A GPU-Enabled Real-Time Framework for Compressing and Rendering Volumetric Videos

Dongxiao Yu, *Senior Member, IEEE*, Ruopeng Chen, Xin Li, Mengbai Xiao, Guanghui Zhang, and Yao Liu

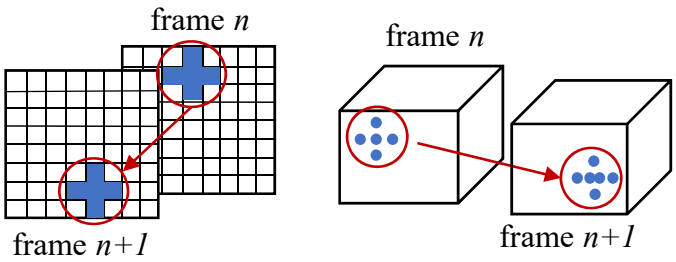
Abstract—Nowadays, volumetric videos have emerged as an attractive multimedia application providing highly immersive watching experiences since viewers could adjust their viewports at 6 degrees-of-freedom. However, the point cloud frames composing the video are prohibitively large, and effective compression techniques should be developed. There are two classes of compression methods. One suggests exploiting the conventional video codecs (2D-based methods) and the other proposes to compress the points in 3D space directly (3D-based methods). Though the 3D-based methods feature fast coding speeds, their compression ratios are low since the failure of leveraging inter-frame redundancy. To resolve this problem, we design a patch-wise compression framework working in the 3D space. Specifically, we search rigid moves of patches via the iterative closest point algorithm and construct a common geometric structure, which is followed by color compensation. We implement our decoder on a GPU platform so that real-time decoding and rendering are realized. We compare our method with GROOT, the state-of-the-art 3D-based compression method, and it reduces the bitrate by up to 5.98×. Moreover, by trimming invisible content, our scheme achieves comparable bandwidth demand of V-PCC, the representative 2D-based method, in FoV-adaptive streaming. The code is available at https://github.com/ChenRP07/patchVVC-plus.

Index Terms—volumetric video, point cloud compression

1 Introduction

The recent advancements in virtual reality (VR) and augmented reality (AR) technologies have catalyzed the emergence of a novel video format known as volumetric video. This type of video enables users to navigate freely within a rendered scene, allowing for both translation and rotation of the viewport. By offering an exceptionally immersive viewing experience, the volumetric video holds the potential to revolutionize industries such as filmmaking[1] and interactive advertising.[2] According to a recently published report,[3] the market for volumetric videos is projected to reach $4.9 billion by 2026.

However, the size of volumetric videos presents a daunting challenge when it comes to streaming and storage. A volumetric video consists of temporally continuous point clouds, which are obtained using LiDARs [1] and RGB-D cameras [2] or are reconstructed from 2D images [3]. A point is represented by the coordinates in $\langle x, y, z\rangle$, the color channels $\langle r, g, b\rangle$, and other attributes like normal and curvature. The coordinates and the colors are required, and the other attributes are optional. In the volumetric video, only adding millions of such points to a point cloud (PC) frame and recording 30 frames in a second could ensure that the reconstructed scene is immersive and realistic enough. But this imposes a bandwidth overhead of over 1 Gbps to stream the video, which can hardly be provisioned by the current wireless network infrastructure[4].



(a) A 2D video (b) A volumetric video

Fig. 1: The difference between searching similar pixels in 2D frames and searching similar points in 3D frames. (a) The 2D frames have the same grid structure, easing the search process. (b) The 3D frames neither dictate where the points appear nor require the point number to be the same between frames.

To effectively reduce the size of volumetric videos, developing compression techniques for point clouds is promising. Researchers have explored two primary approaches: 3D-based methods and 2D-based methods. The 3D-based methods leverage 3D data structures [4]–[7], such as octrees or kd-trees, to directly compress the point cloud. These methods typically construct a compact tree that geometrically represents all points in a frame and then compresses the positional residuals using an entropy encoder. Though with satisfactory speeds in encoding and decoding, these methods generally have low compression ratios. On the other hand, the most representative 2D-based method is

D. Yu, R. Chen, X. Li, M. Xiao, and G. Zhang are with the School of Computer Science and Technology, Shandong University, Qingdao, Shandong, China (Corresponding author is M. Xiao)
Y. Liu is with the Department of Electrical & Computer Engineering, School of Engineering, Rutgers University, New Brunswick, New Jersey, USA

1. https://www.youtube.com/watch?v=iwUkbi4_wWo
2. https://www.anayi.com/include_html/4d/21ss_4d_11.html
3. https://www.marketsandmarkets.com/Market-Reports/volumetric-video-market-259585041.html
4. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

V-PCC [8]. V-PCC compresses a point cloud by projecting its points onto a 2D plane, which is further encoded by conventional video codecs like H.264 [9] or HEVC [10]. By exploiting these well-developed codecs, the rate of bitstream after compression reaches a level substantially lower than the 3D-based methods. However, the compression pipeline of these methods is complex and slow due to the projection and backprojection introduced. Furthermore, the 2D-based methods merge different parts of a point cloud in a 2D plane, making it hard to be integrated into a streaming system delivering visible content only [11], which is another promising technique for saving bandwidth [12]. In such a field-of-view (FoV) adaptive streaming session, the encoder has to recognize the visible parts on-the-fly, rearrange the points on the 2D planes, and re-compress the frames.

Though the 3D-based methods are fast, they can hardly compress the videos effectively because they fail to identify and eliminate inter-frame redundancy in the 3D space. Migrating the 2D motion compensation techniques to the 3D space is not straightforward. Between different frames, the point numbers are not necessarily the same, and the point positions are not inherently aligned. We compare how to search similar pixels of 2D videos and search similar points of volumetric videos in Fig. 1.

In this paper, we extend our prior work patchVVC [13] to patchVVC+. As a 3D-based patch-wise compression framework, our method has fast decoding speed and it also features high compression ratios. A preprocessing step that splits the point cloud frames into patches is set so that the visible ones could be recognized and grouped in the compression domain trivially. This approach is particularly advantageous for FoV-adaptive streaming systems. Compared to the prior work, patchVVC+ segments frames to smaller patches, changes fix-sized GOP to that dynamically determined, generates patch groups in finer granularity, and decides entropy coding adaptively. These techniques improve the compression ratio, but the coding speed is also slowed. We then migrate the decoder to a GPU platform, accelerating the decoding procedure. A rendering module is additionally realized on the GPU so that the volumetric videos could be played in real-time in our framework.

There are three stages in the encoder, *patch segmentation*, *patch deformation*, and *data compression*. In the patch segmentation stage, the PC frames are split into patches by k-means clustering [14]. The centroids of patches of a frame are used as the seeds that cluster the next frame. The patch size is balanced by splitting large ones and merging small ones. In the patch deformation stage, patches from different frames are grouped by the index of their centroids, forming patch groups. The size of patch groups is determined dynamically and we also iteratively construct a common patch that represents the common geometric structure of patches in the group. The common patch is colored differently by distance-weighted interpolation for different patches, which makes the following color compensation straightforwardly. We start a new patch group if a patch is found to deviate from the common patch. In the data compression stage, each patch is organized as a slice. We adopt an entropy encoder to adaptively encode the octrees and the metadata generated in the previous steps. For the colors and color residuals, we employ region-adaptive hierarchical transform (RAHT) [15] to compress them.

In the evaluation, we compress the videos of the 8i dataset [16] with various methods, and the results show that patchVVC+ compresses the videos effectively. The compressed videos are up to 4.24× smaller than MP3DG [17] and up to 5.98× smaller than GROOT [7]. In the FoV-adaptive streaming scenarios, patchVVC+ demands a similar or even less bandwidth than V-PCC [8], the representative 2D-based method. Our framework implements a decoding-rendering pipeline on GPU and plays volumetric videos at more than 30 FPS, which is 20× faster than V-PCC and MP3DG.

We highlight our contributions as follows:

- We effectively exploit the inter-frame redundancy in the 3D space for volumetric videos compression, achieving higher compression ratios than the state-of-the-art 3D-based compression methods.
- Our design is parallel-friendly and has a fast decoding speed. We implement a decoding-rendering pipeline on GPU that justifies our design, which plays volumetric videos with frames each having millions of points at more than 30 FPS.
- We carry out extensive experiments to evaluate our method, whose compression ratio approaches the 2D-based methods and the decoding speed is similar to the 3D-based methods.

