Real-Time Decompression and Rasterization of Massive Point Clouds

RAHUL GOEL, IIIT Hyderabad, India
MARKUS SCHÜTZ, TU Wien, Austria
P. J. NARAYANAN, IIIT Hyderabad, India
BERNHARD KERBL, TU Wien, Austria

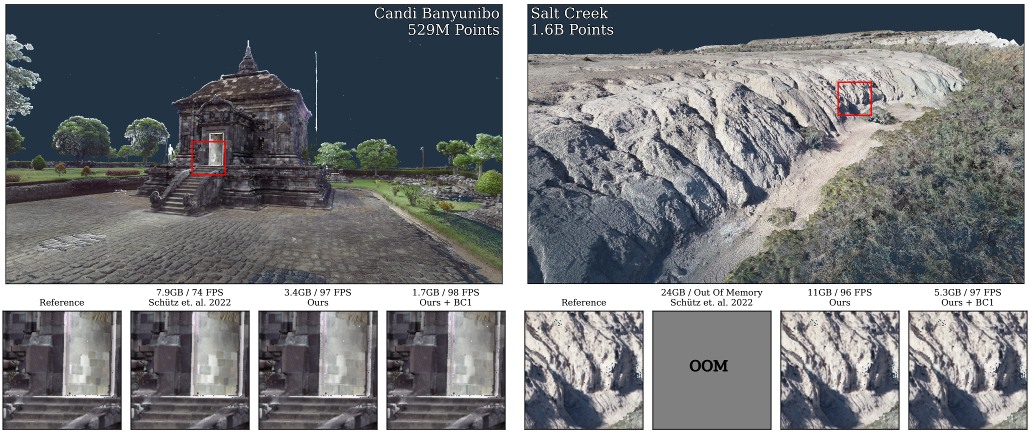


Fig. 1. Views of two large point clouds, visualized with our method. We compare speed and memory consumption to state-of-the-art work for fast rendering [Schütz et al. 2022]. Our method consumes 4× less memory, without impeding performance or visible quality loss (left, insets). The compact representation enables an 8 GB laptop GPU to render scenes that previous work could not handle on a 24 GB desktop GPU (right).

Large-scale capturing of real-world scenes as 3D point clouds (e.g., using LIDAR scanning) generates billions of points that are challenging to visualize. High storage requirements prevent the quick and easy inspection of captured datasets on user-grade hardware. The fastest real-time rendering methods are limited by the available GPU memory and render only around 1 billion points interactively. We show that we can achieve state-of-the-art in both while simultaneously supporting datasets that surpass the capabilities of other methods. We present an on-the-fly point cloud decompression scheme that tightly integrates with software rasterization to reduce on-chip memory requirements by more than 4×. Our method compresses geometry losslessly and provides high visual quality at real-time framerates. We use a GPU-friendly, clipped Huffman encoding for compression. Point clouds are divided into equal-sized batches, which are Huffman-encoded independently. Batches are further subdivided to form easy-to-consume streams of data for massively parallel execution. The compressed point clouds are stored in an access-aware manner to achieve coherent GPU memory access and a high L1 cache hit rate at render time. Our approach can decompress and rasterize up to 120 million Huffman-encoded points per millisecond on-the-fly. We evaluate the quality and performance of our approach on various large datasets against the fastest competing methods. Our approach renders massive 3D point clouds at competitive frame rates and visual quality while consuming significantly less memory, thus unlocking unprecedented performance for the visualization of challenging datasets on commodity GPUs.

CCS Concepts: • **Computing methodologies** → **Rasterization**; *Shared memory algorithms.*

Additional Key Words and Phrases: point cloud, rasterization, compression, real-time rendering

Authors' addresses: Rahul Goel, IIIT Hyderabad, India, rahul.goel@research.iiit.ac.in; Markus Schütz, TU Wien, Austria, mschuetz@cg.tuwien.ac.at; P. J. Narayanan, IIIT Hyderabad, India, pjn@iiit.ac.in; Bernhard Kerbl, TU Wien, Austria, kerbl@cg.tuwien.ac.at.

