# Implementation of Compressed Point Cloud Streams in ROS

Vineeth Bandi
University of Texas at Austin
Austin, Texas 78712
Email: vineeth.bandi@utexas.edu

Nalin Mahajan
University of Texas at Austin
Austin, Texas 78712
Email: nalinmahajan@utexas.edu

***Abstract*—In this paper, we write and test a program that compresses point cloud data generated in real-time through integration with Robot Operating System (ROS). By splitting the data into its composite depth map and RGB image streams, we are able to separately compress these streams to massively reduce the amount of space required for storing the original point cloud data. We then provide a way of recreating the original point cloud from the depth map and RGB image. This allows for essentially lossless storage of point cloud data making transport more efficient and effective.**

**We found that we could reduce the original file by over 3300%, in some cases, through our approach. To accomplish this, we encoded the RGB image stream into a video file using H.264 and then utilized LZ4 frame compression for the depth map images. We then uncompressed each RGB and depth image frame by frame based on timing data and then generated a point cloud in real-time.**

## I. Introduction

Point cloud data is a useful way to represent 3D space or 3D models that can be generated in real time. Point clouds have been traditionally used for the 3D rendering of objects in animation or design. The 3D property of point clouds is especially useful in the field of robotics, providing a cheap and intuitive way to represent objects in 3D space and generate a spatial map. It is also finding use in other industries notably, Augmented Reality. Point clouds tend to produce a large amount of data making it unfeasible to store or transmit in bandwidth limited scenarios. In order to overcome this issue, data compression can be applied. Data compression trades memory usage for computational resources. Data compression is the process of encoding a file into a different format using fewer bits such that the original file's information is recoverable. This process is useful for a multitude of reasons including saving local disk space, transferring large files over the internet with bandwidth limits, or for meeting size limits on data uploads. In this paper, we will seek to implement our own compression algorithm for compressing point cloud streams in ROS.

### *A. Lossy versus Lossless Compression*

Compression can be lossy or lossless. Lossy compression algorithms sometimes eliminate data that is deemed unnecessary. The original data is unrecoverable as a result of this compression. Lossless compression, on the other hand, maintains the integrity of the file such that all of the bits that were removed can be recovered by a matching decompression algorithm. Often, lossy algorithms, although failing to preserve the original data, can have higher compression ratios than lossless compression and in some scenarios are less computationally taxing. Both lossy and lossless algorithms have been implemented successfully as a means to compress point cloud data. Hence, this paper will seek to compare and implement lossy and lossless compression data in ROS and evaluate the viability of lossy compression on point cloud data.

### *B. The University of Texas Building-Wide Intelligence Project*

The University of Texas Building-Wide Intelligence (BWI) lab uses ROS sensor data to test BWI robots. Point cloud data can be used to test how a robot would interact with certain inputs remotely or provide test scenarios for simulations. However, this type of sensor data is space-intensive. For example, a rosbag that stored point cloud data generated from a kinect sensor for less than a minute required approximately 12 gigabytes of space to store. For more advanced scenarios with multiple sensor streams running for longer periods of time, this data will grow to an unmanageable size. As a result research and testing to be conducted in the lab is hampered, as storage is more likely to fail due to a larger amount of writes as well as the time it takes to transfer large files either over network or local storage. In addition, the representation and real time analysis through RVIZ of point cloud data can start to become bandwidth constrained, creating difficulty in visualizing data. For both storing and transferring these data streams, compression is a solution to this problem. By reducing the size of sensor data, the BWI lab can store more data and transfer the data at a faster rate, which is especially important as more work is being conducted remotely. Currently, ROS contains some tools for compressing certain types of ROS sensor messages and images, but it lacks a standardized functionality for compressing point cloud streams which contributes significantly to the size of sensor streams. We are hoping that, through this paper, we can provide a solution and implementation that can help to streamline the process of compressing point cloud streams without compromising on testing validity.

