Compressed Geometric Arrays for Point Cloud Processing

Hoda Roodaki and Mahdi Nazm Bojnordi

Abstract—The ever-increasing demand for immersive applications has made point cloud an important data type for 3D processing. Tree-based data structures are commonly used for representing point clouds where memory pointers are used to realize the connection among points. The significant cost of data storage and irregular access patterns for processing points make such data structures largely inefficient. In this paper, we examine a point cloud representation using compressed geometric arrays (CGA) that reduces the size of point cloud and limits the amount of memory indirection. Our experimental results on a set of critical point cloud operations indicate 998× speed-up, 410× better bandwidth utilization, and 58% storage reduction for CGA over the state-of-the-art point cloud library (PCL).

Index Terms—Point cloud, data representation, memory bandwidth.

I. INTRODUCTION

POINT cloud refers to a set of geometric points generated by computer graphics or acquired by light detection and ranging (LiDAR) scanners to represent a three-dimensional (3D) space. Recent applications of point cloud, such as medical imaging, architecture, 3D printing, manufacturing, 3D gaming, and virtual reality (VR) need gigabytes of memory for representing moderately sized objects. Therefore, memory bandwidth and capacity are keys to high performance point cloud processing. In this article, we extend our previous work on efficient point cloud representation [3] to improve the bandwidth efficiency and performance of point cloud processing based on compressed geometric arrays (CGA).

Point clouds are traditionally represented by 1) linear list of points and 2) tree-based data structures. The linear list is simple and well-suited for processing small point cloud data. In contrast, tree structures are preferred over lists for representing large-scale point clouds. The most popular tree structures are octree and kd-tree. In an octree, each internal node has exactly eight children representing the eight octants of a specific subspace. The octants are defined regardless of object geometry and point density in space, which allows for easy addition and removal of points. However, an octree may become significantly inefficient for representing scenes with large void areas and sparse objects. Instead, a kd-tree divides space into binary sub-spaces at its internal nodes according to the density of points along the x, y, and z coordinates. Therefore, kd-trees are more efficient than octrees for representing void areas, but they suffer from a significant cost of restructured tree nodes at point addition and removal. To address these challenges for point cloud processing, CGA employs arrays of pointers and coordinates that preserve the geometry of objects in a fast accessible data structure while keeping data overhead low. CGA limits memory indirection during point operations (e.g., lookup, removal, and addition) to improve bandwidth. Moreover, CGA reduces the size of point cloud data through avoiding memory allocation for duplicated point coordinates.

Here, we provide an introduction to various point operations such as point cloud merge, projection, nearest neighbor (NN) search, and point cloud compression. Next, we investigate the challenges and opportunities of using octree in point cloud operations. We then explain the CGA representation format and its application to the point operations. Finally, we perform a set of experiments on various point clouds generated by LiDAR and computer graphics tools. The main contributions of this paper are as follows.

- Investigate the challenges and opportunities of using octree for point cloud operations.
- Examine CGA for performing various point cloud operations, including merge, 3D to 2D projection, NN search, and compression.

As compared to the PCL library, CGA achieves 998× speed-up, 410× better bandwidth utilization, and 58% storage reduction for the merge, projection, and NN search operations.

II. BACKGROUND AND RELATED WORK

This section provides the background knowledge on point cloud representation and operations as well as a review of the related work in this area.

A. Point Cloud Representation

Numerous data structures have been proposed in the literature to store point cloud data. In its simplest form, a one-dimensional (1D) array of unordered points may be used for storing point clouds. Such data structure suffers from costly linear scan for

Manuscript received 11 May 2022; revised 20 August 2022 and 28 November 2022; accepted 23 December 2022. Date of publication 30 December 2022; date of current version 8 December 2023. The Associate Editor coordinating the review of this manuscript and approving it for publication was Dr. Yong Luo. (Corresponding author: Hoda Roodaki.)