In the rest of the paper, Section 2 first discusses the related work. Section 3 then briefly introduces background knowledge. We present details of the decoder and decoder design in Section 4 and Section 5, respectively. Section 6 reports the experimental results and Section 7 concludes our work.

2 RELATED WORK

2.1 Point Cloud Compression

To compress point cloud videos with traditional video codecs, one has to project 3D points onto 2D planes as the frame is fed to the encoder. Following this idea, V-PCC [8] is proposed as the MPEG standard. V-PCC could be further improved by introducing additional motion vectors that are discovered in the 3D space [18]. Since the pixels in the projected plane are not fully occupied, the quality of these unused pixels could be sacrificed to further reduce the size of compressed videos [19]. Since the FoV of viewers is limited, view-PCC [20] performs clustering based on distances and projects points from different views. This makes FoV-adaptive streaming become feasible. Though developing compression methods based on the traditional codecs achieves high compression ratios, the additional complexity introduced to the pipeline also leads to slow coding speeds.

On the other hand, a point cloud could be represented by a compact octree, whose size is further reduced by entropy encoding [4], [6]. GROOT [7] breaks the dependencies between nodes in the last three layers of an octree and accelerates the decoding procedure on GPUs. In G-PCC [8], another MPEG standard designed to compress static point clouds, the points are arranged into non-overlapping blocks that are encoded and decoded in parallel. As for the volumetric videos compression, the efforts of compensating

points in 3D space have also been made in a series of methods [17], [21]–[23]. However, they only carry out block-based compensation, and this will fail once the points move across the block boundaries. Additionally, the 3D-based methods commonly project and compress point colors as 2D images [7], [17], which fails to exploit their correlation in 3D space. To address this, RAHT modifies the octree construction and incorporates the colors into it, gaining both a fast compression speed and a high compression ratio. Another line of research focuses on compressing volumetric video using graph-based transformations [24], [25], but these methods suffer from high computational overhead.

2.2 Volumetric Video Streaming

The research direction of volume video streaming is mainly focused on point cloud content sampling and user viewport interaction. This differs from the optimization direction of traditional video streaming in areas such as network protocol [26], content distribution [27], and edge computing [28]. DASH-PC [29] utilizes spatial sub-sampling and creates manifests in a point cloud-specific manner, which is especially helpful for dense point clouds. PCC DASH [30] controls the rate adaptation based on both the buffer status and the user behavior. An alternative approach [31] segments point cloud frames into tiles that are independently decodable so that the FoV-adaptive streaming becomes possible. In contrast to queue buffer approaches, certain tile-based methods [32] employ a window buffer to manage frequent user interactions. Such a method does not queue the data sequentially and allows for responsive handling of unexpected user interactions. Precisely predicting the FoV of users also helps improve volumetric video streaming services [33], [34], which significantly reduces motion-to-photon latency. To enable practical streaming of volumetric video on mobile devices, Nebula [35] offloads video decoding to edge servers, where the 3D points are transcoded to view-constrained frames. Another streaming system, YuZu [36], employs a point cloud upsampling model to save bandwidth.

3 Background

3.1 Iterative Closest Point

Given any two point clouds (one as the source and the other as the target), the iterative closest point (ICP) [37] algorithm extracts a transformation matrix representing rigid moves in translations and rotations. Following the transformation, the source point cloud could best fit the target one. To extract the transformation matrix, there are four steps: 1) searching the nearest points, 2) computing the transformation, 3) applying the registration, and 4) judging the convergence. A rotation vector $\vec{q_r} = (q_0, q_1, q_2, q_3)^T$ represented by the quaternion and a translation vector $\vec{q_t} = (q_4, q_5, q_6)^T$ are iteratively computed to minimize the point-wise distances between two point clouds. The transformation matrix computed in the k-th iteration is

$$M_r^k = \begin{bmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) & q_4 \\ 2(q_1q_2+q_0q_3) & q_0^2+q_2^2-q_1^2-q_3^2 & 2(q_2q_3-q_0q_1) & q_5 \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & q_0^2+q_3^2-q_1^2-q_2^2 & q_6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

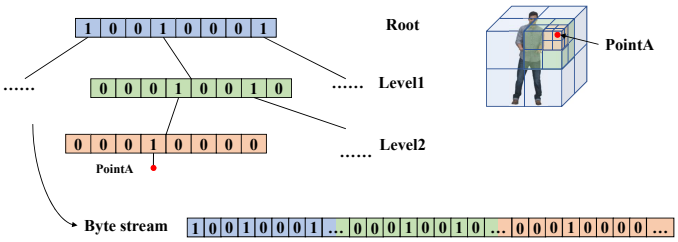


Fig. 2: A point cloud is represented by an octree, which has three levels and discards all the positional residuals.

. After K iterations, we calculate $M_r = M_r^K \cdot M_r^{K-1} \cdot \ldots M_r^1$ as the overall transformation matrix. Transforming the source point cloud $\mathbf{P}_s$ is calculated by $\mathbf{P}'_s = M_r \cdot \mathbf{P}_s$, where $\mathbf{P}'_s$ is the transformed point cloud.

3.2 Octree-based Point Cloud Compression

A point cloud could be represented by an octree [4], which is more compact than the Cartesian coordinates. In an octree, the root node is the entire space, which is recursively split into eight subspaces until no points are involved or we reach a predefined level. An intermediate node of octree has at most eight children, enabling efficient storage in a single byte. In a byte of a tree node, 1 indicates the corresponding subspace contains points and 0 means the opposite. In the end, we do not need the point coordinates anymore since the center of a leaf node serves as the approximate position of the points in it. The precise coordinates of points are the sum of the leaf center and the positional residuals. In practice, an octree is commonly organized into a byte stream with breadth-first search traversal. Fig. 2 illustrates how we cloud use an octree to represent a point cloud and how we use a byte stream to store the octree.

3.3 RAHT

RAHT [15] exploits Haar wavelet transform to effectively compress point cloud attributes like colors. RAHT treats its input as a static point cloud. The input point cloud is first divided into blocks and each constructs an octree itself. The tree nodes are then merged along the x-, y-, and z-axis in a bottom-up manner. Merging tree nodes along the x-axis follows

$$\begin{bmatrix} g_{l-1,x,y,z} \\ h_{l-1,x,y,z} \end{bmatrix} = \frac{1}{\sqrt{w_1+w_2}} \begin{bmatrix} \sqrt{w_1} & \sqrt{w_2} \\ -\sqrt{w_2} & \sqrt{w_1} \end{bmatrix} \begin{bmatrix} g_{l,2x,y,z} \\ g_{l,2x+1,y,z} \end{bmatrix}$$

, where for a position $\langle x, y, z \rangle$ and a layer l, $g_{l,x,y,z}$ and $h_{l,x,y,z}$ means the corresponding attribute and coefficient, respectively. In the tree nodes with $g_{l,2x,y,z}$ and $g_{l,2x+1,y,z}$, their point numbers are defined as w_1 and w_2. After merging, $g_{0,0,0,0}$, the root attribute, and $h_{l,x,y,z}$ are separated into sub-bands. The sub-bands are further quantized and encoded by an arithmetic codec.