## II. Background

3D point cloud compression algorithms already exist and are implemented in a variety of ways, both lossy and lossless, yet there isn't a true standard for compression. The push for a standard compression system has been bolstered recently as point cloud data is being heavily utilized in projects focusing on Augmented Reality. Traditionally, these implementations are targeted at computationally weak and memory strained mobile devices justifying the need for a data compression standard. The Moving Picture Experts Group (MPEG) has been leading this push for standardization through MPEG-3D, their 3D graphics division. Under consideration are 2 algorithms: video-based point cloud compression (V-PCC) and Geometry-based point cloud compression (G-PCC). V-PCC is designed for media and utilizes 2D video compression as a way to compress and then generate point clouds. The already prevalent use and implementation of 2D video specific hardware accelerated encoders makes it easily deployable and already fairly optimized. However, V-PCC can produce artifacts and mitigation algorithms must also be considered.[1] G-PCC sidesteps utilizing 2d space directly by describing points in 3D space. Currently these standards do not implement prediction tools, but other technologies seek to extend both of these standards with the use of predictive techniques utilizing neural networks to filter out and decompress unused data. This has been attempted using a variety of methods including the use of recurrent neural networks in Tu et al. [2]. Other external organizations have implemented novel solutions such as Asymmetric Numeral systems in Google's Draco library, 1D traversal compression, projection and mapping onto 2D domains, and direct exploitation of 3D correlations have been tested using multiple algorithms. [3] These compression algorithms are often made in mind to compress a fully 3D point cloud such as 3D model where all sides of the model are relevant, however, in certain cases there is only one point of view. This makes comprehensive compression algorithms more expensive. Currently there is no standardized or widely popular 2d based compression algorithm for point cloud data that would be highly advantageous for robots where a point cloud is generated from one point of view.

## III. Method

We investigated the compression of point cloud data in terms of efficiency and performance as they relate to ROS point cloud data. The point cloud data worked on by the lab is often generated by a Kinect style device which generates an RGB image and a depth map in addition to a point cloud. We wrote a node that subscribes to the rectified RGB image and depth map sensor data. The ROS Node then utilizes OpenCV's VideoWriter class to compress the RGB image using a user specified encoding such as FFmpeg. Concurrently, the depth map is compressed using LZ4 frame compression and the output of both of these is stored locally in two separate files. Figure 1 shows an abstracted visualization of this process. We then wrote another node that accepts the two compressed files and generates the original RGB and depth frame which is published through ROS. Another node then reconstructs the point cloud from the given RGB image and depth map. The node then broadcasts the reconstructed point cloud to a topic. This simulates rosbag usage where the rosbag is a recording of selected topics. Then, when ran, it publishes the recorded data to those same topics allowing users to run simulations that utilize point cloud data. Figure 2 shows an abstracted visualization of this process.

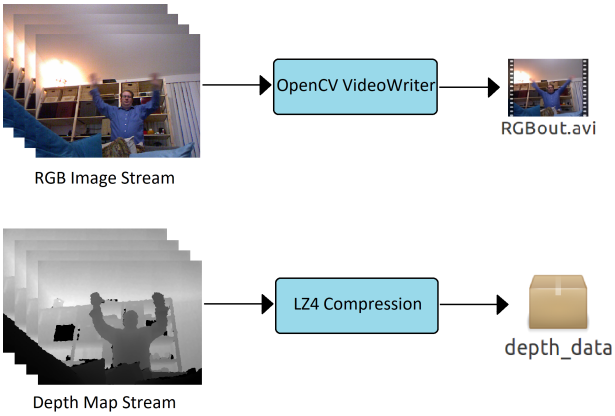


Fig. 1. Abstracted sample workflow of our compression node.

### A. RGB Image Compression

To handle the compression of the RGB image stream, we pass our stream frame by frame into an OpenCV VideoWriter which accepts a file output, a FourCC code specifying the codec, video frame rate, and video dimensions. The videowriter will then write each frame into a video file with the specified encoding. This functionality allows us to test different codecs and their effect on file size. Since some codecs are lossy, we can also test to see if we can still reconstruct a point cloud using a lossy compression. The current implementation allows us to reliably utilize FFmpeg, Huffman, MPEG-4, H.264 and MJPEG video encoding, newer video codecs such as H.265 or VP9 are not properly supported in OpenCV's VideoWriter class. This implementation is uniquely applied to RGB image data since most codecs support up to 8 bit, 3 channel image encoding, whereas depth maps that utilize 32 bit encoding are widely unsupported. When recreating the point cloud, we don't decompress the video but rather directly map the frame whether lossy or lossless and publish it to a topic.