Hoda Roodaki is with the Department of Computer Engineering, K. N. Toosi University of Technology, Tehran 16315-1355, Iran, and also with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran 19395-5746, Iran (e-mail: hroodaki@kntu.ac.ir).

Mahdi Nazm Bojnordi is with the School of Computing, University of Utah, Salt Lake City, UT 84112 USA (e-mail: bojnordi@cs.utah.edu).

Digital Object Identifier 10.1109/TMM.2022.3233256

1520-9210 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
point lookup in large-scale point clouds [7]. Voxel hashing is a mechanism that stores points in a hash table using unique keys, where each key is translated to a memory index by a hash function [8]. Windowed priority queue enables fast point data retrieval by holding references to the first dimension of each point in a queue [4].

Octree is a popular point cloud format that employs a tree structure to represent volumes of rectangular cuboid [1], [23]. In its simplest form, each internal tree node includes eight pointers to eight octants of the 3D space. Leaf nodes do not have pointers to children. Additional pointers may be considered at child nodes to parents for fast bottom-up traversals, which further increase the memory size. To save memory, it is possible for each node to store only one pointer to an array of eight children, instead of eight individual pointers. This, however, makes point addition/deletion challenging. To enable on-demand memory allocation, one may replace the array of children with a linked list; therefore, each node would need one pointer for a sibling and another for its first child. The new tree, however, requires more memory pointers and metadata. Prior work [6] proposes a serialized pointer-free octree to alleviate the high bandwidth and storage costs of pointers. Such data structure, however, requires an \( O(n) \) access time, where \( n \) is the number of points. An integral octree is another variation of octrees comprising a binary tree and an integral 1D array that stores the pre-computed sums of tree nodes. The state-of-the-art PCL library employs a more efficient tree structure, called kd-tree, that organizes points in a k-dimensional space [5]. Regrettably, the existing point cloud representations suffer from 1) time-consuming linear scans per operation, 2) large volume stored and transferred data, and 3) significant bandwidth overhead caused by tree node links. The proposed CGA representation aims to address these challenges through interconnecting arrays of compressed coordinates (Section V).

B. Point Cloud Processing

Point cloud processing includes a set of tasks related to 1) acquiring points at a source using sensors or computer graphics, 2) transferring points to a destination, 3) storing the point data in memory, and 4) computing them in the processor cores. Most of these tasks are extended from the conventional signal and image processing [9]. For example, point cloud compression aims to reduce the storage and transmission costs of point data. Complete compression frameworks for 3D dynamic point clouds have been proposed by the prior work [16], [17]. In recent years, deep learning has enabled novel methods of point cloud processing, including shape classification, object detection, 3D segmentation, object tracking, and point cloud compression [10], [11], [12], [13]. All these tasks require a point cloud representation for performing point operations. Therefore, the proposed CGA format has the potential to enhance a wide range of point cloud processing tasks.

C. Point Cloud Operations

The key to efficient point cloud processing is to make the basic point operations efficient.

a) Merge: Merging point clouds is necessary when several views of an object or scene are collected from different angles. A pre-processing step may be necessary to ensure a common coordinate system for all views and resolve their overlapping points. In a merge operation, we consider one point cloud as a base that dominates the attribute of points in overlapping regions.

b) Projection: A 3D to 2D projection is a mapping that can be orthogonal or perspective. An orthogonal projection represents a 3D object with three or more 2D views along parallel lines that are perpendicular to each view plane. In contrast, a perspective projection linearly maps the 3D objects onto a 2D view such that distant objects appear smaller than the nearer ones [21].

c) NN Search: Given a query point \( s_0 \), an NN search refers to selecting a point from a 3D space (S) such that it is closer to \( s_0 \) than the rest of the points in S. Most NN methods are based on recursive subdivisions of the 3D space [22].

d) Compression: Point cloud compression employs an internal tree for processing points. In an example point cloud compressor [17], an octree of space is first created using a recursive mechanism. In addition to the geometry information of points, an octree needs to store the color attributes of points. Once the octree is formed internally, temporal and spatial prediction algorithms are used to compress the point cloud data. These algorithms heavily rely on some basic point operations such as lookup, NN search, and addition.