4 Encoder

The input of our encoder is a sequence of PC frames, and the output is a bitstream compressed. The encoder compresses the PC frames in groups of pictures (GOPs). A GOP is composed of an I-frame and the other P-frames. The size

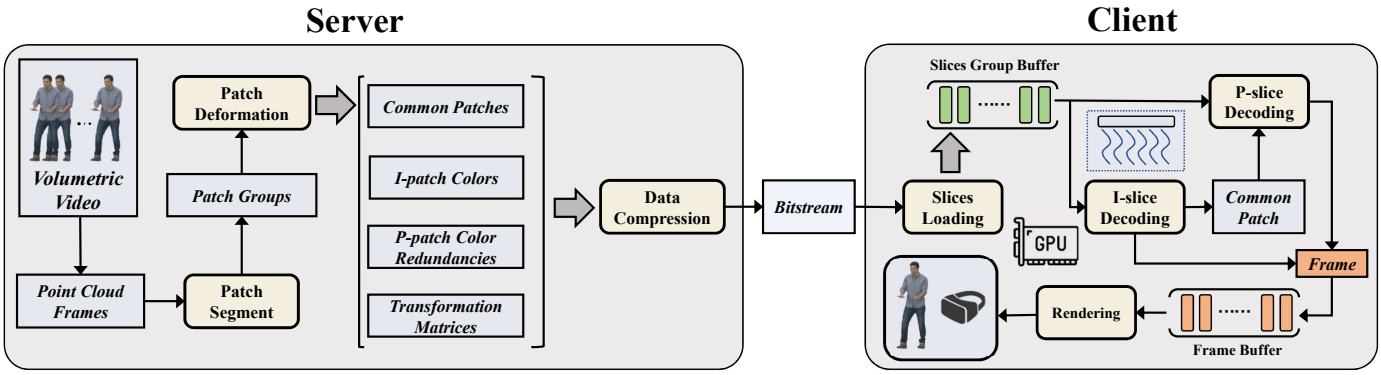


Fig. 3: The overview of patchVVC+

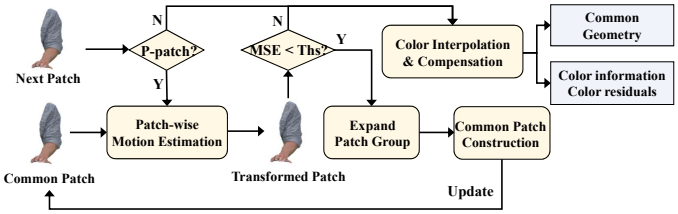


Fig. 4: The pipeline of patch deformation

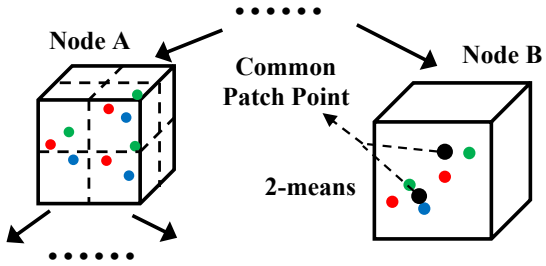


Fig. 5: An example of generating the common patch. Points from different frames (the red, blue, and green ones) form a dense point cloud and we build an octree over it. Node A is an intermediate node and Node B is a leaf node. To generate the common patch points (black ones), we perform k-means in the leaf nodes.

of GOPs varies. The pipeline of compressing a GOP has three phases: *patch segmentation*, *patch deformation*, and *data compression*. The overview of our encoding pipeline is shown on the left side of Fig. 3.

4.1 Patch Segmentation

Our method compresses a PC frame sequence GOP by GOP. In patch segmentation, we determine the current GOP size and segment inside PC frames into patches for capturing subtle movements like a person walking forward with her hand swinging backward. We consider the first of the remaining frame sequence as the I-frame of the current GOP. The I-frame is segmented into patches, namely *I-patches*. Then, we iteratively segment the next frame into patches, and based on these patches, we test if the frame could be added to the current GOP as a P-frame. The patches of a P-frame are *P-patches*.

4.1.1 I-frame Segmentation

For an I-frame, we generate its I-patches with the k-means clustering algorithm [14]. The initial centroids are evenly distributed in the point cloud [38]: 1) The point cloud is split into 8×8×8 blocks. 2) For each block, we split it into halves at edges having the maximum span as long as it contains points more than a threshold T. This is recursive until we have all blocks having points less than T. 3) We calculate a centroid from the points in a block and the centroids of all blocks are the k inputs to the k-means algorithm. After clustering, we have k patches.

However, the size of a patch, i.e., the number of points contained, varies significantly in practice. To balance the point number among patches so that their encoding and decoding workloads could be more friendly to the parallel architecture, we attempt to divide large patches and merge small patches. Specifically, a patch is represented by a block whose boundaries are determined by the maximum and minimum coordinates along the x-, y-, and z-axis. For a large patch with points more than T, we recursively cut the block into halves at the longest edges until the points in new blocks are less than T. We also test if a block could be merged with one of its neighbors, and the merged block still has its points less than T. By this means, the I-patches are adjusted to have approximate T points.

4.1.2 P-frame Segmentation

To test if a P-frame could be involved in the current GOP, we need first segment it into P-patches. The P-patches are also generated with the k-means algorithm. We use the patch centroids from the previous frame as the inputs of k-means, and the clustering results are P-patches.

We only involve frames with similar patches in a GOP otherwise exploiting inter-frame similarity becomes infeasible. In our design, we test if the newly generated patches are similar to the ones previously generated. If any of the patches are not similar, we terminate the current GOP and the current frame becomes the I-frame of the next GOP.

Since the patches are generated based on the patch centroids in the previous frame, the patches of the two frames are inherently paired. For each pair, we test if the new patch would fail any of the following two conditions: 1) The patch has a similar size, i.e., its size is less than $2\times T$, and 2) the patch is similar to its predecessor.

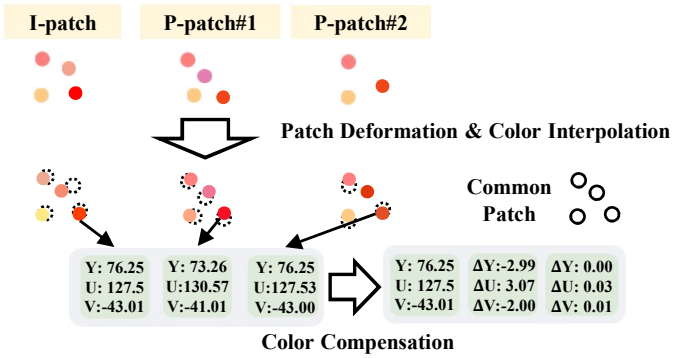


Fig. 6: An example of color interpolation and compensation

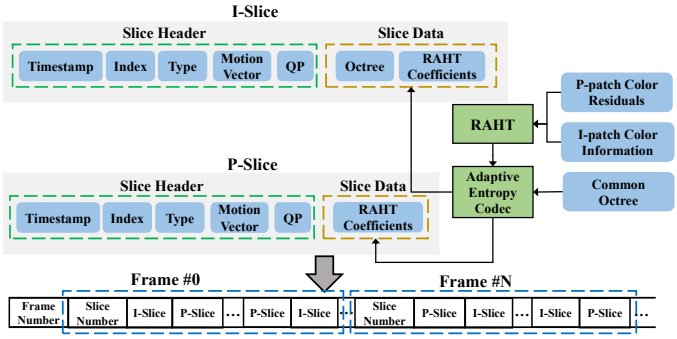


Fig. 7: The overview of the data compression stage

The similarity of two patches is defined by their mean squared error (MSE), which is computed as

$$\mathrm{MSE}(\mathbf{P},\mathbf{Q}) = \max(f(\mathbf{P},\mathbf{Q}), f(\mathbf{Q},\mathbf{P})),$$

where

$$f(\mathbf{P},\mathbf{Q}) = \sum_{p\in\mathbf{P}} \min_{q\in\mathbf{Q}} \|p-q\|$$

is the distance from $\mathbf{P}$ to $\mathbf{Q}$. The newly generated P-patch is considered not similar to its predecessor if the MSE is found large.

As a result, we compress the video in GOPs having varying sizes. The more similar the consecutive PC frames are, the larger the GOP size is. With more P-frames, we could achieve higher compression ratios by extracting their inter-frame redundancy.

4.2 Patch Deformation

To extract the inter-frame redundancy, we further organize patches in *patch groups*. Inside a patch group, we deform patches into a *common patch* that represents their geometric similarity, and with the common patch, we perform *color interpolation and compensation* that eliminate redundancy in the color domain. Fig. 4 shows an overview of patch deformation.

4.2.1 Patch group generation

We separate patches of a GOP into patch groups as the basic coding unit. In a GOP, we inherently have k initial patch groups, where k is the number of I-patches. The patches of an initial group are from different frames. We further temporally subdivide the initial patch groups that guarantee the inter-frame changes in a group are constrained.

Common patch construction. To construct the common patch of several patches, we first merge the patches (transformed). Then, we construct an octree for the merged point cloud, and its leaf nodes must contain points from all patches. This ensures that the leaf nodes have information from all frames. Once the octree has been built, we perform k-means for each leaf node, and the number of clusters is decided by the average point number in the node. Fig. 5 shows an example that 2-means is carried out in a leaf node containing 1, 2, and 2 points from three frames. After clustering in all leaf nodes, the centroids form the complete common patch. It is worth noting that the aforementioned octree is temporally built and discarded after common patch construction. Another octree will be used to represent the common patch in the following compression.

