

Real-Time Spatio-Temporal LiDAR Point Cloud Compression

Yu Feng¹, Shaoshan Liu², and Yuhao Zhu¹

Abstract—Compressing massive LiDAR point clouds in real-time is critical to autonomous machines such as drones and self-driving cars. While most of the recent prior work has focused on compressing individual point cloud frames, this paper proposes a novel system that effectively compresses a sequence of point clouds. The idea to exploit both the spatial and temporal redundancies in a sequence of point cloud frames. We first identify a key frame in a point cloud sequence and spatially encode the key frame by iterative plane fitting. We then exploit the fact that consecutive point clouds have large overlaps in the physical space, and thus spatially encoded data can be (re-)used to encode the temporal stream. Temporal encoding by reusing spatial encoding data not only improves the compression rate, but also avoids redundant computations, which significantly improves the compression speed. Experiments show that our compression system achieves 40× to 90× compression rate, significantly higher than the MPEG’s LiDAR point cloud compression standard, while retaining high end-to-end application accuracies. Meanwhile, our compression system has a compression speed that matches the point cloud generation rate by today LiDARs and out-performs existing compression systems, enabling real-time point cloud transmission.

I. INTRODUCTION

LiDAR has become an essential sensor in autonomous machines such as self-driving vehicles, autonomous drones, and robots. LiDARs generate massive amounts of point cloud data. For instance, the Velodyne HDL64E LiDAR generates hundreds of thousands of points each frame, amounting to up to 26 MB of raw data per second. Effectively compressing point cloud in real-time enables autonomous machines to be closely connected with each other and with the cloud, ushering in a new era in distributed and cloud robotics. For instance, efficient point cloud compression would enable offloading compute-intensive perception tasks (e.g., object detection) to the cloud to reduce the perception latency; similarly, collaborative decision making across robots relies on efficient point cloud compression to exchange information.

While prior work mostly focuses on the compression rate [11], [16], [27], our work aims to simultaneously improve the compression rate and compression speed while maintaining high accuracy for end-to-end applications of interest (e.g., registration and object detection). High compression speed let the compressed point cloud be transmitted in real-time without local buffering, easing the storage pressure.

We propose a real-time Spatio-temporal point cloud compression technique that delivers high compression rate, maintains high application-level accuracy while delivering a com-

pression speed (>10 Hz) that matches/exceeds the LiDAR point cloud generation speed. We use range image [27] as the basic data representation for point clouds. Range images not only inherently provide a lossless compression of point clouds, but also “regularize” the unstructured 3D point cloud data into a structured 2D data structure, which enables computationally-efficient subsequent processing.

Our method exploits the unique redundancies inherent in LiDAR point cloud capturing — both spatially (within a point cloud) and temporally (across point clouds). We first identify a key point cloud (K-frame) in a point cloud sequence, and transform the rest of the point clouds, which we call predicted clouds (P-frames), into K-frame’s coordinate system using IMU measurements. We spatially encode the K-frame by exploiting that many points in real-world scenes lie on the same plane and can be encoded using planes.

Building on top of the spatial encoding of the K-frame, we exploit the fact that consecutive point clouds share a great chunk of overlapped areas of the scene. Thus, the same set of planes could be used to encode points across point clouds. Our temporal encoding scheme reuses planes identified in the K-frame to encode overlapped scenes in P-frame. The temporal encoding also compensates for the inaccuracies in IMU measurements, improving the robustness of the method.

We evaluate the proposed method using the KITTI dataset. Our compression method achieves up to 90× compression rate, significantly out-performing MPEG’s LiDAR point cloud compression standard [16], [22]. Meanwhile, our compression method operates at least 10 Hz, which matches today’s LiDAR point cloud generation speed and is higher than prior methods. The high compression speed is achieved both by avoiding redundant computations (e.g., reusing the spatially encoded planes calculated in the K-frame) and by a careful parallel implementation of our algorithm.

Finally, unlike prior work that evaluates quality metrics such as Peak Signal-to-Noise Ratio (PSNR) that are not directly tied to end-to-end application-level accuracy, our compression method directly focuses on application-level accuracy. We show that our compression system on three point cloud applications—registration, object detection, and segmentation—retains the similar accuracy as the original point cloud while out-performing the accuracies of existing point cloud compression schemes. Our method delivers high application-level accuracy because spatial and temporal encoding inherently preserve the geometry of points in the scene and denoise the point clouds.

Our main contributions of this paper are as follows:

- To the best of our knowledge, this is the first work that leverages both the spatial and temporal redundancies to

¹These authors are with the Department of Computer Science, University of Rochester, Rochester, NY, USA. yfeng28@ur.rochester.edu, yzhu@rochester.edu

²This author is with PerceptIn, Inc., Santa Clara, CA, USA. shaoshan.liu@perceptin.io

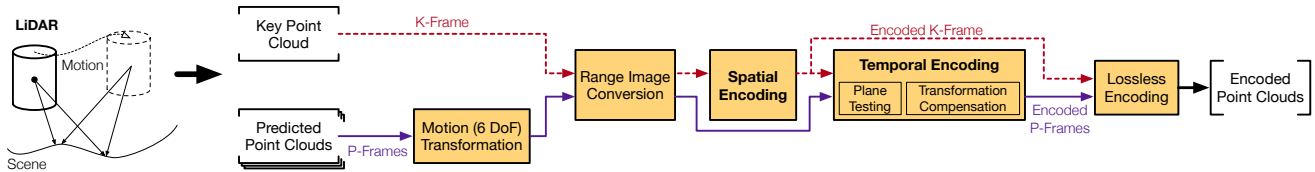


Fig. 1: Overview of our compression system, which compresses a sequence of consecutive point clouds. All the points clouds are converted to range images to accelerate the compression speed. We first spatially encode the key point-cloud (K-frame) in the sequence, typically the middle one. The spatial encoding results of the K-frame are then used to temporally encode the rest of the point clouds, which we call predicted point clouds (P-frames).

compress LiDAR points clouds.