### B. Depth Map Compression and LZ4

For compressing the depth map, we use LZ4 frame compression. We cannot use VideoWriter with a relevant encoder as it does not natively support single precision, single channel, floating point encoding. LZ4 compression is a form of lossless compression that works on all file types as it directly encodes byte streams. LZ4 is in the family of LZ77 compression algorithms and is focused on fast and low computation decompression. LZ4 in this regard sacrifices compression for speed.

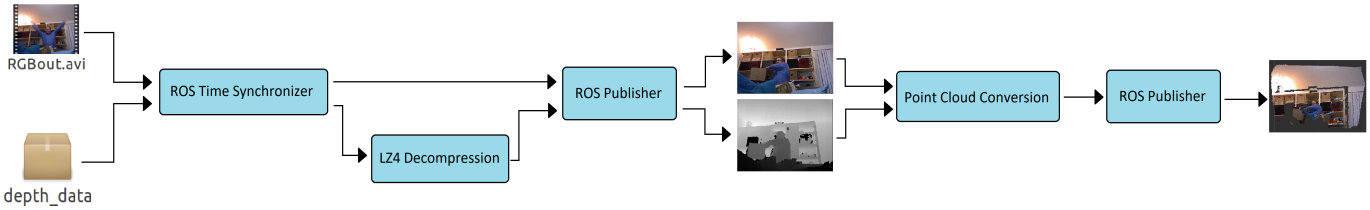


Fig. 2. Abstracted sample workflow of our broadcast node.

This makes LZ4 a great candidate for decompressing data that must be worked with in real-time or in small time frames. This is important for decompressing point cloud data as the depth image and RGB images should match the time that they were originally broadcast at to prevent synchronization issues. We are specifically using LZ4 frame compression which is a form of LZ4 compression that allows the addition of separate compressed frames of data. The frames are made of a frame descriptor, data blocks, end mark, and a checksum. The frame descriptor details the size of the frame and different aspects of how the frame is to be decompressed. The compressed data is stored in separate blocks some dependent on each other. The frame concludes the data blocks with an end mark allowing for compression and decompression with variable sizes. The checksum provides a robust check to ensure that data is compressed or decompressed correctly.[4] The efficient lossless compression and more impressive speed of this type of compression along with its frame compression that meshes well with storing depth map frames made it an ideal way to compress depth map data. LZ4 compression is implemented in the node by first subscribing to a depth Image topic and converting the image into an openCV matrix with type single precision floating point data. From here the matrix data can be converted into a byte array and stored into a buffer. An LZ4 frame is then generated by specifying the number of bytes to be compressed and placed into an output buffer which is then written to a file sequentially. This forms an LZ4 compressed file containing multiple frames each representing the sequential data of an OpenCV image matrix in the form a byte array. This file can then be read and decompressed by another node.

### C. Synchronization

A side affect of splitting the data into its RGB video and depth map is that the timing of the data is no longer preserved. This is important for real time and simulation applications as the depth map and RGB image need to processed together. In order to circumvent this problem, we used a ROS Time Synchronizer which queued up to 5 frames of a data and then ran a callback once it received an RGB image and depth map with matching times. The callback would then compress both the RGB and depth map and wait for the next synced frames to compress. For decompression, we must also ensure that both images are published in the same time frame at the same rate. To do so we specified a ROS rate based on the FPS of our recorded RGB video. Within this time frame the image would be decompressed and published and the next image queued up for publishing at the specified rate. For instance a video captured at 30 FPS requires that the node be run at a rate of 30 Hz. This means that the RGB image frame and depth map frame must be decompressed and broadcast within the 1/30 of a second allotted. This is especially important for choosing a compression algorithms for the depth frames since it must be able to decompress one frame quickly to meet the required broadcast rate, hence the use of LZ4.