III. PERFORMANCE ANALYSIS OF TREE-BASED REPRESENTATIONS

In this section, we analyze the state-of-the-art tree-based formats in terms of computational complexity, memory bandwidth requirements, and the volume of processed data. The setup for this analysis is explained in Section VI. We measure the execution time, the amount of transferred data to/from memory, and a proxy for memory bandwidth across a set of mixed point clouds generated by LiDAR and computer graphics. All the performance metrics are evaluated for the previously mentioned point operations. As shown in Fig. 1, the x-axis of each plot represents a range of sequence sizes for which the basic operations are evaluated. Despite the increasing amounts of processed data, the memory bandwidths for merge, projection, and compression degrade for larger inputs. This anomaly is due to the lack of scalability in tree structures for representing large point clouds.

IV. CGA: COMPRESSED GEOMETRIC ARRAYS

In this section, we explain the proposed CGA representation, which is an extension of our previous work on geometric arrays (G-Arrays) for point cloud processing [3]. Fig. 2 represents an example point cloud with five points. Regardless of the number of points, CGA employs six arrays to store every point cloud. \( RGB_{value} \) is an array of RGB components for points. \( Z_{index} \) is an array of z coordinates, and \( Y_{index} \) is an array of y coordinates. \( Y_{pointer} \) is a vector storing the cumulative number of existing points. The vector values are defined as \( Y_{pointer}[i + 1] = Y_{pointer}[i] + N_y \), where \( N_y \) represents the


number of points that have a certain $x$ value and $y = Y_{index}[i]$. Similarly, $X_{index}$ is an array of $x$ coordinates, and $X_{pointer}$ is a vector representing the cumulative number of points as $X_{pointer}[i + 1] = X_{pointer}[i] + N_X$, where $N_X$ is the number of points with $x = X_{index}[i]$.

Please note that duplicated coordinates are not stored in the CGA arrays; therefore, it requires less memory storage than the existing tree-based and list data structures. The main structural difference between CGA and our prior work [3] is the addition of $X_{index}$ to CGA. G-Arrays consider the $x$ coordinates of points to be used as indices of $X_{pointer}$, which forces the $x$ coordinates to be integer values only. Moreover, every element of $X_{pointer}$ must be allocated in the G-Arrays even if no corresponding point is assigned to its index. This limits the efficiency of G-Arrays.

V. PERFORMING POINT OPERATIONS WITH CGA

We further extend our prior work [3] via examining new point operations—e.g., point cloud merge, 3D to 2D projection, nearest neighbor (NN) search, and point cloud compression.

A. NN Search

We develop a CGA-based NN search algorithm that finds the nearest point to a given query $(x_Q, y_Q, z_Q)$ in two steps.

1) Step 1: We look for a point that exactly matches the query coordinates in $X_{index}, Y_{index},$ and $Z_{index}$. This coordinate lookup is performed sequentially, and a miss in any of the arrays results in a lookup miss. If $X_{index}[i] = x_Q, X_{pointer}[i]$ and $X_{pointer}[i + 1]$ determine the range of candidates for matching $y$ coordinates in $Y_{index}$. Among those candidates, if $Y_{index}[j] = y_Q, Y_{pointer}[j]$ and $Y_{pointer}[j + 1]$ determine the range of candidates for matching $z$ coordinates in $Z_{index}$. The search outcome will then be a hit if $Z_{index}[k] = z_Q$. Moreover, $RGB\text{value}[k]$ shows the color attributes of the matching point.