- The compression method simultaneously achieves higher compression rate, higher compression speed, and higher application-level accuracy than today’s compression methods, including the MPEG’s point cloud compression standard.

II. RELATED WORK

Unstructured Point Cloud Encoding Perhaps the most common way to encode point cloud data is to use space-partitioning trees, among which Octree is the most widely used [9], [10], [12], [14], [16], [23], [25]. The G-PCC method in MPEG’s point cloud compression standard falls into this category [16]. Each Octree leaf node could be encoded by either a single occupancy bit, which could be lossless if each leaf node contains exactly one point, or by plane extraction, which preserves more details if each leaf node contains multiple points. G-PCC provides both options. Based on the space-partitioning tree representation, prior work has explored various methods to reduce redundant information, such as 2D projection [10] or surface fitting [23].

Prior work also exploited temporal redundancies in space-partitioning trees such as XORing the two consecutive Octrees [14], using motion compensation in 3D space [25], or applying video compression directly [9].

Other unstructured point cloud representations include shape adaptive wavelet [6], [20] and hierarchical height map [10], [18]. While effective in certain use-cases, the downside of unstructured representations is that they do not exploit the unique characteristics exposed by LiDAR point clouds, leading to generally low compression rate.

Structured Point Cloud Compression Instead of encoding point clouds using space-partitioning trees, another category of compression methods convert point clouds into 2D images using spherical projection [24], [26], [27] or orthogonal projection such as the V-PCC method in the MPEG’s standard [13], [15]. Existing image/video compression methods are then used to further compress the projected images [13], [24], [27]. However, directly applying image/video compression algorithms does not preserve the spatial information inherit in the point cloud, and thus generally results in low application accuracy.

III. SPATIO-TEMPORAL COMPRESSION

This section introduces our spatial-temporal LiDAR point cloud compression algorithm. We first present an overview

of our compression system (Sec. III-A), followed by the detailed designs of the three key components: range image conversion (Sec. III-B), spatial encoding (Sec. III-C), and temporal encoding (Sec. III-D). Finally, we discuss our parallel implementation that further improves the compression speed (Sec. III-E).

A. Main Idea

The idea of our compression system is to exploit redundancies both within a point cloud (spatial) and across point clouds (temporal). Spatially, many surfaces in the real-world are planes (e.g., walls and ground); even non-plane surfaces could be approximated by a set of planes. Temporally, consecutive point clouds share a great chunk of overlapped areas of the scene; thus, the same set of planes could be used to encode points across point clouds. While intuitive, exploiting spatial and temporal redundancies in real-time is challenging due to the irregular/unstructured point cloud and the compute-intensive plane fitting process.

We propose a compression system that simultaneously achieves the state-of-the-art compression rate and compression speed while maintaining high application accuracies. Fig. 1 provides a high-level overview of our system, which consists of three main blocks: range image conversion, spatial encoding, and temporal encoding. Fig. 2 shows the relevant data structures during the encoding process.

Given a sequence of consecutive point clouds, we differentiate between two point cloud types: key point cloud (K-frame) and predicted point cloud (P-frame). A sequence has only one K-frame and the rest is P-frames. P-frames are first transformed (both translation and rotation) to K-frame’s coordinate system using the IMU measurements. After transformation, each point cloud is converted to a range image [27] for subsequent computations. The range image not only provides an initial compression to the original point cloud, but also provides a structured representation of the (unstructured) point cloud that is hardware-friendly.

We then spatially encode K-frame by fitting planes; the fitted planes in the K-frame are then (re-)used to temporally encode P-frames, greatly improving the overall compression rate and speed. In order to be robust against transformation errors, which might be introduced due to noisy IMU observations, we propose a set of techniques that compensate the sensor noise and preserve the encoding quality.

In the end, after spatial and temporal encoding, most of the tiles in the range images are plane-encoded; the unfit

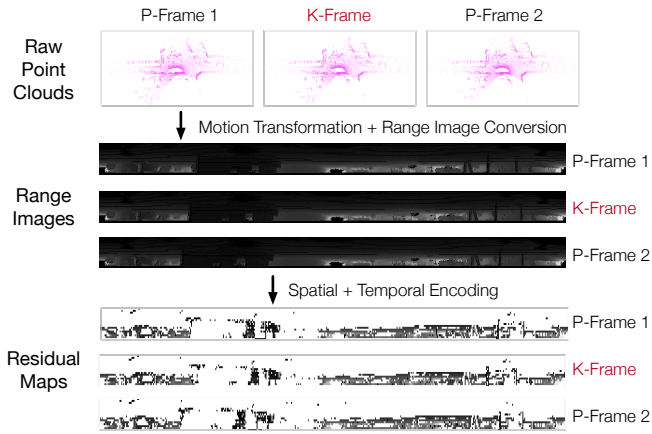


Fig. 2: Different data structures used in our compression. The raw point clouds are converted to range images. After spatial and temporal encoding, most of the tiles in the range images are plane-encoded; the unfit tiles are left in the residual maps.

tiles are left in what we call residual maps (Fig. 2). The planes and the residual maps are then further compressed by a lossless compression scheme (e.g., Huffman encoding) to generate the final encoded data.