### D. RGB Image and Depth Map Decompression

In order to retrieve the original RGB Image and depth map we must decompress the data we compressed originally. The compressed data is specified as two files which are passed to the decompression node. This node uses an OpenCV VideoReader to read the encoded video frame by frame while a file stream is opened with the depth map data in order to read the file as a byte stream. Based on the synchronizing solution mentioned earlier we read one frame of the RGB video and simultaneously decompress one frame of the LZ4 compressed depth map. LZ4 frame decompression is implemented by originally reading parts of the file to a byte array since we do not know the size of each LZ4 frame. Then we can specify the number of decompressed bytes we most likely need. The LZ4 frame decompression method then reads in the specified number of bytes and decompresses it. If the number of bytes read is not enough to fully decompress one frame the method will output a guess as to how many bytes more must be read from the source file. We then repeat this using the guessed number of bytes until the frame is fully decompressed. From here we can generate an OpenCV matrix using the byte array and broadcast it as a depth map alongside the RGB Image.

### E. Recreating the Point Cloud

The final step in this process now that we have the published the depth map and RGB image to their respective topics, is to publish the generated point cloud. To accomplish this, another node subscribes to the published decompressed RGB Image, depth map and user specified camera information topic. The point cloud node populates an array representing each point cloud combining the depth of the object computed using the depth map and overlaying it on the RGB data for that specific point. The points are then stretched and placed based on the optical center of the camera and its focal length. Note that in our implementation the depth map is computed in terms of millimeters and must be scaled accordingly to meters. After

recreating the point cloud, we have all the relevant pieces and running our node publishes all the relevant data from the compressed depth map and RGB video file.

## IV. Evaluation

To evaluate our solution, we compared the total size of the compressed depth map and compressed RGB image streams with a rosbag containing only point cloud data. Since all compression and decompression occurs withing the 30 Hz window of the sensor stream data, there is no use measuring compression or decompression time. In addition to comparing sizes, we sought to see the effect different encoders for RGB image compression had on the reconstruction of the original point cloud. As such we tested our source data with five different encoders including both lossy and lossless encoders. The encoders used are FFmpeg, MJPEG, Huffman, H.264, and MPEG-4. For the purposes of depth map compression, we used LZ4 so that compression and decompression could be close to real-time and there wouldn't be a need to wait after transferring compressed files.

We then compared the compressed file size of the RGB video and compared it to the original rosbag. We then calculated the compression ratio of each encoder to provide a standard metric for comparison. We then chose the best lossless and lossy encoder and evaluated the total compression ratio compared to our original point cloud rosbag. These experiments were conducted for two separate rosbags. The compression ratio is calculated using the formula listed below and is the average of 3 runs to ensure consistency.

$$\text{Compression Ratio} = \frac{\text{Original File Size}}{\text{Compressed File Size}} \tag{1}$$

## V. Results

The results of different encoders on total RGB image compression can be seen in Table I. Since we used a variety of lossy and lossless encoders, some of the results heavily favor the lossy encoders. All of the encoders were tested using the same image stream. File size represents the size of the compressed video file. While we got the best results from H.264, its important to note that this is a lossy codec. It is still uncertain whether a lossy compression has an inadvertent effect on the usage of the reconstructed point cloud. If the integrity of the color portion of the point cloud needs to be maintained, a lossless encoder should be used. The data suggests that, for the purposes of lossless compression, FFmpeg should be utilized.

TABLE I
RGB Image encoders

| Encoding | File 1 Size | File 2 Size | Avg Compression Ratio |
|---|---|---|---|
| None | 735.7 MB | 281.2 MB | 1 |
| Huffman | 250.1 MB | 79.3 MB | 3.2438 |
| FFmpeg | 143.6 MB | 45.6 MB | 5.64498 |
| MJPEG | 78.8 MB | 25.4 MB | 10.2036 |
| MPEG-4 | 14.6 MB | 5.0 MB | 53.3152 |
| H.264 | 3.4 MB | 1.2 MB | 225.3580 |

The lossy compression algorithms feature the highest compression ratio's by a substantial margin, but one thing to consider is whether the point cloud is still usable for the purpose that the original would be. At the time of writing, our tests on this were inconclusive. It also worth considering that the RGB image is a relatively small portion of the total size of the original rosbag which means that savings here are not as significant, making the choice of lossy or lossless encoding only provide marginal savings.