Patch-wise motion estimation. We measure the MSE between the common patch and the tested patch to determine if they are similar, but before this, we need to carry out patch-wise motion estimation to align the two patches. We first align the centroids of two patches, which results in a translation vector. Then, the tested patch is translated and used as the source of ICP while the target is the common patch. After the ICP algorithm, a transformation matrix is found to describe the rigid move that matches two patches. The MSE between the transformed tested patch and the common patch is calculated and used to decide if this patch should be added to the current patch group.

Patch group generation. Starting from the second patch of an initial patch group, we iteratively test if a new patch deviates from the common patch representing the common geometric structure of patches that have been tested. In the beginning, the common patch is the first patch. If the patch deviates from the common patch, all previous patches are packed as a patch group and we keep testing the remaining patches. Otherwise, the common patch is updated by involving the newly tested patch and the algorithm continues. In the end, an initial patch group is temporally subdivided into one or more patch groups.

4.2.2 Color interpolation and compensation

The points in the common patch should be colored individually in different frames to reflect the changing lighting conditions. For the n-th patch in a patch group, we interpolate the color of a point by

$$\sum_{j=1}^{k} \frac{c_j^n \cdot w_j}{w},$$

where we define the j-th nearest point in the n-th patch having the color c_j^n. The reciprocal of distance from this neighbor is denoted as w_j, and w sums all k weights. The interpolation operation is consistent in various color channels. We show an interpolation example in Fig. 6.

Since the patches in a patch group now have strictly aligned points (the common patch) with different colors, it is straightforward to compensate for the colors. We subtract the point colors of a patch (since the second patch) from the first patch, resulting in color residuals with little energy. These residuals are retained for the following compression.

Fig. 6 also explains how the colors are compensated, where the colors of points are reduced to quite small values.

4.3 Data Compression

The final stage of the encoder is to compress all data, including transformation matrices, octrees, colors, and color residuals into a bitstream. We employ RAHT [15] to compress the point colors and color residuals. The octrees are newly built over the common patches for compression, and the positional residuals of points are preserved optionally. Since the entropy of octree bytestreams and RAHT coefficients vary significantly from patch to patch, an adaptive entropy encoding approach is employed to improve the compression ratio. In practice, we test if the entropy codec outputs a bitstream smaller than the input octree and RAHT coefficients. If the output bitstream is larger, we use the original data instead.

The compressed bitstream follows a specific structure. It starts with a header indicating the number of frames in the video. The compressed patches are grouped and appended to the bitstream frame by frame. Each frame begins with a header indicating the number of patches. A patch is organized as an I-slice or a P-slice in the bitstream. A slice is composed of a header and the slice data. The slice header contains the fields of the frame timestamp, the patch index, the slice type, the transformation matrix, and the quantization parameters of RAHT. An I-slice and a P-slice only differ in the slice data part. An I-slice contains the common octree and the RAHT coefficients representing the colors, while a P-slice only contains the RAHT-coded coefficients representing the color residuals. The structural details of the compressed bitstream are shown in Fig. 7

5 DECODER

Encoding the patches in a frame is not dependable on each other, thus the slices are independently decodable. As an acceleration device, GPUs have been widely deployed towards applications such as neural network training [39], [40], high-dimension big data processing [41], search algorithms [42], video processing and rendering [43]. To speed up the volumetric video playback at a practical FPS, we design the decoder to be GPU-based for exploiting its massive parallel capability. Decoding the video on the GPU are also convenient for rendering its frames, which further accelerates the decoding-rendering pipeline.

The decoder works in three stages that decompress a bitstream to frames for rendering: 1) It loads the slices from the main memory to the GPU memory frame by frame. 2) The slices are decoded into a complete frame. The decoding processes of the I-slice and P-slice are different. 3) The decoded frames are directly rendered from the GPU memory. The overall workflow of the decoder is presented at the right side of Fig. 3.

The decoder employs three threads on to accomplish the tasks in data transmission, scheduling, and management throughout the decoding-rendering pipeline: 1) *Frame Loading Thread* (**FLT**): is responsible of loading data into GPU memory; 2) *Frame Decoding Thread* (**FDT**): invokes CUDA kernels to decode frames; 3) *Frame Rendering Thread* (**FRT**): focuses on rendering frames to the display.

5.1 Slice Loading

According to the metadata contained in the bitstream headers, the decoder could locate the boundaries of frames as well as slices. A *slice group buffer* is also allocated in the GPU memory. Every time the decoder has read all slices of a frame, no matter from the disk or the network, FLT copies them to the slice group buffer, appending to the slices of the previous frame. This is a ring buffer, and we only copy the slices if there is enough space. After successfully decompressing a frame, the space occupied by its slices is released. FLT is launched to continuously copy slices to the GPU memory, and after the copy of a frame, it notifies the start position in the slice group buffer to FDT.

5.2 Slice Decoding

At the CPU side, FDT is launched to keep checking if there are new slices copied to the slice group buffer. As long as a frame is completely copied to the buffer, it starts a GPU kernel to decode the slices in parallel. In the GPU kernel, one thread decodes one patch, and the decoded patches are sent to the frame buffer. The offset where a GPU thread should copy the point coordinates and colors in the frame buffer can be calculated according to the slice index and the number of points in its previous slices. Thus, we do not have to impose synchronization mechanisms between GPU threads though the output data to the same frame buffer.

5.2.1 I-slice

An I-slice is decoded into an I-patch. Based on the octree contained in the slice data, the decoder recovers point coordinates. If the positional residuals are additionally preserved, they are added back to the centroids of leaf nodes. Once the positions are determined, the decoder proceeds to associate the points with colors. Since the colors are organized in the order that how the points are extracted from the octree, the coloring task becomes straightforward. Both the octree and the I-patch will reside in the GPU memory until a new GOP starts because they are necessary for P-slice decoding.

5.2.2 P-slice

To decode a P-slice, the decoder first locates the previously decoded I-patch with the same index. The common patch is then transformed according to the transformation matrix stored in the P-slice header, which serves as the skeleton of the P-patch. Next, the color residuals are decoded and added to the colors of the I-patch. The recovered colors are associated with the transformed common patch and we eventually get the P-patch.

5.3 Frame Rendering

At the end of our decoding pipeline, FRT is started to render decoded frames from the frame buffer. It maps a *vertex buffer object* (VBO) in OpenGL[5] to the frame buffer. While FRT finds the frame buffer is filled up, it calls `glDrawArrays()` rendering a frame to the viewer. FRT also accepts inputs from the viewer in the form of translations and rotations and updates the viewport accordingly.

5. https://www.opengl.org/

5.4 Synchronization

Synchronization among the three decoding threads is achieved through the use of message queues. Initially, FLT copies a frame from CPU memory to a piece of GPU memory allocated as *slice group buffer*. Following this, FLT places a destination pointer for the copied frame into *filled buffer queue*, which serves as a synchronization mechanism between FLT and FDT. Upon receiving the pointer from the *filled buffer queue*, FDT allocates a block of memory from VBO based on the number of points in the frame. Subsequently, FDT invokes a CUDA kernel to decode the frame and stores the results in the allocated VBO. After decoding, FDT adds the offset and length of the result in VBO to *rendering frame queue*, which facilitates synchronization between FDT and FRT. Finally FRT retrieves the offset and length of a portion of the VBO from the *rendering frame queue* and utilizes OpenGL for rendering.

6 Evaluation

Implementation: Our coding framework is written in approximately 10K lines of CUDA and C++ code, with 8.5K lines of C++ code and 1.5K lines of CUDA code. The algorithm for searching the nearest neighbors as well as the ICP algorithm are from the PCL library.[6] We employ RAHT [15] to compress colors and adopt Zstandard[7] as the entropy encoder. The number of neighbors k described in Section 4.2.2 for color interpolation is chosen as 10, because we experimentally find that a higher k hardly improves the visual quality while introducing non-trivial computation overhead.