Overall, in addition to providing high compression rate and speed, our compression system also preserves application accuracy. This is because plane fitting inherently removes noises and outliers in the point clouds without requiring explicitly removing outliers that prior work employs [24].

B. Range Image Conversion

We first convert the raw point cloud data to a range image, which essentially converts every point (x, y, z) in the 3D Cartesian space to a pixel at coordinates (θ, ϕ) in the range image with a pixel value r :

$$r = \sqrt{x^2 + y^2 + z^2}; \quad (1)$$

$$\theta = \arctan\left(\frac{x}{y}\right)/\theta_r; \quad \phi = \arccos\left(\frac{z}{r}\right)/\phi_r \quad (2)$$

where θ_r and ϕ_r are the horizontal and vertical resolutions of the LiDAR, respectively.

A range image naturally compresses the original point cloud, because each point (x, y, z) can be encoded with just a range value r of the corresponding pixel in the range image; θ and ϕ are the pixel’s coordinates and do not have to be explicitly encoded. If θ_r and ϕ_r are the same as the resolutions of the LiDAR, range image is a lossless compression of the corresponding point cloud. Mathematically, however, θ_r and ϕ_r could be any arbitrary positive values; larger θ_r and ϕ_r would lead to a lower range image resolution, providing a lossy compression of the original point cloud.

In addition to providing an inherent compression scheme, range image brings two key advantages. First, operating on range images is computationally more efficient than directly accessing the point cloud, which requires tree traversals that lead to high cache misses and branch mis-predictions on

today’s hardware architecture [17], [30]. Second, adjacent pixels in the range map are likely to lie on the same plane, because they correspond to consecutive scans from the LiDAR. This characteristic allows us to encode the entire range image more efficiently.

C. Spatial Encoding

The goal of spatial encoding is to encode all the points that lie on the same plane using that plane. Intuitively, many surfaces in the real-world are planes (e.g., walls and ground); non-plane surfaces could be approximated by a set of planes.

In the 3D Cartesian space, a plane can be expressed as:

$$x + ay + bz - c = 0 \quad (3)$$

where $(1, a, b)$ is the normal vector of the plane and $\frac{|c|}{\sqrt{1+a^2+b^2}}$ is the distance from the origin (LiDAR center) to the plane. Thus, all the points on the same plane could be encoded with just the three coefficients of the plane. Note that the exact position of each point on the plane is not explicitly encoded. The decoding process would simply have to simulate a ray casting process to find the intersection of a ray and the plane to reconstruct the position of a point.

To encode the entire point cloud, which contains points that lie in many different planes, we use a “divide and conquer” strategy. Specifically, we first uniformly divide the range image into unit tiles (e.g. 4×4). We start by fitting a plane for points in the first tile, and gradually grow to include adjacent tiles, essentially forming a bigger tile. Each time we grow, we test whether the plane fit so far can be used to encode all the points in the new (bigger) tile under a predefined threshold. If so, all the points in the new tile are encoded with the plane. Otherwise, we start from the current tile and repeat the process until all the tiles in the range image are processed.

Our spatial encoding process grows tiles horizontally, which we find coalesces many more adjacent tiles than growing vertically. This is inherently because today’s LiDARs have a much more fine-grained horizontal resolution than vertical resolution. For instance, Velydone’s HDL-64E has a 0.08° horizontal resolution, and a 0.4° vertical resolution. As a result, points in horizontally adjacent tiles are closer to each other and, thus, more likely to fit in the same plane than points from vertically adjacent tiles.

Fitting a plane given points can be naturally formulated as a linear least squares problem [19]. While classic iterative methods such as RANSAC [7] are widely used, we find that directly calculating the closed-form solution is generally faster, because deriving the closed-form solution requires less computation and also the computations could be parallelized.

Note that we intentionally do not encode the deltas of plane fitting (i.e., the difference between a true point and a predicted point on the plane). Instead, we find that when a reasonably small threshold is used, discarding deltas effectively *denoises the point cloud*, leading to higher application accuracy than even the original point cloud (Sec. V-A).

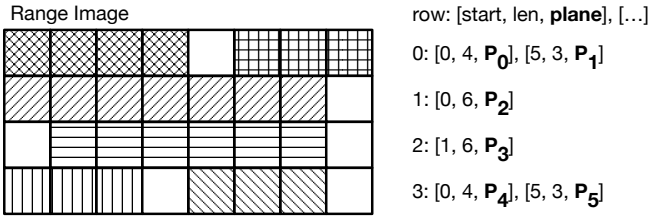


Fig. 3: A spatial encoding example. The range image on the left is first tiled, and then iteratively plane-fitted. Horizontally adjacent tiles fit by the same plane are shaded by the same stripe pattern. Tiles that are plane-fitted are encoded using the format shown on the right. Points in unfit tiles are encoded individually using their range values (not shown).