2) Step 2: If no matching point is found, we set a base point $(x_B, y_B, z_B)$ for completing the nearest neighbor search. Where, $x_B = X_{index}[i]$ such that $|x_B - x_Q|$ has the minimum value; $y_B = Y_{index}[j]$ such that $X_{pointer}[j] \leq j < X_{pointer}[j + 1]$ and $|y_B - y_Q|$ has the minimum value; and $z_B = Z_{index}[k]$ such that $Y_{pointer}[k] \leq k < Y_{pointer}[k + 1]$ and $|z_B - z_Q|$ has the minimum value. We compute the Euclidean distance between the query and base points as $\epsilon$. Then, we identify a set of candidate points as $\{p\} | |(x_B - x_Q) + |y_B - y_Q| + |z_B - z_Q| < \epsilon\}$. Finally, we compute the Euclidean distances between the query and candidates to find the nearest point.

B. Merge

Consider merging two point clouds stored in separate CGA instances (Fig. 3). We first create $\Sigma$, a blank CGA instance to which the points from the original CGAs are added. To access a point $s_0 = (s_0^x, s_0^y, s_0^z)$ in each CGA, we need to perform a point lookup as explained in the first step of NN search (Section V-A). We start scanning the index arrays of the original CGA instances using dedicated pointers, simultaneously. The goal is to copy points from a CGA with smaller coordinate values to $\Sigma$ and advance its pointer until all input points are visited.

C. Projection

We design an algorithm for projection from the 3D point cloud space to 2D. The algorithm is implemented for both PCL and CGA. Generally, a transformation matrix is applied to all the points of a point cloud for an orthogonal or perspective projection. As a result, a linear scan of the entire point cloud is
necessary to perform a projection. Scanning a tree-based data structure requires a tree traversal through memory pointers that is more costly than scanning the linear arrays of CGA.

D. Compression

Fig. 4 shows the proposed mechanism for point cloud compression using CGA. We consider both spatial and temporal techniques to achieve a superior compression performance. The first frame of each input sequence is considered as a reference frame and is spatially encoded using the MPEG G-PCC [30]. All of the following frames are encoded/predicated according to the reference frame. The rest of this section describes the main steps of our proposed data-flow for point cloud compression.

1) Clustering Points: As the first step, we use point clustering to compute motion vectors during the temporal prediction of frames. Theoretically, the goal is to find the most similar point of the reference frame to each point of the predicted frame. A vector representing the geometric difference between the two points is a motion vector. However, computing a vector per each point results in excessive time and memory costs. Instead, we propose to compute one vector for every group of neighboring points—a.k.a., cluster. We employ the k-means algorithm for clustering the points of reference and predicted frames. Notice that the number of clusters determines the amount of distance computation and the quality of clustering. To limit the execution time, we limit the number of clusters per frame to eight. The initial centroids of the k-means algorithm are set to the corners of each point cloud frame based on its range of coordinate values. Moreover, we limit the number of k-means iterations to 40.

2) Estimating Cluster Motion: In temporal prediction, estimating the movement of textures or objects among consecutive frames is called motion estimation. Typically, encoding algorithms use a vector to represent a motion between blocks of pixels or points. A search algorithm is necessary to find the matching blocks between the predicted and reference frames. Our proposed mechanism employs a two-level algorithm for point motion estimation. First, a coarse-grained motion vector (CMV) is computed for each cluster between the reference and predicted frames. Note that a cluster may include multiple blocks of points. We then use the CMV of every predicted cluster to compute the counterparts of predicted blocks in the reference frame. Next, we re-evaluate the similarity of each predicted block to its reference counterpart and a few more neighbors for fine-tuning the estimation. As a result, each predicted block defines a small delta vector added to CMV for better motion estimation. This way, we introduce a two-level motion estimation mechanism for point cloud compression.