Experimental setup: We evaluate patchVVC+ and other compression methods on the same machine, which is equipped with two Intel CPUs of 2.9 GHz, one NVIDIA Geforce RTX 3090 GPU, and 128 GB main memory. To evaluate our decoding rendering pipeline, we use two separate machines. In addition to the above machine, another machine has two 2.10 GHz Intel CPUs, one NVIDIA Quadro RTX A6000 GPU, and 128 GB main memory. We compare patchVVC+ with four methods, patchVVC [13], GROOT [7], V-PCC [8], and MP3DG [17], where patchVVC is our previous implementation that adopts a coarse-grained patch segmentation method and a fixed GOP size. GROOT is also a 3D-based compression solution for volumetric videos. It accelerates the coding process by modifying the classical octree structure so that the tree traversal could be run on GPUs. Additionally, JPEG is used to compress the color of points. Following the above idea, we re-implement GROOT for comparison. Another 3D-based method we compared with is MP3DG [22]. It splits the point cloud frames into non-overlapping blocks and tries to extract similarities between inter-frame blocks. Official implementation of MP3DG is publicly available.[8] As the most representative 2D-based compression method for volumetric videos, V-PCC has its reference implementation.[9]

6. https://github.com/PointCloudLibrary
7. https://github.com/facebook/zstd
8. https://github.com/cwi-dis/cwi-pcl-codec
9. https://github.com/MPEGGroup/mpeg-pcc-tmc2

TABLE 1: The summary of videos in the 8i dataset

Video Name	Avg. # of Points/Frame	Required Bandwidth	Description
Loot	794 K	2.86 Gbps	A man playing piano
Longdress	834 K	3.00 Gbps	A female in a colorful dress
Soldier	1.01 M	3.87 Gbps	A man on guard with a gun
Redandblack	707 K	2.63 Gbps	A lady in a red and black skirt

Datasets, metrics, and traces: The dataset we used in the experiments is four volumetric videos [16], which are summarized in TABLE 1. All videos are 10 seconds long and 30 FPS. The coordinates of a point are represented by 3 32-bit integers. We turn the RGB colors into the YUV format following the ITU-R standard[10]. The resolution of octrees built in the experiments is always 1, and the positional residuals are not preserved. When compressing the videos with patchVVC, we set the GOP size to 15 for *Soldier* and *Loot*, and to 5 for the other two videos. We choose GOP size to 15 and 5 because this maximizes the compression ratio of the datasets without impairing the visual quality. PSNR.GEO and PSNR.Y [44] are defined to report the geometric fidelity and the color fidelity, respectively. Experiments are also conducted to simulate the FoV-adaptive streaming scenarios, where the network bandwidth demands are measured for different schemes. We use a publicly available viewport trace [31] with 26 records of watching the same videos.

6.1 RD-Curve

In the evaluation, we compare the compression performance of patchVVC+, patchVVC, GROOT, MP3DG, and V-PCC. The compression performance is measured in Rate-Distortion(RD) curve, which is the fundamental metric in video coding [9], [10]. Each point on an RD-curve reflects the bitrate of a video after encoding under specific parameters and the visual quality distortion of the reconstructed video after decoding. The RD-curve reflects the performance of the same codec under different parameter environments. The RD-curve plays a pivotal role in video codec assessment because it helps find the trade-off between the bitrate (compression) and video quality (distortion).

For the distortion, we measure PSNR.GEO to represent the geometric fidelity between the input point cloud frames and the decompressed ones. To control the different trade-offs between PSNR.GEO and the compression rate, we vary the MSE threshold of distinguishing the deviated patches and the non-deviated ones. The threshold ranges from 5 to 20. According to preliminary experiments, this is a range in which the compression ratio and distortion are well balanced. We also measure PSNR.Y for color fidelity and the quantization step of RAHT is scaled from 10 (30) to 50 when compressing the colors (color residuals).The selection of RAHT quantization step follow the original paper [15]. The RAHT quantization step is fixed to 10 (30) or the MSE threshold is fixed to 10 as another metric is changed. As the

10. https://www.itu.int/rec/R-REC-BT.601

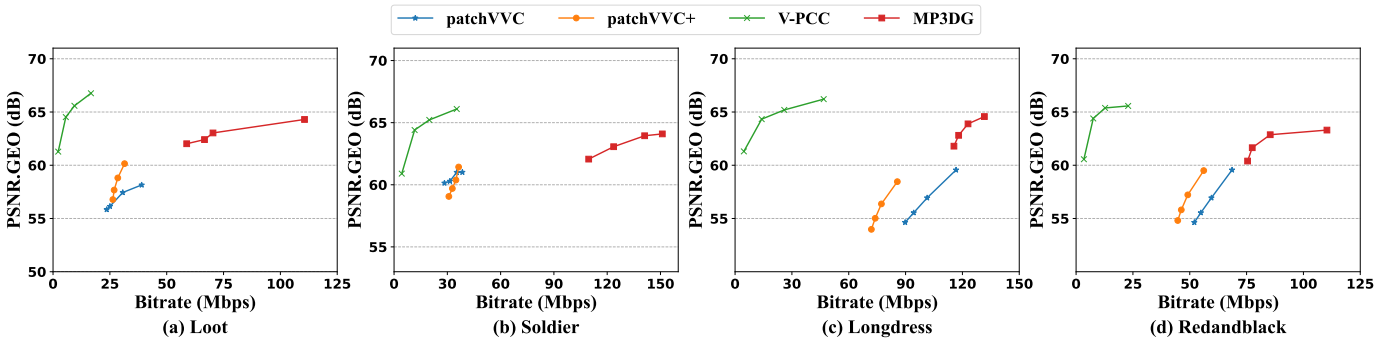


Fig. 8: The RD-curves (PSNR.GEO) of different compression schemes

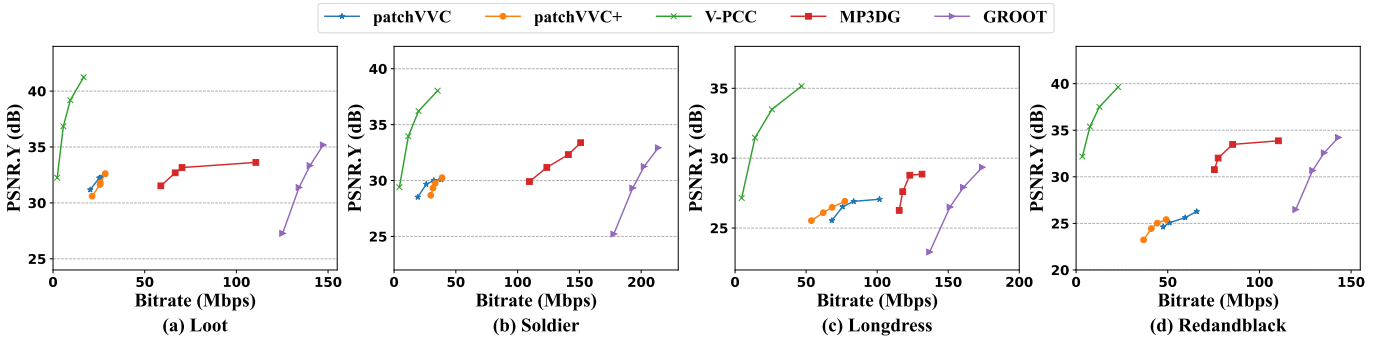


Fig. 9: The RD-curves (PSNR.Y) of different compression schemes

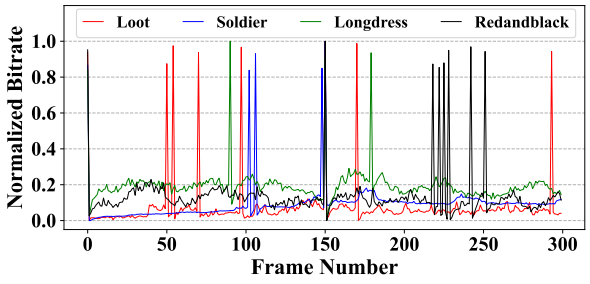


Fig. 10: The normalized bitrate of each frame.

results of experiments with varying parameter sets have the similar performance trend, we only report a set of results with higher visual quality (the results with MSE thresholds fixed at 5 and 10 have similar visual quality). In addition to patchVVC+ and patchVVC, we also need to make trade-offs between compression rates and visual distortion of other compression methods to plot their RD-curves. Five compression profiles, i.e., *ctc-r1/-r2/-r3/-r4/-r5*, have been preset for V-PCC, and we use four of them in the experiment. We discard *ctc-r2* since its results are almost the same as *ctc-r1*. GROOT compresses the geometric information is a lossless manner so we plot the RD-curves with the color fidelity only, where the JPEG encoding quality is ranged from 10 to 70. In MP3DG, the size of predictive macroblocks controls the rate-distortion trade-off, and we adjust its value in 8, 16, 32, and 64.