In order to reconstruct/decode the range image later, each row in the range image is encoded with a row ID followed by a set of three-tuples $[s, len, \mathbf{P}]$. Fig. 3 provides an example. Each three-tuple corresponds to a sequence of adjacent tiles in that row, starting from s to $s + len$, that are fit by the same plane \mathbf{P} , which is parameterized by the three coefficients (Equ. 3). Inevitably, there are tiles that contain points that can not be fit on planes, because, for instance, those points are sparse samples of an irregular surface. These “unfit” points are left in what we call a residual map (Fig. 2) and are directly encoded using their raw range values.

D. Temporal Encoding

Spatial encoding provides a building block to encode point clouds individually. LiDARs in autonomous machines, however, generate a sequence of point clouds. While it is possible to individually apply spatial encoding to each point cloud, doing so loses opportunities exposed by the temporal correlations across consecutive point clouds.

Consecutive point clouds have large chunks of overlaps, because they are just different samples of the same physical scene. Using the KITTI dataset, we find that on average 99% of each point cloud is geometrically overlapped with the previous point cloud. Motivated by this observation, temporal encoding encodes a set of consecutive point clouds together. The idea is to use one plane to encode the overlapped scene across multiple consecutive point clouds. Doing so improves both the compression rate and the compression speed by avoiding plane fitting in each point cloud.

Transformation Each point cloud has its own coordinate system when generated by the LiDAR. In order to fit planes across a sequence of point clouds, we convert all the point clouds to the same coordinate system—by performing a 6 DoF (translation and rotation) transformation. Fig. 4 compares the effect of overlaying five consecutive point clouds together in the same coordinate system before and after the transformation. Without motion transformation point clouds at different timestamps are mis-aligned, making temporal encoding challenging.

To unify point clouds in the same coordinate system, we must 1) decide a key point cloud K , whose coordinate system is used as the transformation target, and 2) calculate

the corresponding transformation matrix \mathbf{M}_i between K and every other point cloud P_i .

In our system, we calculate the transformation matrix using the IMU measurements, which provides the translational acceleration ($\hat{\mathbf{a}}$) and rotational rate ($\hat{\omega}$). Using the IMU measurements, we estimate the translation vector $T_{3 \times 1}$ as:

$$T_{3 \times 1} = [\overline{\Delta x} \quad \overline{\Delta y} \quad \overline{\Delta z}] \quad (4)$$

where $\overline{\Delta x}$, $\overline{\Delta y}$, and $\overline{\Delta z}$ are translational displacements integrated from $\hat{\mathbf{a}}$ using the first-order Runge-Kutta numerical method. Similarly, the rotation matrix $R_{3 \times 3}$ is estimated as:

$$R_{3 \times 3} = \begin{bmatrix} \cos(\overline{\Delta \alpha}) & \sin(\overline{\Delta \alpha}) & 0 \\ -\sin(\overline{\Delta \alpha}) & \cos(\overline{\Delta \alpha}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos(\overline{\Delta \beta}) & 0 & -\sin(\overline{\Delta \beta}) \\ 0 & 1 & 0 \\ \sin(\overline{\Delta \beta}) & 0 & \cos(\overline{\Delta \beta}) \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\overline{\Delta \gamma}) & \sin(\overline{\Delta \gamma}) \\ 0 & -\sin(\overline{\Delta \gamma}) & \cos(\overline{\Delta \gamma}) \end{bmatrix} \quad (5)$$

where $\overline{\Delta \alpha}$, $\overline{\Delta \beta}$, and $\overline{\Delta \gamma}$ are rotational displacements integrated from $\hat{\omega}$ using the first-order Runge-Kutta method.

We use the middle point cloud in a consecutive point cloud sequence as the key point cloud (K-frame). This minimizes the impact of cumulative IMU sample errors when calculating the transformation matrix. Every other point cloud, which we call predicted cloud (P-frame), is transformed to K-frame’s coordinate system by:

$$p'_{4 \times 1} = \mathbf{M} p_{4 \times 1} = \begin{bmatrix} R_{3 \times 3} & T_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}_{4 \times 4} p_{4 \times 1} \quad (6)$$

where $p_{4 \times 1}$ and $p'_{4 \times 1}$ denote a point in a predicted cloud before and after transformation, respectively.

All the N point clouds in a sequence, after transformation, are converted to range images with the same dimension. For the ease of manipulation, we stack the N range images together to form a N -channel image.

Note that it is possible that points in a P-frame after transformation could collide, i.e., mapped to the same range image pixel, in which case we preserve the nearest point. On the KITTI dataset, about 4.6% of the points collide when transforming between two adjacent point clouds, and this percentage increases as the gap two point clouds increases. This suggests that the number of consecutive point clouds that are encoded together (N) affects the encoding results. We show the sensitivity to N in Sec. V-C.

Encoding We use the same “divide-and-conquer” strategy used in spatial encoding to encode across channels (point clouds). A naive implementation would be to fit all the points in a tile across all N channels (e.g., $4 \times 4 \times N$) and then grow to adjacent tiles. However, this approach is susceptible to IMU measurement errors. Inaccurate IMU observations lead to inaccurate point cloud transformations. As a result, points in the same tile across different channels might not end up lying on the same plane, leading to poor plane fitting results.