3) Generating Point Residual: After computing motion vectors, a reconstructed frame is generated using the predicted frame and the estimated motion vectors. The points of each cluster are displaced according to their corresponding motion vector. The reconstructed error frame, often referred to as the residual frame, contains the information to correct the error of motion estimation process. We compute the difference between the attributes of the corresponding points in the reference and reconstructed frames. Here, the main issue is finding a corresponding point in the reference frame for each point of the reconstructed block. Unlike the computer-generated data or 2D videos, most point clouds (in particular, those generated by LiDAR scanners) exhibit a limited spatial and temporal locality. Therefore, there is a small chance for points of a predicted frame to exist in the reference frame, which makes the search for predicted points in the reference frame a nontrivial effort. To alleviate the high cost of this process, we represent the reference frame in CGA that enables quick point search operations. The proposed NN search method, described in Section V-A, is used to find the corresponding point in the reference frame. This way, we improve the quality of compression, considerably. Two metrics are commonly used for evaluating the cost of residuals, namely, mean squared error (MSE) or sum of absolute differences (SAD). SAD is preferable for very large-scale integration (VLSI) implementation due to its simple computational steps. Therefore, we use SAD to choose the best residual blocks in this work. After each block search, the smallest SAD candidate is chosen as the best
matching reference for each point of the reconstructed frame. Finally, the residual frame is created through calculating the difference between the RGB values of the reconstructed and reference points.

VI. EXPERIMENTAL SETUP

In this section, we describe the experimental setup for analyzing the performance potentials of our proposed CGA.

A. Algorithms and Representation Formats

We implement four basic point operations (i.e., point cloud merge, 3D to 2D projection, NN search, and point cloud compression). For the baseline point cloud representation, we use the octree structure provided by the PCL library [2], [28]. PCL is a C++ library for 3D point cloud processing, where most mathematical operations are implemented based on an open-source linear algebra library, called Eigen. The primary data structures in PCL are fully optimized. For example, to make the algorithms more efficient, PCL functions pass data around using shared pointers to avoid the need for data re-copying [2].

PCL includes both kd-tree and octree for storing point clouds. This library enables spatial partitioning, down-sampling, and other point cloud operations. In a PCL octree, each node has either eight or no children. The octree root node defines a cubic bounding box that includes all the points. At every level of the tree, a non-empty space is divided into eight sub-spaces until reaching spaces comprising only one point (leaves) [2]. We employ the pcl_octree library for implementing the baseline version of the evaluated point cloud operations.

The moving picture expert group (MPEG) has suggested a coding solution for various categories of point clouds: 1) LiDAR point cloud for dynamically acquired data, 2) surface point cloud for static point cloud data, and 3) video-based point cloud for dynamic content. According to these categories, video-based point cloud processing (V-PCC) and geometry-based point cloud processing (G-PCC) have been developed for [30] and [14], respectively. V-PCC focuses on point sets with a relatively uniform distribution of points; whereas, G-PCC is more suited to point clouds with sparse distributions. In this paper, we evaluate G-PCC as an application of point cloud compression.

B. Performance Measurement

We use a Windows(R) 10 machine with 8 GB of DRAM and an Intel(R) Core(TM) i7 CPU operating at 2.5 GHz. For the evaluations, we employ the Intel VTune Profiler [29] to measure performance in terms of execution time ($\tau$), number of last-level cache misses ($\mu$), and the amount of transferred data to/from memory ($\delta$). We use $\mu/\tau$ as a proxy for the memory bandwidth utilization. Notice that every last-level cache miss needs at least one access to DRAM that consumes a significant time and energy. To provide a range of problem sizes, we create several versions of each input sequence by sampling certain number of points. We use the MATLAB function pcdownsample for down-sampling point clouds (https://www.mathworks.com/help/vision/ref/pcdownsample.html).

C. Point Cloud Data

In this paper, we evaluate LiDAR and surface point clouds that exhibit significant sparsity. We evaluate six point cloud sequences of this kind. For the surface point clouds, we use the 8i voxelized surface light field (8iVSLF) dataset [31]. For each point cloud in 8iVSLF, the full body of a human subject is captured using 39 synchronized RGB cameras at 30 frames per second (fps). We select Soldier, Longdress, Loot, and RedandBlack [24] from the 8iVSLF dataset for our evaluations. The other two point cloud sequences, GT_Madame and Urban scene [32], are generated by LiDAR scanners [25]. GT_Madame contains two PLY files, each one with 10 million points that are collected from a street in France, called Rue Madame. The Urban scene dataset consists of point cloud with around 2 KM of mobile laser scanning acquired in two cities.