1 INTRODUCTION

Point clouds are one of the simplest, most commonly used graphics primitives that represent 3D data. Capturing real-world data often results in point clouds as the recovered representation. High-resolution, large-scale LIDAR scanning can produce a few billion to trillions of points. Such datasets are challenging to store, process, and render. Visualizing such massive point clouds in their entirety or even in parts is challenging since such datasets fail to fit in the on-chip VRAM of the graphics processing unit (GPU). In this paper, we introduce a scheme that uses lossless geometry compression and achieves real-time decompression and rendering of billions of points. This allows us to store and render such massive point clouds on a wide range of commodity GPUs. There are several real-world use cases for handling massive point clouds, such as cultural heritage preservation and architecture design. The ability for users to directly render their massive captured datasets on-site—perhaps on a laptop—will allow them to make corrections and adjustments in real-time. The lossless geometry compression helps us achieve accurate rendering upon close-up inspection. Our lossless approach, while showcased for rendering, also paves the way for GPU-based point processing that benefits from high-precision inputs (measuring, classification, triangulation) on extremely large datasets.

Current hardware rasterization pipelines cater to a triangle-based workflow dealing with vertex buffers, index buffers, UV maps, and textures. Point clouds are simpler and often do not require complex pipeline stages for appropriate rendering (e.g., quad shading). Recently, it has been shown that software rasterization, using atomic min-max operations, is more efficient than hardware rasterization for certain primitives [Evans 2015; Karis et al. 2021]. Schütz et al. [2021] build on this basic idea to create a *point sample rendering* software rasterizer for pixel-sized point primitives, where one point projects to precisely one pixel [Grossman and Dally 1998]. Such point-sample rendering methods are commonly employed for intuitive and authentic visualization of data by domain experts. They also can be easily combined with hole-filling/anti-aliasing for high-quality renderings as a fast alternative to more expensive splatting, as shown by ADOP [Rückert et al. 2022], VET [Franke et al. 2023] and TRIPS [Franke et al. 2024]. Recent state-of-the-art solutions [Schütz et al. 2022] can rasterize up to a billion points on commodity GPUs in real-time, but they are constrained due to GPU memory for scenes beyond that scale. We revisit the fundamental design of fast software rasterizers for point clouds to address this issue. Software rasterization provides higher flexibility in customizing the rendering pipeline than hardware pipelines. In order to maximize efficiency, we can choose the underlying data structures to store the primitives: trees, hash-tables, codebooks, or, in our case, compressed bitstreams.

In this paper, we present a compression-based approach that significantly reduces GPU memory requirements, coupled with a fast, on-the-fly decompression method that causes little overhead and easily integrates into software rasterization. We use a GPU-friendly *clipped* Huffman encoding to compress point cloud data. We use a two-level work distribution scheme: The initial point set is divided into equally-sized batches, each of which is further subdivided into equally-sized segments. A clipped Huffman codebook is constructed for the delta-encoded coordinates of points in each batch. The encoded bitstream is stored in an *access-aware* manner to accommodate render-time GPU memory access and raise cache hit rate. On the GPU, each batch of points can be independently processed by a compute unit, with its Huffman table being small enough to keep it in fast, locally-cached memory. Each segment of the block is decompressed and rasterized by one individual GPU thread. Achieving high compression rates usually implies unpredictable memory access patterns; however, our access-aware bitstream ordering ensures highly coalesced memory access when fetching from slow, device-wide memory. The proposed compression method is lossless and provides high visual quality while rendering massive point clouds. Finally, to allow users to target desired frame rates for their application, we combine our solution with an on-the-fly level-of-detail (LOD) method that requires no additional pre-processing.

We summarize the key contributions in this paper, which are given below.

- A *Clipped Huffman* encoding scheme that compresses points by more than 4× in a lossless, GPU-friendly manner.
- A 2-level partitioning scheme for points into batches and segments, with independent Huffman encoding of each segment. The bitstream of each segment is stored in an *access-aware* manner for efficient reading by GPU threads during rendering time.
- A level-of-detail methodology coupled with the partitioned batches, which allows faster renderings close to original visual quality if desired.

The remainder of this paper is structured as follows: In Section 2, we discuss previous related work. Section 3 establishes necessary preliminary context for describing our contributions. In Section 4, we present our Clipped Huffman stream design, the proposed (de-)compression schemes for point clouds on the GPU, and our accompanying level-of-detail rendering solution. Section 5 evaluates our methods in terms of resource consumption, quality, and performance. Section 6 discusses these results and draws final conclusions.

2 RELATED WORK

In this section, we review existing methods relevant to our proposed methodology. To provide the necessary context, we review the beginnings and current state of the art for point cloud rasterization, compression, and level-of-detail methods.