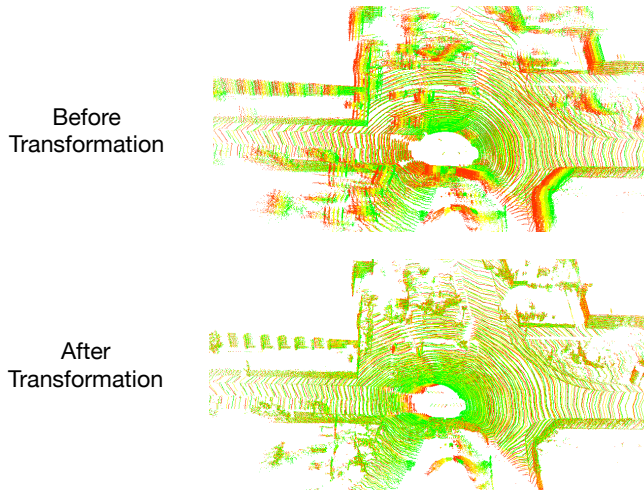


Fig. 4: Stacking five consecutive point clouds before (top) and after (bottom) motion transformation. Colors indicate different point clouds. Motion transformation better aligns different point clouds in one coordinate system.

We propose an effective method to temporally encode across channels while compensating the transformation errors. Our idea is to first fit a tile in the K-frame’s channel, and use the fitted plane Q to test against the same tile in each of the other channels. Critically, we hold Q ’s normal vector constant while varying its distance to the origin (i.e., varying c in Equ. 3). This effectively compensates the translation error in the IMU measurements. If the relaxed plane Q' (parameterized by a, b, c') fits all the points in a channel under a certain threshold, only c' needs to be encoded for that channel rather than all three plane coefficients.

We apply the same horizontal growing strategy until all the tiles of all the channels in the range image are processed, at which point we remove all the encoded tiles from the range image. The remaining range image I' contains tiles that could not be fitted across channels even after compensating translation errors. We then spatially encode I' channel by channel using the same process described in Sec. III-C. Effectively, this channel-wise spatial encoding compensates the rotation errors in transformation. In the end, the unfit tiles are left in the residual map (Fig. 2), which is further compressed in a lossless fashion along with the fit planes.

Temporal encoding not only provides high compression rate, but also improves the compression speed compared to spatially compressing each point cloud individually. This is because the planes fit in the K-frame are reused in P-frames, reducing the plane fitting overhead.

E. Parallel Optimizations

The speed of the sequential implementation of our algorithm scales linearly with respect to number of channels and angular resolution of the LiDAR. To further improve the compression speed, we exploit the parallelisms exposed by our encoding system and leverage parallel hardware available in modern processors.

At the high level, we exploit both the thread-level parallelism (TLP) and data-level parallelism (DLP). During the range image conversion, we exploit the TLP where each thread is responsible for converting one point cloud into the corresponding range image. During spatial encoding, we leverage TLP where each thread is responsible for encoding a row in the K-frame. During temporal encoding, each thread is responsible for testing planes in a P-frame.

The actual computation in each thread also exposes data-level parallelism such as computing the immediate results (radius, indexes) and the various matrix operations in the plane-fitting and plane-testing processes. Our implementation uses the OpenMP programming model in C++ to exploit both TLP and DLP.

IV. EVALUATION METHODOLOGY

Applications and Evaluation Metrics We evaluate our compression method on three common point cloud applications: registration, object detection, and scene segmentation:

- Registration: we use a recent ICP-based registration pipeline [30] developed using the widely-used PCL [21].
- Object Detection: we use VoxelNet [31], a Deep Convolution Neural (DNN)-based approach.
- Scene Segmentation: we use SqueezeSeg [29], a DNN-based approach.

We use three evaluation metrics: compression rate over the uncompressed point clouds, compression speed in FPS, and application-level accuracy. We evaluate the application-level accuracy instead of common quality metrics such as PSNR or RMSE because we want to assess how compression affects point cloud applications, which is what ultimately matters.

Dataset We use the widely-used KITTI dataset [8] for evaluating registration and object detection. We evaluate on all the sequences and frames for comprehensiveness. To evaluate segmentation, we use SemanticKITTI [5], which augments KITTI dataset for segmentation tasks. We report geometric mean results unless otherwise noted.

Baseline We compare against four baselines:

- G-PCC: It is a point cloud compression standard proposed by the MPEG [16] specifically designed to compress LiDAR point cloud data. It constructs an Octree for a point cloud and encodes the Octree.
- V-PCC: It is a point cloud compression standard proposed by the MPEG [13], [15] designed to compress dense point clouds used in volumetric rendering. It maps point clouds to images and uses existing video compression to compress the images.
- JPEG: It compresses each point cloud’s range image individually using the (lossy) JPEG codec [4].
- H.264: It compresses a sequence of point clouds by compressing the corresponding range image sequence using the H.264 video codec [28]. We shows results of both the lossy and lossless versions.

Variants Our method can be configured in two modes: the *single-frame* mode that applies only spatial encoding to individual frames and the *streaming* mode that applies both

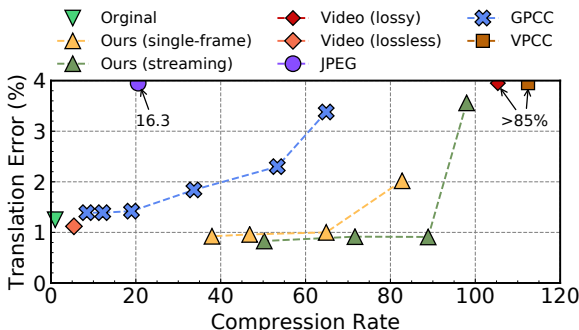


Fig. 5: Registration translation error and compression rate comparison of various compression methods.

spatial and temporal encoding to a sequence of frames. For both versions, we vary the threshold of plane fitting to form different design points.