Fig. 8 reports the RD-curves of different codecs. The y-axis is the distortion in PSNR.GEO and the x-axis are the sizes of compressed data. The RD-curves in Fig. 9 show the distortion in PSNR.Y. Because patchVVC+ exploits the inter-frame redundancy in both the geometric and color domain effectively, it achieves the highest compression ratio among 3D-based methods, i.e., patchVVC, GROOT, and MP3DG. But patchVVC+ does not show comparable performance against V-PCC in most cases due to the mature compression techniques developed for decades in the traditional video codecs. In cases like *Soldier*, patchVVC+ approaches V-PCC as their average bitrate of compressed data is 36.36 Mbps and 35.29 Mbps, respectively. If we look into the compression ratio of different videos against GROOT and MP3DG, patchVVC+ reduces the average size of the compressed video by 5.88× and 3.88× for *Loot*, by 5.98× and 4.24× for *Soldier*, 2.55× and 2.15× for *Longdress*, and 3.25× and 2.25× for *Redandblack*. For the videos of *Longdress* and *Redandblack*, patchVVC+ achieves compression ratios lower than the other two videos because the videos contain more non-rigid moves and more color variance. Both the PSNR.GEO and PSNR.Y of patchVVC+ are slightly lower than V-PCC in most videos. The lowest average PSNR.Y of patchVVC+ appears in the video of *Redandblack*, which is 23.23 dB. The color interpolation step in our method can not tackle the sharp color variance frequently shown in this video.

By integrating a mechanism that generates patches at a finer granularity and organizes GOPs in an adaptive manner, patchVVC+ achieves superior performance against patchVVC in the videos of *Longdress* and *Redandblack*. For the original patchVVC, it is challenging to capture the inter-frame redundancy because of the non-rigid moves in the two videos. With the newly introduced mechanisms, patchVVC+ identifies more P-patches and reduces the av-

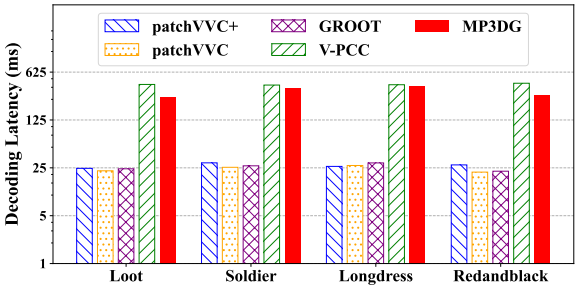


Fig. 11: The average decoding latency per frame of different compression schemes

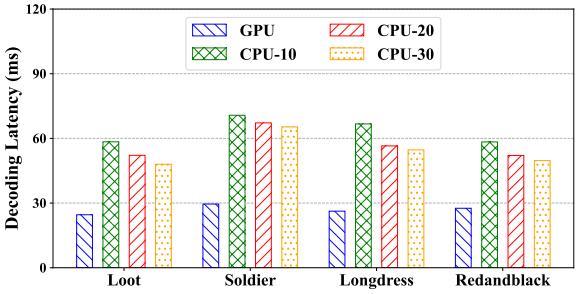


Fig. 12: The average decoding latency per frame of patchVVC+ on GPU and CPU

erage bitrate of compressed data by up to 1.36× (*Longdress*) and 1.34× (*Redandblack*). For the videos of *Loot* and *Soldier*, both versions of patchVVC have similar performance. The reason is that the moves in the videos are mostly rigid, and patchVVC could already eliminate the inter-frame redundancy effectively.

Since our design uses a constant quantization step, the bitrate of encoded volumetric video stream is not constant. We measure the size of each frame of the encoded volumetric video stream and normalize it, as shown in Fig. 10. All videos have thread similar trend. A small number of frames require a higher bitrate, which is normalized to that greater than 0.8. These frames are key frames. The remaining frames are predicted frames. The bitrate of those frames fluctuates slightly in a low range, which is normalized to that less than 0.3.

6.2 Decoding Latency

In this experiment, we measure how much time different codecs take to decode a frame on average. Fig. 11 reports the experimental results when measuring patchVVC+, patchVVC, GROOT, MP3DG, and V-PCC. patchVVC+, patchVVC, and GROOT have similar decoding speeds. Decoding a frame takes on average 26.99 ms in patchVVC+, 24.20 ms in patchVVC, and 25.69 ms in GROOT. The other two methods, V-PCC and MP3DG, take ∼10× longer time to decode a frame. The average decoding time of V-PCC is 413.81 ms and that of MP3DG is 394.05 ms. It is worth noting that MP3DG also supports multi-threading and we launch 30 threads when running it.

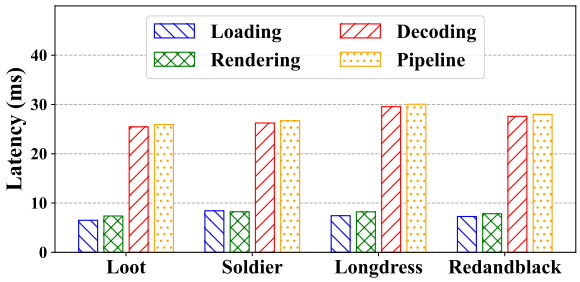


Fig. 13: The latency breakdown of the decoding-rendering pipeline in patchVVC+

TABLE 2: Average frames-per-second (FPS) of each video

GPU \ Video Name	*Loot*	*Soldier*	*Longdress*	*Redandblack*
RTX3090	38.57	37.44	33.29	35.71
RTXA6000	33.04	32.39	29.87	30.59

Though patchVVC+ uses GPU to accelerate the decoding, it can only achieve a comparable decoding speed as patchVVC. This is due to that we generate smaller patches. For a point cloud frame, we have $N_p \propto N_q^{-1}$, where N_p is the number of patches and N_q is the average number of points per patch. Inside a patch, the decoding process is octree-based, which has the time complexity of $O(\log N_q)$. As a result, to decode a complete frame, the time complexity is $O(N_p \log N_q)$. While we segment the frame into smaller patches, clearly we have to spend more time decoding all of them. To verify this, we also implement a version of patchVVC+ without GPU acceleration but activating multi-threading at the CPU side. We launch 10, 20, and 30 threads to decode the videos in the experiments, and the decoding latency is reported in Fig. 12. We can see that since more patches are generated for the videos, the decoding latency reaches as high as 70.73 ms, which doubles the decoding latency of the original patchVVC. The GPU memory occupied by the decoding program is about 38 Byte per point on average.

6.3 Rendering Performance

To verify that patchVVC+ could support real-time playback of volumetric videos, we implement a complete decoding-rendering pipeline. The pipeline consists of a loading module, a decoding module, and a rendering module, and the details are discussed in Section 5. In the experiment, we feed the compressed videos to the decoding-rendering pipeline and measure the time required by each individual module and by the whole pipeline. The results are shown in Fig. 13. We can see that loading a frame requires 6.50 ms to 8.43 ms, decoding a frame takes 25.47 ms to 29.56 ms, and rendering a frame takes 7.37 ms to 8.23 ms. Due to the well-designed pipeline, the connection overhead between modules is trivial. The latency of the complete pipeline is measured from 25.93 ms to 30.04 ms. We also measure the

(a) Loot (b) Soldier (c) Longdress (d) Redandblack

Fig. 14: Rendering screenshots during volumetric videos playback on a screen with 1920x1080 resolution.

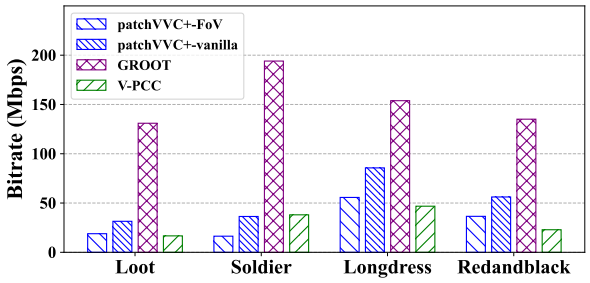


Fig. 15: The bandwidth demand of streaming single-object scenes in an FoV-adaptive manner

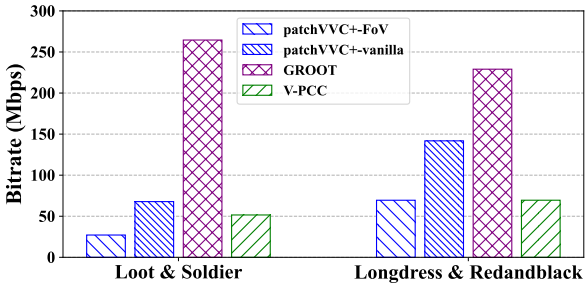


Fig. 16: The bandwidth demand of streaming multi-objects scenes in an FoV-adaptive manner

FPS of the renderer on two machines and present the results in Table 2. We can see that the rendering FPS ranges from 33.29 to 38.57 on RTX3090 and 29.87 to 33.04 on RTXA6000, which means fluent playback speeds in practice. Screenshots of the rendered volumetric videos are shown in Fig. 14.