VII. EVALUATION

In this section, we first provide an analysis of time-space complexity for the proposed CGA compared to the state-of-the-art data representations. Then, we evaluate the performance potentials of the proposed CGA for the basic point cloud operations.

A. Analysis of Time and Space Complexity

Construction complexity, access time, and memory space requirement are some key metrics to evaluate the efficiency of point cloud representations. Construction complexity determines the ease of use and practicality of a representation for real-world applications. In particular, lightweight data structures are desirable for online and frequent (re-)construction of point clouds. Moreover, random accessing and scanning points are inevitable for almost all point cloud operations.

During an octree construction, a three-dimensional space between the minimum and maximum coordinates of points is divided into eight octants. Recursively, each octant is further divided into smaller octants until we reach an empty space. The depth of an octree (i.e., $\delta$) for a point cloud with $N$ points is defined as $\lceil \log_8 N \rceil \leq \delta < N$. As a result, the best case access time to a point in an octree exhibits an $O(\log N)$ complexity. Note that an octree traversal requires a memory indirection for transitioning from each node to a child, which often results in more significant bandwidth overheads than scanning linear arrays. In contrast, accessing a points in CGA requires scanning the $X_{index}$ array, followed by partial scans of the $Y_{index}$ and $Z_{index}$ arrays. Firstly, CGA array scans are more bandwidth efficient than memory indirection in octrees. Secondly, given a point cloud with $N$ points, the length of a search in CGA is defined as $3 \leq \lambda \leq N$. Therefore, the best case access time for a point in a CGA exhibits an $O(1)$ complexity. CGA provides a significantly less time complex method for point access.

The complexity of memory storage used by CGA depends on the size of arrays used for representing points, which linearly grows with the number of points (i.e., $O(N)$). In contrast, the complexity of memory space for octree is $O(N \times \log N)$. Windowed priority queue (WinPQ) [4] is a data structure from the literature that employs pointers similarly to the proposed CGA.
WINPQ holds references to points in a one-dimensional array. Intervals of points are extracted from WINPQ, where each interval contains all of its points within a specific one-dimensional window. The interval of points can be updated by moving the window in discrete steps. The one-dimensional WINPQ can be used to scan higher-dimensional point cloud data by instantiating it repeatedly within a nested loop code structure. Therefore, for three-dimensional point clouds, the time complexity of point operations using WINPQ is $O(N^3)$, where $N$ is the number of points. WINPQ is much more complex than the proposed CGA.

We consider the five data structures explained in Section II-A as baselines for performance evaluation of the proposed CGA data structure. We compare the amounts of memory consumed by each data structure as well as their construction time. Table I shows the construction time and the required memory for various point cloud representations and for computer-generated point cloud sequences. The number are averages across all evaluated point cloud frames. CGA exhibits a higher time-memory efficiency than the other formats.

### Table I

| Point Cloud Representation | Construction Time (s) | Memory Requirement (MB) |
|---------------------------|-----------------------|-------------------------|
| 1D-Array                  | 12.24                 | 138.94                  |
| Voxel Hashing             | 13.08                 | 76.70                   |
| Octree                    | 15.58                 | 80.31                   |
| Integral Octree           | 19.45                 | 117.25                  |
| kd-tree                   | 7.30                  | 80.04                   |
| CGA                       | 3.8                   | 11                      |

**B. Performance of Point Operations**

In this section, we assess the performance of CGA for point cloud processing, i.e., merging, projection, NN search, and compression of 3D point clouds. The point cloud operations are implemented as explained in Section V. For the baseline method, we have considered the octree structure from the PCL library [2].

For the merge operation, two views of each point cloud sequence are merged together as shown in Fig. 3. For projection, the orthogonal projection method is implemented. The NN search algorithm is implemented as explained in Section V-A. For the evaluations, the number of nearest neighbors is considered to be five.