Hardware Platform We implement our compression method in C++ and evaluate the compression speed on both a PC, Intel i5-7500 with 4 cores, and a mobile platform, Nvidia Jetson TX2 [2], which represents the compute capability of mobile robots or drones.

V. EVALUATION

We first show the end-to-end accuracy and compression rate of our compression method on three general robotic applications: localization, object detection, and 3D scene segmentation, compared against a range of existing methods (Sec. V-A). We then demonstrate that our compression speed matches the point cloud generation speed and surpasses other methods (Sec. V-B). Last, we evaluate the sensitivity of our compression method (Sec. V-C).

A. Compression Rate vs. Accuracy

This section assumes that we compress five consecutive point clouds together unless otherwise noted. We will later study the sensitivity of different frame configurations.

Localization Our compression method outperforms other methods in both application accuracy and compression rate. Fig. 5 compares the translation error (y-axis) against the compression rate (x-axis) of different compression methods.

Our method in the streaming mode can achieve an $88.9\times$ compression rate with only 0.91% translation error, and the single-frame mode achieves $59.7\times$ compression rate with 0.96% translation error. In comparison, the best G-PCC compression has a 1.38% translation error with only $8.5\times$ compression rate. Interestingly, our compression methods have lower errors than using the original point clouds (1.25%). This is because our plane fitting process inherently reduces the noise from the point cloud.

Other baselines including JPEG compression on range images, lossy H.264 video compression, and V-PCC have much higher localization errors ($> 16\%$) as Fig. 5 shows. Although the lossless video compression has better localization accuracy, its compression rate ($5.4\times$) is much lower.

Object Detection On KITTI dataset object detection uses only individual point clouds instead of point cloud

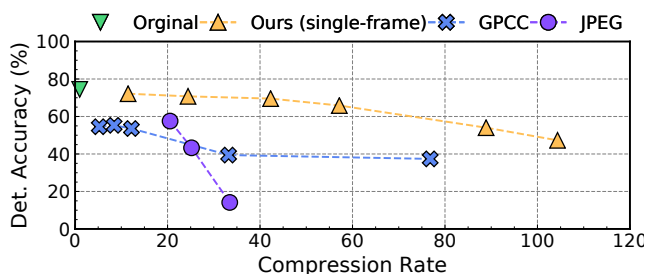


Fig. 6: The object detection accuracy and compression rate comparison of various compression methods.

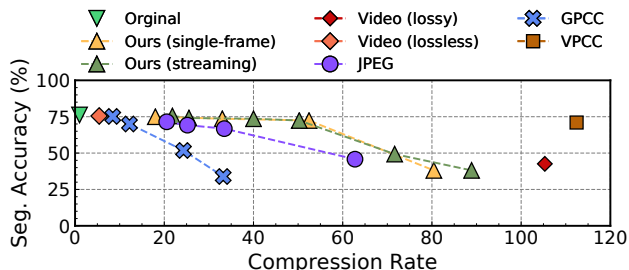


Fig. 7: The segmentation error and compression rate comparison of various compression methods.

sequences. Thus, we present only the single-frame variant of our compression system. For the same reason, V-PCC and H.264 compression methods are not applicable. Fig. 6 compares the object detection accuracy against compression rate across different compression methods. Our method Pareto-dominates the prior methods.

Comparing against the 74.4% accuracy using the original point clouds, our compression method achieves a comparable accuracy of 72.2% with a $11.5\times$ compression rate. In addition, our compression method achieves more than $42.3\times$ compression rate while still keeping the accuracy over 70%. In contrast, the best accuracy that G-PCC and JPEG achieve is 42.1% and 57.6% with the compression rate of 15.3 and 20.6, respectively.

Segmentation Fig. 7 shows the compression rate vs. segmentation accuracy trade-offs across the different compression schemes. Our methods Pareto-dominates other methods except for lossless video compression. In particular, our method achieves a better compression rate ($21.8\times$) than G-PCC ($8.5\times$) and JPEG ($20.6\times$) with a similar accuracy at 75.5%. The accuracies of G-PCC and JPEG drop quickly as the compression rates increase while our method maintains a high accuracy (72.4%) even at a compression rate of $50.3\times$.

Lossless video compression achieves little accuracy drop with only a $5.4\times$ compression rate; lossy video compression, in contrast, has the highest compression rate—at the expense of over 30% accuracy drop.

B. Compression Speed

Fig. 8 and Fig. 9 show the compression speeds on both a PC and the Nvidia TX2 mobile platform, respectively. Our compression method outperforms G-PCC by about one order

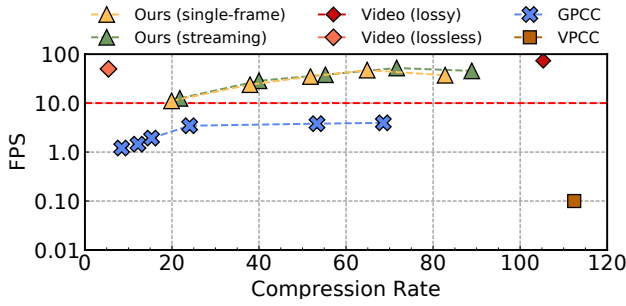


Fig. 8: Compression speed vs. compression rate of various methods on Intel i5-7500 CPU.