6.4 patchVVC+ in FoV-Adaptive Streaming

Since patchVVC+ organizes patches at the similar positions of different frames together (patch groups), we expect to filter out the video content outside the user's frustum effortlessly. In contrast, V-PCC faces challenges in supporting this functionality due to its reliance on projecting point cloud objects onto a limited number of 2D frames. Cutting a 2D frame in the compression domain is hard because of the intra-frame correlation between blocks. To evaluate the data transmission requirements of an FoV-adaptive streaming system across different codecs, we conduct the following experiment. For patchVVC+, we configure the MSE threshold as 10 and the quantization step as 10/30 for compressing colors/color residuals. We apply the *ctc-r5* compression profile to V-PCC.

We simulate a streaming system with the FoV-adaptive functionality using a trace publicly available [31]. The trace records are presented in Unity global coordinate system, and to make them compatible with our implementation, we turn the data to be presented in the voxelized coordinate system. As in the 8i dataset, a dimension of 1.8 meters is mapped to an integer range $[0, 1024)$, we calculate the new camera position $C_v^{x/y/z}$ with

$$C_v^{x/y/z} = \frac{C_u^{x/y/z}}{1.8} \times 1024,$$

where $C_u^{x/y/z}$ means the $x-/y-/z-$ coordinate of a point in Unity global system. In the experiment, the FoV is constrained in a frustum of $100° \times 100°$.

To determine if a patch is visible, we need to identify each point of the patch is inside the FoV. If a patch contains such points, it needs to be transmitted. The other patches are considered invisible and are excluded from network delivery. Among the patches within the FoV, we further exclude occluded patches, whose points are all blocked by other patches in the frustum. We name the solution excluding invisible patches as *patchVVC+-FoV* and the original

one as *patchVVC+-vanilla*. On the other hand, V-PCC has to deliver all compressed data in the experiment. We also tweak GROOT to adapt to the FoV-adaptive streaming. To achieve this, points outside the FoV are removed and only the visible points are compressed by GROOT. We do not discuss MP3DG as it is not specifically optimized for FoV-adaptive streaming.

6.4.1 Single-object scenes

We first conduct experiments using the original videos from the 8i dataset, which are all single-object scenes. Following the viewport traces, we calculate the amount of visible data in network delivery. The results are presented in Fig. 15, with the y-axis denoting the bandwidth demand in Mbps, and the x-axis representing different videos. The findings reveal that GROOT does not significantly benefit from FoV-adaptive streaming. Despite removing points outside the FoV, GROOT always has the highest bitrate. For patchVVC+, removing invisible data substantially reduces the streaming bandwidth requirement, resulting in an average bandwidth saving of 20.55 Mbps. In the cases of V-PCC, though all data have to be transmitted, we still need low bandwidth because of its high compression ratio. Overall, patchVVC+-FoV and V-PCC demand much less bandwidth than GROOT in FoV-adaptive streaming. Notably, patchVVC+-FoV spends even less bandwidth (56.99% lower) than V-PCC to transmit *Soldier*.

6.4.2 Multi-objects scenes

We also combine two videos to form multi-object scenes in our evaluation. The videos of *Loot* and *Soldier* are combined into one video, and *Longdress* and *Redandblack* are combined into another. To create these multi-object scenes, we performed translation on *Soldier* and *Redandblack* using the vector $[600, 0, 0]^T$, and combined them with *Loot* and *Longdress*, respectively. The watching traces towards any of the involved object is directly applied to the combined video in the experiment. When compressing videos with patchVVC+, we additionally incorporate DBSCAN [45] to generate better partitioning results. The results depicted in Fig. 16 are observed similar to single-object scenes. Specifically, GROOT performs poorly across both scenarios. patchVVC+-FoV demonstrates a 47.37% bandwidth reduction when streaming *Loot & Soldier* against V-PCC. For the video of *Longdress & Redandblack*, the bandwidth consumption of the two methods is almost the same.

7 Conclusion

We extend our previous work patchVVC [13] to patchVVC+ in this paper, which is a 3D-based patch-wise compression framework for volumetric videos. Our framework features both high compression ratios and fast decoding speed. Different from patchVVC, patchVVC+ reduces the patch size in the segmentation stage, dynamically determines the GOP size, generates finer patch groups, and adaptively adopts the entropy encoding. While the newly introduced techniques further improve the compression efficiency, the decoding speed is slowed. We then migrate the decoder to GPU and implement a full-fledged decoding-rendering pipeline to verify that our design could be deployed in practice. patchVVC+ approaches the compression ratio of V-PCC, the representative 2D-based method, and has a similar decoding speed as GROOT, the state-of-the-art 3D-based method.

References

[1] X. Yue, B. Wu, S. A. Seshia, K. Keutzer, and A. L. Sangiovanni-Vincentelli, "A LiDAR Point Cloud Generator: From a Virtual World to Autonomous Driving," in *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*, 2018, pp. 458–464.

[2] H. P. H. Shum, E. S. L. Ho, Y. Jiang, and S. Takagi, "Real-Time Posture Reconstruction for Microsoft Kinect," *IEEE Transactions on Cybernetics*, vol. 43, no. 5, pp. 1357–1369, 2013.

[3] S. Choi, A.-D. Nguyen, J. Kim, S. Ahn, and S. Lee, "Point Cloud Deformation for Single Image 3d Reconstruction," in *2019 IEEE International Conference on Image Processing*, 2019, pp. 2379–2383.

[4] R. Schnabel and R. Klein, "Octree-based Point-Cloud Compression," in *Proceedings of the 3rd Eurographics*, 2006, pp. 111–120.

[5] P.-M. Gandoin and O. Devillers, "Progressive Lossless Compression of Arbitrary Simplicial Complexes," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 372–379, 2002.

[6] 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*, 2006, pp. 103–110.

[7] K. Lee, J. Yi, Y. Lee, S. Choi, and Y. M. Kim, "GROOT: A Real-time Sstreaming System of High-Fidelity Volumetric Videos," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.

[8] S. Schwarz, M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. A. Chou, R. A. Cohen, M. Krivokuća, S. Lasserre, Z. Li *et al.*, "Emerging MPEG standards for Point Cloud Compression," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 1, pp. 133–148, 2018.

[9] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H. 264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, 2003.

[10] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, 2012.

[11] U. Assarsson and T. Moller, "Optimized view frustum culling algorithms for bounding boxes," *Journal of graphics tools*, vol. 5, no. 1, pp. 9–22, 2000.

[12] B. Han, Y. Liu, and F. Qian, "Vivo: Visibility-aware mobile volumetric video streaming," in *Proceedings of the 26th annual international conference on mobile computing and networking*, 2020, pp. 1–13.

[13] R. Chen, M. Xiao, D. Yu, G. Zhang, and Y. Liu, "patchvvc: A real-time compression framework for streaming volumetric videos," in *Proceedings of the 14th Conference on ACM Multimedia Systems*, 2023, pp. 119–129.

[14] K. Krishna and M. N. Murty, "Genetic K-means Algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.

[15] R. L. De Queiroz and P. A. Chou, "Compression of 3D point clouds Using a Region-Adaptive Hierarchical Transform," *IEEE Transactions on Image Processing*, vol. 25, no. 8, pp. 3947–3956, 2016.

[16] E. d'Eon, B. Harrison, T. Myers, and P. A. Chou, "8i Voxelized Full Bodies–A Voxelized Point Cloud Dataset," *ISO/IEC JTC1/SC29 Joint WG11/WG1 (MPEG/JPEG) input document WG11M40059/WG1M74006*, vol. 7, p. 8, 2017.

[17] R. Mekuria and P. Cesar, "MP3DG-PCC, Open Source Software Framework for Implementation and Evaluation of Point Cloud Compression," in *Proceedings of the 24th ACM International Conference on Multimedia*, 2016, pp. 1222–1226.