Finally, we study the benefits of temporal prediction in the compression of geometry in 3D point cloud sequences as explained in Section V-D. For the baseline method, all of the frames are coded using their best set of parameters in the MPEG G-PCC open-source library [30]. To find the corresponding point to each point of the reconstructed block in the reference frame, we need to store the points in a searchable data structure. The data structure may be a sparse matrix because the point cloud does not necessarily include all the possible points of a 3D space. Storing the non-existing points would lead to polluting memory. Even in its sparse representation form, a typical point cloud comprises millions of points, which imposes a significant pressure on the memory capacity and bandwidth.

Our proposed CGA format enables a faster lookup than octrees to remove the spatial and temporal redundancy during the compression process. The results in Table II indicate that CGA can replace octrees in point cloud compression to enable significantly faster point lookup operations. For the baseline method, all of the frames are encoded using their best set of parameters in the MPEG G-PCC [30].

We measure performance in terms of the processing time, memory bandwidth, and the amount of transferred data between CPU and memory for the computer-generated and LiDAR sequences as shown in Fig. 5. The x-axis of each plot shows the number of points for each sequence. We vary the number of points by downsizing the point clouds to three different versions in addition to their original sequence using the MATLAB `pc-downsample` function. Then, the processing time, bandwidth utilization, and the amount of data transferred in various point cloud operations are investigated using CGA and octree structures. The y-axes show the improvement of processing time, bandwidth utilization, and the amount of data transferred for various point cloud operations. The results indicate that CGA can replace octree in point cloud processing to improve the processing time, bandwidth utilization, and the amount of data transferred, significantly.

The main reason for the significant improvements in the execution times of LiDAR point cloud compared to the computer-generated ones is that the LiDAR point cloud data used in this paper have no color attributes and their geometric values are much smaller than the computer-generated point cloud sequences. Therefore, the amount of processed data is much less than that of the computer-generated sequences. The existence of peaks and valleys depends on the type of point cloud data and the hardware configuration. The evaluations are performed on real hardware and the number of cache misses affects the bandwidth utilization, considerably.

Increasing the number of points in a point cloud results in a higher resolution model where the points are sampled closely. Therefore, the similarity across neighboring points in a high resolution point cloud increases, which may result in more correlated coordinates and attributes in siblings of a tree structure. Such high correlation in large point cloud models may result in placing similar points under the same branches of the tree thereby improving spatial locality and reducing cache misses. This trend is observed in our experiments across multiple versions of point clouds generated through down-sampling. The peaks in Fig. 5 show the related bandwidth utilization for the downsized point clouds. Nevertheless, the large point clouds cannot fit in our 384 KB cache system, which results in significant time overheads. However, some downsized versions of...
point clouds, can better fit in the cache, which result in less bandwidth overheads—i.e., the valleys shown in the figure.

The normalized bandwidth utilization of CGA over PCL for the computer-generated and LiDAR sequences demonstrate the effectiveness of CGA at gaining a significantly higher bandwidth utilization for the evaluated point cloud operations. One of the main reasons for the superior performance of CGA over PCL is the reduced amounts of data transfer during point cloud processing. Table III shows the details of speedup, memory bandwidth, and transferred data for CGA. CGA achieves $998 \times$ speedup, $410 \times$ better bandwidth utilization, and 58% reduction in the volume of transferred data as compared to the state-of-the-art tree-based structures from the PCL library. Our evaluation results indicate that the proposed CGA requires a smaller footprint than the other point cloud representation formats while enabling a faster point lookup. As future work, we plan to leverage CGA for improving learning-based point cloud processing applications.