2.1 Point-based Software Rendering

Initiated by the arrival of the unified shader model and general-purpose programmability, the growing flexibility of the GPU as a massively parallel co-processor has renewed the interest in the exploration of custom rendering methods and new geometry primitives. Consequently, software-based rendering methods have returned. Kenzel et al. [2018] created a software-based replacement to the OpenGL pipeline with performance within one order of magnitude. Advancements in differentiable rendering have further fueled software rasterization to optimize primitives like triangles [Laine et al. 2020], splines [Li et al. 2020], point clouds [Rückert et al. 2022], and splats [Kerbl et al. 2023].

Günther et al. [2013] suggested rendering point primitives using custom GPGPU pipelines instead of the hardware rasterization pipeline and showed on-par or better performance. More recently, Schütz et al. [2021] exploited new atomic operations to design a simple, but effective visibility solution for point clouds with an interleaved framebuffer: they encoded projected depths and point IDs into a single 64-bit integer and used 64-bit atomic-min operations to preserve the closest points to the camera in each pixel. Substituting point IDs with their colors creates the image. This approach has been used for real-time rendering of point clouds that have been obtained by differentiable optimization [Franke et al. 2023; Rückert et al. 2022]. In our work, we also employ an approach that is based on software rasterization.

While effective, these fast, compute-based rendering solutions are still at an early exploratory stage. For instance, they do not address the concerns of memory consumption, relying on sufficient VRAM to store entire datasets in-core. Storage on commodity GPUs becomes the constraint for point clouds approaching more than a billion points and hinders visualization. Building on top of state-of-the-art methods [Schütz et al. 2021, 2022], we introduce lossless geometry compression and optional lossy color compression scheme, pushing the limits imposed by GPU memory severalfold.

2.2 Compressing Point Clouds

Point clouds captured from real-world data are bulky in size. Outside of high-performance point cloud rendering, methods to compress them without loss of visual quality have been widely explored. Ochotta and Saupe [2004] divided the point cloud into patches, and for each patch, calculated a height field onto a plane orthogonal to the normal cone axis. In their solution, the height fields were encoded using image compression methods. Golla and Klein [2015] used a similar approach for compressing points at speeds that match the data generation speeds of scanners. However, these compression methods are lossy, leading to changes and inaccuracies in the scanned geometry.

Data-driven deep learning methods have played an essential role in achieving higher compression rates for point clouds recently. One of the most significant methods is RIDDLE [Zhou et al. 2022], which represents the point cloud as range images. It predicts the pixels using a deep neural network, and entropy encodes the residual values. While such neural approaches have the potential for promising results, they are unsuitable for our goal of fast on-the-fly decompression due to the slow inference of neural networks.

LASZip [Isenburg 2013] is a popular reference method for losslessly compressing the standard LAS format to 7%–25% of the original size. They employ arithmetic encoding [Moffat et al. 1998; Witten et al. 1987] of deltas and combine it with predictive encoding to improve the deltas. This method uses context-based encoding to leverage correlation between attributes (e.g., compressing a point's intensity value based on its return number) to obtain the best compression ratio. Since arithmetic encoding is expensive to decode, we instead build on *Huffman coding* [Huffman 1952]: Huffman coding trades compression ratio for a higher decompression performance that is needed for decoding massive amounts of points on-the-fly.

Schuster et al. [2021] compress textured splats by learning a dictionary of atoms that are conceptually similar to DCT basis functions but targeted to a specific data set. Each splat stores indices and weights of up to 8 atoms, which can be used to reconstruct the splat's original texture directly during rendering. They also use a lossy quantization scheme to reduce memory for coordinates. In contrast, our method is lossless for 3D point geometry, which allows for authentic visualization. In practice, losslessness enables high-precision use cases in visual workflows, e.g., distance measuring.

2.3 Level of Detail Point Cloud Rendering

Out-of-core level of detail structures are an additional option to reduce the memory footprint of large data sets, as they enable us to only load and display subsets of the data that are needed for the current viewpoint. *QSplat* is one of the first point-based LOD structures, initially with the goal to render large triangle models [Rusinkiewicz and Levoy 2000]. Sequential Point Trees [Dachsbacher et al. 2003] flattens such a hierarchy into an array, sorted from lower to higher levels of detail, which allowed more efficient rendering on the GPU at the time by drawing appropriate subsets of that array. Layered Point Clouds [Gobbetti and Marton 2004], or LPC for short, introduces a GPU-friendly and view-dependant LOD structure for point clouds. Since then, variations and improvements of LPCs have become a standard for rendering massive point cloud data sets with hundreds of billions of points [Martinez-Rubi et al. 2015; Scheiblauer and Wimmer 2011; Wand et al. 2008].

