qiyue Li, Xianfu Chen, Celimuge Wu, Susumu Ishihara, Jie Li, and Yusheng Ji. 2021. Point cloud video streaming: Challenges and solutions. *IEEE Network* 35, 5 (2021), 202–209.

[39] Lixin Xue and Oliver Batchelor. 2022. FRNN. https://github.com/lxxue/FRNN.

[40] Rufael Mekuria, Kees Blom, and Pablo Cesar. 2016. Design, implementation, and evaluation of a point cloud codec for tele-immersive video. *IEEE Transactions on Circuits and Systems for Video Technology* 27, 4 (2016), 828–842.

[41] Microsoft. 2022. Kinect for Windows. "https://learn.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows".

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

[43] NVIDIA Corporation. 2022. Jetson AGX Xavier Series. "https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/".

[44] NVIDIA Corporation. 2022. NVIDIA Tensor Cores. "https://www.nvidia.com/en-us/data-center/tensor-cores/".

[45] Stephen M Omohundro. 1989. *Five balltree construction algorithms*. International Computer Science Institute Berkeley.

[46] Charles R. Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J. Guibas. 2018. Frustum PointNets for 3D Object Detection from RGB-D Data. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 918–927.

[47] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 77–85.

[48] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. 2017. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 5105–5114.

[49] Mayank Raj. 2020. Point Clouds and it's significance in AR! "https://medium.com/arway/point-clouds-and-its-significance-in-ar-155db2673865".

[50] Rama C. Hoetzlein. 2014. Fast Fixed-Radius Nearest Neighbor Search on the GPU. https://tinyurl.com/4rcjdu7p.

[51] Joshua Romphf, Elias Neuman-Donihue, Gregory Heyworth, and Yuhao Zhu. 2021. Resurrect3D: An Open and Customizable Platform for Visualizing and Analyzing Cultural Heritage Artifacts. In *The 26th International Conference on 3D Web Technology*. Association for Computing Machinery, New York, NY, USA, 7 pages.

[52] Ari Rubinsztejn. 2018. What Is The N-body Problem? "https://gereshes.com/2018/05/07/what-is-the-n-body-problem/".

[53] SAMSUNG. 2022. Specifications | Galaxy S20, S20+ and S20 Ultra - Samsung. "https://www.samsung.com/levant/smartphones/galaxy-s20/specs/".

[54] Anup Sarma, Sonali Singh, Huaipan Jiang, Ashutosh Pattnaik, Asit K. Mishra, Vijaykrishnan Narayanan, Mahmut T. Kandemir, and Chita R. Das. 2021. Exploiting Activation based Gradient Output Sparsity to Accelerate Backpropagation in CNNs. *CoRR* abs/2109.07710 (2021).

[55] John Schulman, Alex Lee, Jonathan Ho, and Pieter Abbeel. 2013. Tracking deformable objects with point clouds. In *2013 IEEE International Conference on Robotics and Automation*. IEEE, 1130–1137.

[56] Martin Simon, Karl Amende, Andrea Kraus, Jens Honer, Timo Samann, Hauke Kaulbersch, Stefan Milz, and Horst Michael Gross. 2019. Complexer-YOLO: Real-Time 3D Object Detection and Tracking on Semantic Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE Computer Society, Los Alamitos, CA, USA, 1190–1199.

[57] Sonali Singh, Anup Sarma, Sen Lu, Abhronil Sengupta, Mahmut T. Kandemir, Emre Neftci, Vijaykrishnan Narayanan, and Chita R. Das. 2022. Skipper: Enabling efficient SNN training through activation-checkpointing and time-skipping. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 565–581.

[58] Stanford University Computer Graphics Laboratory. 1994. The Stanford Models. "http://graphics.stanford.edu/data/3Dscanrep/".

[59] Jiaming Sun, Yiming Xie, Siyu Zhang, Guofeng Zhang, Hujun Bao, and Xiaowei Zhou. 2021. You Don't Only Look Once: Constructing Spatial-Temporal Memory for Integrated 3D Object Detection and Tracking. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Los Alamitos, CA, USA, 3265–3174.

[60] The PyTorch Foundation. 2022. An open source machine learning framework that accelerates the path from research prototyping to production deployment. "https://pytorch.org/".

[61] TruePoint Laser Scanning, LLC. 2022. What Is 3D Laser Scanning? "https://www.truepointscanning.com/what-is-3d-laser-scanning".

[62] VREX. 2022. How to Bring Point Clouds into VR. "https://www.vrex.no/blog/point-cloud-vr/".

[63] Xiaogang Wang, Yuelang Xu, Kai Xu, Andrea Tagliasacchi, Bin Zhou, Ali Mahdavi-Amiri, and Hao Zhang. 2020. Pie-net: Parametric inference of point cloud edges. *Advances in neural information processing systems* 33 (2020), 20167–20178.

[64] Wenxuan Wu, Zhongang Qi, and Li Fuxin. 2019. PointConv: Deep Convolutional Networks on 3D Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 9613–9622.

[65] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 2015. 3D ShapeNets: A deep representation for volumetric shapes. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 1912–1920.

[66] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 2022. Princeton Model Net. "https://modelnet.cs.princeton.edu/".

[67] Xu Yan, Chaoda Zheng, Zhen Li, Sheng Wang, and Shuguang Cui. 2020. PointASNL: Robust Point Clouds Processing Using Nonlocal Neural Networks With Adaptive Sampling. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 5588–5597.

[68] Ziyu Ying, Shulin Zhao, Sandeepa Bhuyan, Cyan Subhra Mishra, Mahmut T. Kandemir, and Chita R. Das. 2022. Pushing Point Cloud Compression to the Edge. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 282–299.

[69] Ziyu Ying, Shulin Zhao, Haibo Zhang, Cyan Subhra Mishra, Sandeepa Bhuyan, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2022. Exploiting Frame Similarity for Efficient Inference on Edge Devices. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 1073–1084.

[70] Haoxuan You, Yifan Feng, Rongrong Ji, and Yue Gao. 2018. PVNet: A Joint Convolutional Network of Point Cloud and Multi-View for 3D Shape Recognition. In *Proceedings of the 26th ACM International Conference on Multimedia*. Association for Computing Machinery, New York, NY, USA, 1310–1318.

[71] Jie-Fang Zhang and Zhengya Zhang. 2021. Point-X: A Spatial-Locality-Aware Architecture for Energy-Efficient Graph-Based Point-Cloud Deep Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 1078–1090.

[72] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 8 pages.

[73] Shulin Zhao, Haibo Zhang, Sandeepa Bhuyan, Cyan Subhra Mishra, Ziyu Ying, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2020. DéJà View: Spatio-Temporal Compute Reuse for Energy-Efficient 360° VR Video Streaming. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE Press, 241–253.

[74] Shulin Zhao, Haibo Zhang, Cyan Subhra Mishra, Sandeepa Bhuyan, Ziyu Ying, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita Das. 2021. HoloAR: On-the-Fly Optimization of 3D Holographic Processing for Augmented Reality. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 494–506.

[75] Yin Zhou and Oncel Tuzel. 2018. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 4490–4499.

[76] Yuhao Zhu. 2022. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 76–89.