TABLE II
Depth Map Compression

| Encoding | File 1 Size | File 2 Size | Avg Compression Ratio |
|---|---|---|---|
| None | 979.6 MB | 206.5 MB | 1 |
| LZ4 | 189.4 MB | 55.2 MB | 4.4565 |

Table II shows the results of our depth map compression using LZ4. The other substantial part of our savings comes from compressing the depth map. In our tests, we achieved an average compression ratio of 4.4565, which is comparable to the FFmpeg encoder's compression ratio. While this isn't as drastic as the savings of lossy image compression, one thing to consider is that LZ4 is a lossless compression algorithm. Although, other lossless compression algorithms can achieve better compression ratios LZ4 is unique in that it has a low compression and decompression time which is important for being able to hit refresh rate targets. Further works can investigate other compatible compression algorithms for more saving or to support higher frame rates.

TABLE III
H.264 Point Cloud Compression Sizes

| File | File Size | Compressed Size | Compression Ratio |
|---|---|---|---|
| 1 | 6399.9 MB | 192.8 MB | 33.1945 |
| 2 | 1435.3 MB | 56.4 MB | 25.4486 |

TABLE IV
FFmpeg Point Cloud Compression Sizes

| File | File Size | Compressed Size | Compression Ratio |
|---|---|---|---|
| 1 | 6399.9 MB | 333 MB | 19.2189 |
| 2 | 1435.3 MB | 100.8 MB | 14.2391 |

Tables III and IV show our aggregate results when considering FFmpeg and H.264 for RGB image compression and LZ4 for depth map compression, respectively. The original file size is the size of just the original point cloud data from the rosbag and the compressed size is the aggregate of data needed to reconstruct the point cloud. Compression ratio is calculated as before. This data shows that using a lossy RGB encoder, specifically H.264, we were able to achieve a 25-33 compression ratio which is a substantial decrease in required file size as a 6GB file now only required 192.8 MB. We hypothesize that this could scale up with larger data files providing even greater space efficiency. A lossless encoder

only achieves a 14-19 compression ratio which is much lower than lossy compression. However, compared to the original file size, there is still substantial space savings as a 6GB file can be compressed into a 333 MB file.

## VI. Discussion

Our findings suggest that our method is a feasible approach for the BWI lab to compress point cloud streams with a high space efficiency. One major difference between our approach and other state-of-the-art algorithms, is that ours records and publishes the data in real-time. This allows for multiple different uses. One example being where a point cloud is generated by one sensor, and then quickly sent to a server for testing or logging. This compression algorithm could be modified for even greater gains by exploring other compression algorithms for depth map data and more efficient lossless video encoders. In addition, the affect of lossy RGB video may be minimal for different projects although we were unable to confirm whether or not this had any effect on the validity of the generated point cloud for program testing. Further tests for measuring artifacting could include Mean Square Error (MSE) and other similar methods that are effective on RGB images. An extension of this algorithm could include artifact mitigation algorithms to ensure that a lossy encoding can be used for the lab's purposes.

## VII. Conclusion

To improve the BWI Lab's ability to use ROS sensor data, we realized and implemented a method to compress point cloud data. The compression algorithm splits a point cloud into an RGB image and a depth map. The RGB images were encoding into a viewable video file allowing for flexible encoder types and better compression. The depth map was compressed using the LZ4 frame format allowing for a fast compress and decompress time in order to hit frame rate targets while maintaining lossless compression. This data was then synced and then composed in order to reconstruct the original point cloud in a mostly lossless fashion. This compression algorithm allowed point cloud data specifically to be successfully compressed with compression ratios ranging from 15-30 providing close to a 20 fold decrease in file size.

## References


[1] D. Graziosi, O. Nakagami, S. Kuma, A. Zaghetto, T. Suzuki, and A. Tabatabai, "An overview of ongoing point cloud compression standardization activities: video-based (v-pcc) and geometry-based (g-pcc)," *APSIPA Transactions on Signal and Information Processing*, vol. 9, p. e13, 2020.

[2] C. Tu, E. Takeuchi, A. Carballo, and K. Takeda, "Point cloud compression for 3d lidar sensor using recurrent neural network with residual blocks," 05 2019.

[3] C. Cao, M. Preda, and T. Zaharia, "Real-time decoding and ar playback of the emerging mpeg video-based point cloud compression standard," in *The 24th International Conference on 3D Web Technology*, ser. Web3D '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–9. [Online]. Available: https://doi.org/10.1145/3329714.3338130

[4] lz4, "lz4," https://github.com/lz4/lz4, 2020.