### TABLE III

**Speedup, Bandwidth Utilization, and the Volume of Transferred Data for CGA Relative to PCL While Performing the Basic Point Operations on Various Datasets**

|                | Soldier | Longdresst | Loot | Redbndblack | Urban scene | GT_Madame | Average |
|----------------|---------|------------|------|-------------|-------------|-----------|---------|
| **Speedup**    |         |            |      |             |             |           |         |
| Merge          | 10.58   | 8.90       | 9.15 | 8.38        | 2429.43     | 11702.4   | 2361.47 |
| Projection     | 2.66    | 2.68       | 3.46 | 3.35        | 250.21      | 6186.62   | 1074.83 |
| NN search      | 104.65  | 123.23     | 123.61| 346.15      | 2181.44     | 435.15    | 552.37  |
| Compression    | 3.583   | 3.87       | 4.35 | 5.27        | 12.47       | 6.11      | 6.30    |
| Average        | 998.74  |            |      |             |             |           |         |
| **Bandwidth utilization** |         |            |      |             |             |           |         |
| Merge          | 10.74   | 9.40       | 11.13| 13.97       | 1649.32     | 3065.27   | 793.30  |
| Projection     | 4.81    | 2.70       | 4.48 | 4.18        | 141.85      | 1423.59   | 263.60  |
| NN search      | 217.67  | 203.78     | 193.75| 1949.66     | 769.06      | 155.60    | 580.92  |
| Compression    | 2.59    | 1.47       | 1.77 | 2.37        | 4.61        | 4.19      | 2.83    |
| Average        | 410.16  |            |      |             |             |           |         |
| **The volume of transferred data** |         |            |      |             |             |           |         |
| Merge          | 0.58    | 0.50       | 0.38 | 0.50        | 0.45        | 0.29      | 0.5     |
| Projection     | 0.82    | 0.64       | 0.51 | 0.59        | 0.45        | 0.34      | 0.59    |
| NN search      | 0.72    | 0.89       | 0.85 | 0.82        | 0.46        | 0.41      | 0.69    |
| Compression    | 0.53    | 0.42       | 0.44 | 0.41        | 0.79        | 0.82      | 0.56    |
| Average        | 0.58    |            |      |             |             |           |         |

### VIII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new data structure for processing point clouds, called CGA. We examined various point cloud operations, including merge, projection, NN search and point cloud compression. Our simulation results on a set of computer-generated and LiDAR point cloud sequences showed that the proposed format provides significant speed-ups, better bandwidth utilization, and less transferred data compared to the state-of-the-art tree-based structures from the PCL library. Our evaluation results indicate that the proposed CGA requires a smaller footprint than the other point cloud representation formats while enabling a faster point lookup. As future work, we plan to leverage CGA for improving learning-based point cloud processing applications.

### REFERENCES

[1] J. Chen and W. Fang, “An improved spatial data structure for web-based large-scale 3D scene,” in Proc. IEEE 10th Int. Conf. Softw. Eng. Serv. Sci., Beijing, China, 2019, pp. 681–684, doi: 10.1109/ICSESS47205.2019.9040768.

[2] R. B. Rusu and S. Cousins, “3D is here: Point cloud library,” in Proc. IEEE Int. Conf. Robot. Automat., Shanghai, China, 2011, pp. 1–4, doi: 10.1109/ICRA.2011.5980567.

[3] H. Roodaki, M. Dehyadegari, and M. N. Bojnordi, “G-arrays: Geometric arrays for efficient point cloud processing,” in Proc. Int. Conf. Acoust., Speech, Signal Process., Toronto, ON, Canada, 2021, pp. 1925–1929, doi: 10.1109/ICASSP39728.2021.9413902.
In 2015, she has been an Assistant Professor with the University of Tehran, Tehran, Iran. Her research interests include multi-view/3D and 360-degree video coding, scalable video coding, video quality assessment, and cloud gaming.

Mahdi Nazm Bojnordi received the Ph.D. degree in electrical and computer engineering from the University of Rochester, Rochester, NY, USA, in 2016. He is currently an Assistant Professor with the School of Computing, University of Utah, Salt Lake City, UT, USA, where he leads the Energy-Efficient Computer Architecture Laboratory. His research interests include energy-efficient architectures, low-power memory systems, and the application of emerging memory technologies to computer systems. His research has been recognized by two IEEE Micro Top Picks Awards, an HPCA 2016 Distinguished Paper Award, and a Samsung Best Paper Award.