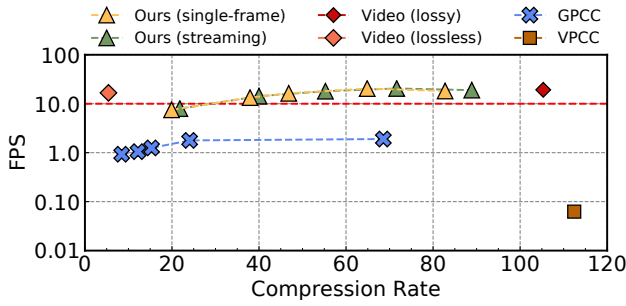


Fig. 9: Compression speed vs. compression rate of various methods on Nvidia mobile TX2 platform.

of magnitude on both platforms. The compression speed on a PC could be as high as 52.1 FPS, and even on the mobile TX2 the compression speed could be as high as 20.5 FPS. As today’s LiDARs generally operate at between 5 Hz to 20 Hz [1], [3], our compression method could be executed in real-time as the point clouds are being generated. Lossy and lossless video compressions have a similar compression speed. However, as shown before, they either have a much lower compression rate or lead to much lower application accuracies. V-PCC is much slower than other methods.

C. Sensitivity Study

All results shown so far assume that five consecutive point clouds are encoded together. Fig. 10 shows how compression rate, application accuracy, and compression rate vary with the number of consecutively encoded frames. All results are normalized to the results where the number of consecutive frames is five. Object detection uses individual frames, so its accuracy numbers are not shown.

We find that the compression speed is most sensitive to the number of encoded frames. This is because our implementation parallelizes many operations across frames such as the range image conversion and plane testing. More frames provides more opportunities for parallelization, leading to higher speeds. The application accuracies are mostly insensitive to the number of frames, because our compression method is able to preserve the vast majority of points during motion transformation and encoding.

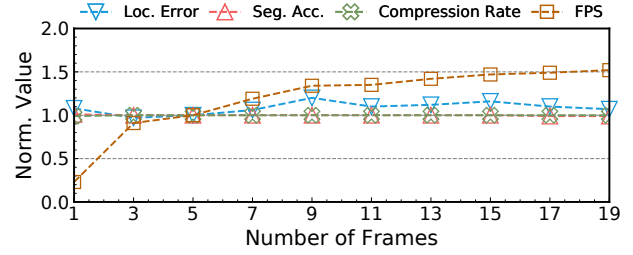


Fig. 10: Sensitivity study on application accuracy, compression rate, and compression speed by varying the number of consecutive frames that are encoded together.

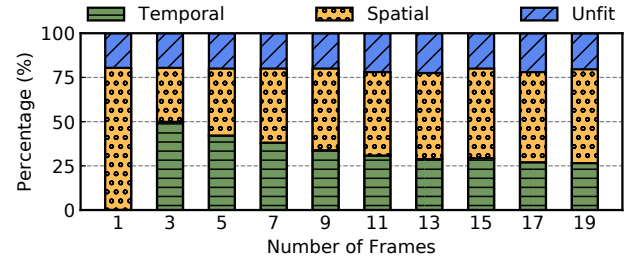


Fig. 11: Distribution of different encoding types within a point cloud sequence as the number of consecutively encoded frames varies. “Unfit” refers to points that could not be encoded in either method. Note that with only one frame there is no temporally encoded frame.

It is worth noting that the compression rate is relatively insensitive to the number of frames. To understand why, Fig. 11 shows the distribution of how different points are encoded in a sequence. As the the number of frames increases, the percentage of temporally encoded points decreases because the overlapped region becomes smaller, while the percentage of spatially encoded points increases. The overall percentage of points that are encoded by either methods stays roughly the same, leading to a roughly stable compression rate. Note that as the number of frames increases the decoding speed is faster with similar accuracy as shown in Fig. 10, indicating longer sequences are preferred in encoding point clouds.

Fig. 12 shows the speedup of our parallel compression system over a sequential baseline. Recall from Sec. III-E that our implementation exploits various forms of parallelism to improve the speed. With five frames available for compression, we achieve a $3.8\times$ speedup over a sequential implementation. With 19 frames available, the speedup is $5.4\times$. As the number of consecutive frames increases, the speedup saturates because of the hardware resource limitation.

VI. FUTURE WORK

While effectively, our existing compression algorithm also points out several areas of improvement for future work.

First, our current design uses native IMU measurements to register consecutive point cloud frames, which could suffer from inaccurate IMU measurements. Performing precise registration (e.g., using SLAM) could be computationally costly,

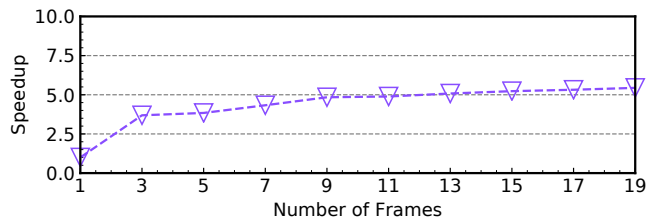


Fig. 12: Speedup of our parallel implementation over a sequential implementation on a four-core Intel i5-7500 CPU as the number of consecutive frames increases.

defeating our goal of real-time compression. It would be interesting to investigate lightweight yet precise registration algorithms specifically designed for compression.