[18] L. Li, Z. Li, V. Zakharchenko, J. Chen, and H. Li, "Advanced 3D Motion Prediction for Video-based Dynamic Point Cloud Compression," *IEEE Transactions on Image Processing*, vol. 29, pp. 289–302, 2019.

[19] L. Li, Z. Li, S. Liu, and H. Li, "Occupancy-Map-Based Rate Distortion Optimization for Video-based Point Cloud Compression," in *2019 IEEE International Conference on Image Processing*, 2019, pp. 3167–3171.

[20] W. Zhu, Z. Ma, Y. Xu, L. Li, and Z. Li, "View-Dependent Dynamic Point Cloud Compression," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 2, pp. 765–781, 2021.

[21] R. L. de Queiroz and P. A. Chou, "Motion-Compensated Compression of Dynamic Voxelized Point Clouds," *IEEE Transactions on Image Processing*, vol. 26, no. 8, pp. 3886–3895, 2017.
[22] R. Mekuria, K. Blom, and P. Cesar, "Design, Implementation, and Evaluation of a Point Cloud Codec for Tele-Immersive Video," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 4, pp. 828–842, 2016.
[23] Y. Xu, W. Hu, S. Wang, X. Zhang, S. Wang, S. Ma, Z. Guo, and W. Gao, "Predictive Generalized Graph Fourier Transform for Attribute Compression of Dynamic Point clouds," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 5, pp. 1968–1982, 2020.
[24] D. hanou, P. A. Chou, and P. Frossard, "Graph-based Compression of Dynamic 3D Point Cloud Sequences," *IEEE Transactions on Image Processing*, vol. 25, no. 4, pp. 1765–1778, 2016.
[25] C. Zhang, D. Florencio, and C. Loop, "Point Cloud Attribute Compression with Graph Transform," in *2014 IEEE International Conference on Image Processing*, 2014, pp. 2066–2070.
[26] C. Rezende, A. Boukerche, H. S. Ramos, and A. A. Loureiro, "A reactive and scalable unicast solution for video streaming over vanets," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 614–626, 2014.
[27] J. Kangasharju, F. Hartanto, M. Reisslein, and K. W. Ross, "Distributing layered encoded video through caches," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 622–636, 2002.
[28] F. Chen, P. Li, D. Zeng, and S. Guo, "Edge-assisted short video sharing with guaranteed quality-of-experience," *IEEE Transactions on Cloud Computing*, 2021.
[29] M. Hosseini and C. Timmerer, "Dynamic Adaptive Point Cloud Streaming," in *Proceedings of the 23rd Packet Video Workshop*, 2018, pp. 25–30.
[30] J. van der Hooft, T. Wauters, F. De Turck, C. Timmerer, and H. Hellwagner, "Towards 6DoF HTTP Adaptive Streaming Through Point Cloud Compression," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 2405–2413.
[31] S. Shishir, H. Alan, and C. Pablo, "User Centered Adaptive Streaming of Dynamic Point Clouds with Low Complexity Tiling," in *Proceedings of the 28th ACM International Conference on Multimedia*, 2020, pp. 3669–3677.
[32] J. Park, P. A. Chou, and J.-N. Hwang, "Rate-Utility Optimized Streaming of Volumetric Media for Augmented Reality," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 1, pp. 149–162, 2019.
[33] S. Gül, D. Podborski, J. Son, G. S. Bhullar, T. Buchholz, T. Schierl, and C. Hellge, "Cloud Rendering-based Volumetric Video Streaming System for Mixed Reality Services," in *Proceedings of the 11th ACM Multimedia Systems Conference*, 2020, pp. 357–360.
[34] S. Gül, D. Podborski, T. Buchholz, T. Schierl, and C. Hellge, "Low-Latency Cloud-based Volumetric Video Streaming using Head Motion Prediction," in *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2020, pp. 27–33.
[35] F. Qian, B. Han, J. Pair, and V. Gopalakrishnan, "Toward Practical Volumetric Video Streaming on Commodity Smartphones," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, 2019, pp. 135–140.
[36] A. Zhang, C. Wang, B. Han, and F. Qian, "YuZu : Neural-Enhanced Volumetric Video Streaming," in *19th USENIX Symposium on Networked Systems Design and Implementation*, 2022, pp. 137–154.
[37] P. J. Besl and N. D. McKay, "Method for Registration of 3-D Shapes," *Sensor fusion IV: control paradigms and data structures*, vol. 1611, pp. 586–606, 1992.
[38] A. F. Guarda, N. M. Rodrigues, and F. Pereira, "Constant size point cloud clustering: a compact, non-overlapping solution," *IEEE Transactions on Multimedia*, vol. 23, pp. 77–91, 2020.
[39] H. Jin, W. Wu, X. Shi, L. He, and B. B. Zhou, "Turbodl: Improving the cnn training on gpu with fine-grained multi-streaming scheduling," *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 552–565, 2020.
[40] F. Chen, P. Li, C. Wu, and S. Guo, "Hare: Exploiting inter-job and intra-job parallelism of distributed machine learning on heterogeneous gpus," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 253–264.
[41] M. Kim, L. Liu, and W. Choi, "A gpu-aware parallel index for processing high-dimensional big data," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1388–1402, 2018.
[42] T. Van Luong, N. Melab, and E.-G. Talbi, "Gpu computing for parallel local search metaheuristic algorithms," *IEEE transactions on computers*, vol. 62, no. 1, pp. 173–185, 2011.
[43] G. Shen, G.-P. Gao, S. Li, H.-Y. Shum, and Y.-Q. Zhang, "Accelerate video decoding with generic gpu," *IEEE Transactions on circuits and systems for video technology*, vol. 15, no. 5, pp. 685–693, 2005.
[44] S. Schwarz, G. Martin-Cocher, D. Flynn, and M. Budagavi, "Common Test Conditions for Point Cloud Compression," *Document ISO/IEC JTC1/SC29/WG11 w17766, Ljubljana, Slovenia*, 2018.
[45] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *The Second International Conference on Knowledge Discovery and Data Mining*, vol. 96, no. 34, 1996, pp. 226–231.

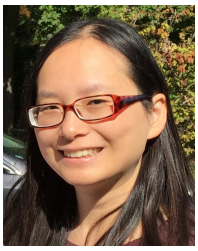
Dongxiao Yu received the B.S. degree in 2006 from the School of Mathematics, Shandong University and the Ph.D degree in 2014 from the Department of Computer Science, The University of Hong Kong. He is currently a professor in the School of Computer Science and Technology, Shandong University. His research interests include wireless networks, distributed computing and graph algorithms.

Ruopeng Chen received his B.S. degree from the School of Computer Science and Technology, Shandong University, China. He is currently pursuing the master's degree with the School of Computer Science and Technology, Shandong University, China. His research interests include video coding, video streaming and volumetric video compression.

Xin Li received his B.S. degree from the School of Information Science and Technology, Hebei Agricultural University, China. He is currently pursuing the master's degree with the School of Computer Science and Technology, Shandong University, China. His research interests include database systems and parallel computing.

Mengbai Xiao is currently a Qilu Young Professor in School of Computer Science and Technology at Shandong University. Before this, he was a postdoctoral researcher in Department of Computer Science and Engineering at the Ohio State University for two years. He received his Ph.D. from the Department of Computer Science at George Mason University in 2018.

Guanghui Zhang is currently a Qilu Young Professor in School of Computer Science and Technology. He was a Research Assistant Professor at the Department of Computer Science, Hong Kong Baptist University. From 2020 to 2021, he worked as a Post-Doctoral Fellow with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, and with the Centre for Advances in Reliability and Safety, The Hong Kong Polytechnic University. Before that, he received the Ph.D. degree in Information Engineering from The Chinese University of Hong Kong in 2020, and the M.S. degree in Electronic Science and Technology from Peking University in 2016. His research interest broadly lies in networking system, multimedia system, and machine learning.



Yao Liu is currently an Assistant Professor at Rutgers University. Previously, she was an Associate Professor (with tenure) in the Department of Computer Science at Binghamton University, State University of New York. She received the B.S. degree in computer science from Nanjing University and the Ph.D. degree in computer science from George Mason University. Her research areas include Internet mobile streaming, multimedia computing, Internet measurement and content delivery, and cloud computing.