Second, our method extracts the geometric and temporal information from point cloud frames such as the plane coefficients, which could be used to assist certain robotic applications such as localization. Finally, it would be interesting to investigate point cloud algorithms that directly operate on compressed (encoded) point cloud.

VII. CONCLUSION

Efficient point cloud compression will enable autonomous machines to be more connected with each other and with the cloud, and thus usher in a new era in distributed and cloud robotics. This paper proposes a novel spatio-temporal scheme for compressing LiDAR point clouds. We show that by exploiting spatial and temporal redundancies across consecutive point clouds, our compression method achieves up to $90\times$ compression rate, maintains high application accuracy while achieving real-time (>10 FPS) compression speed. It out-performs the state-of-the-art point cloud compression standards on compression rate, speed, and accuracy.

REFERENCES

- [1] "HDL-64E: High Definition Real-Time 3D Lidar." [Online]. Available: <https://velodynelidar.com/products/hdl-64e/>
- [2] "NVIDIA Jetson TX2 Module." [Online]. Available: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/>
- [3] "Velodyne Lidar Introduces Velabit." [Online]. Available: <https://velodynelidar.com/press-release/velodyne-lidar-introduces-velabit/>
- [4] "Information technology — jpeg 2000 image coding system," 2000.
- [5] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall, "Semantickitti: A dataset for semantic scene understanding of lidar sequences," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 9297–9307.
- [6] I. Daribo, R. Furukawa, R. Sagawa, H. Kawasaki, S. Hiura, and N. Asada, "Point cloud compression for grid-pattern-based 3d scanning system," in *2011 Visual Communications and Image Processing (VCIP)*, 2011.
- [7] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, 1981.
- [8] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.
- [9] T. Golla and R. Klein, "Real-time point cloud compression," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

- [10] A. Hornung, K. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous Robots*, 2013.
- [11] H. Houshfar and A. Nüchter, "3d point cloud compression using conventional image compression for efficient data transmission," in *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*, 2015.
- [12] Y. Huang, J. Peng, C.-C. J. Kuo, and M. Gopi, "Octree-based progressive geometry coding of point clouds," in *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, 2006.
- [13] E. S. Jang, M. Preda, K. Mammou, A. M. Tourapis, J. Kim, D. B. Graziosi, S. Rhyu, and M. Budagavi, "Video-based point-cloud-compression standard in mpeg: From evidence collection to committee draft [standards in a nutshell]," *IEEE Signal Processing Magazine*, 2019.
- [14] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach, "Real-time compression of point cloud streams," in *2012 IEEE International Conference on Robotics and Automation*, 2012.
- [15] M. Krivokuća, P. A. Chou, and M. Koroteev, "A volumetric approach to point cloud compression—part ii: Geometry compression," *IEEE Transactions on Image Processing*, 2020.
- [16] S. Lasserre, D. Flynn, and S. Qu, "Using neighbouring nodes for the compression of octrees representing the geometry of point clouds," in *Proceedings of the 10th ACM Multimedia Systems Conference*, 2019.
- [17] Z. Liu, H. Tang, Y. Lin, and S. Han, "Point-voxel cnn for efficient 3d deep learning," *arXiv preprint arXiv:1907.03739*, 2019.
- [18] V. Morell, S. Orts, M. Cazorla, and J. Garcia-Rodriguez, "Geometric 3d point cloud compression," *Pattern Recognition Letters*, 2014.
- [19] Y. Nievergelt, "Total least squares: State-of-the-art regression in numerical analysis," *SIAM review*, 1994.
- [20] T. Ochotta and D. Saupe, "Compression of point-based 3d models by shape-adaptive wavelet coding of multi-height fields," in *Proceedings of the First Eurographics Conference on Point-Based Graphics*, 2004.
- [21] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011.
- [22] S. Schwarz, M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. Chou, R. Cohen, M. Krivokuća, S. Lasserre, Z. Li, J. Llach, K. Mammou, R. Mekuria, O. Nakagami, E. Siahaan, A. Tabatabai, A. Tourapis, and V. Zakharchenko, "Emerging mpeg standards for point cloud compression," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2018.
- [23] J. Smith, G. Petrova, and S. Schaefer, "Progressive encoding and compression of surfaces generated from point cloud data," *Computers & Graphics*, 2012.
- [24] X. Sun, H. Ma, Y. Sun, and M. Liu, "A novel point cloud compression algorithm based on clustering," *IEEE Robotics and Automation Letters*, 2019.
- [25] D. Thanou, P. A. Chou, and P. Frossard, "Graph-based compression of dynamic 3d point cloud sequences," *IEEE Transactions on Image Processing*, 2016.
- [26] C. Tu, E. Takeuchi, A. Carballo, and K. Takeda, "Point cloud compression for 3d lidar sensor using recurrent neural network with residual blocks," 2019.
- [27] C. Tu, E. Takeuchi, C. Miyajima, and K. Takeda, "Compressing continuous point cloud data using image compression methods," 2016.
- [28] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the h.264/avc video coding standard," *IEEE Trans. Cir. and Sys. for Video Technol.*, 2003.
- [29] B. Wu, A. Wan, X. Yue, and K. Keutzer, "SqueezeSeg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.
- [30] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 629–642.
- [31] Y. Zhou and O. Tuzel, "Voxelnet: End-to-end learning for point cloud based 3d object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.