Our compression and the developments in the domain of out-of-core LOD structures are complementary. Although we do not utilize the latter in this paper, our compression method should be beneficial to LOD rendering as it makes loading data more efficient. Further, the compressed data size would also allow to reduce out-of-core solutions' load-unload frequency, since we can keep larger number of points in the same chunk of memory. While we do not employ out-of-core

LODs, we do use a subset-based heuristic to improve rendering times to reach a user's performance targets in cases where we would overdraw by hundreds to thousands of points per pixel.

3 PRELIMINARIES

In this section, we briefly explain the concepts and methods that we use in our proposed approach. To meet the requirement of lossless compression of the point cloud geometry, we leverage Huffman Encoding to compress the coordinates. We incorporate this scheme in the existing state-of-the-art rendering methods for large point clouds [Schütz et al. 2021, 2022].

3.1 Huffman Codes

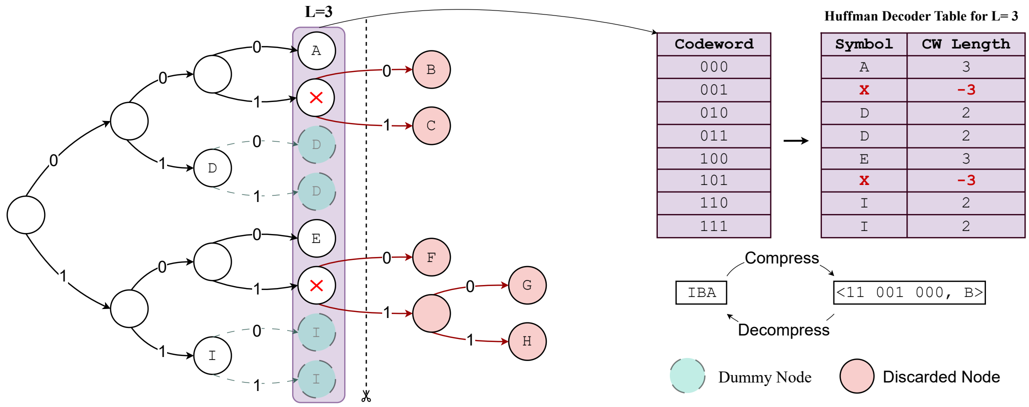
Huffman Coding [Huffman 1952] is a widely-used lossless compression scheme that uses fewer bits to represent frequent symbols. It is particularly well-suited for representing data with a skewed probability distribution. Huffman codes are variable-length codes that enable high compression. However, the default decoding process is essentially sequential; as a result, the corresponding parallel decoding of Huffman codes on the GPU is nontrivial. In the original formulation, decoding needs repeated traversal of the Huffman tree, which is suboptimal for the single-instruction-multiple-data (SIMD) architecture of the GPU. Nonetheless, encouraged by its effectiveness, several previous works propose to perform Huffman processing on the GPU: Tian et al. [2020] propose an efficient and customized parallel Huffman Encoding scheme. Shah et al. [2023] pre-compute a dictionary of codebooks, such that during compression, the most adequate one of them can be chosen on-the-fly. Our goal, however, is a different one: we seek a solution that, above all, enables fast <u>de</u>compression, such that compressed points can be used for image synthesis without impeding real-time performance. Weißenberger and Schmidt [2018] use the self-synchronizing property of Huffman codes to perform parallel decompression compatible with Huffman's original method. Inspired by their work, we extend the ideas from their method and propose our *Clipped Huffman* method that reduces the problems for simultaneous decompression and rendering.

3.2 Software Rasterization of Point Clouds

Due to its canonic nature, extensibility, and established use cases, we develop our solution in the context of basic one-point-to-one-pixel point sample rendering. We build on top of the state-of-the-art software rasterization methods proposed by Schütz et al. [2021, 2022] for large point clouds. Our pipeline is similar to their two-pass method: (1) the first pass iterates over the points and projects them to pixel coordinates. The depth and color of only the closest point are stored per pixel, using atomic-min operations after a cheap, conservative early depth test, (2) the seconds pass iterates over the pixels and de-couples the color from the frame buffer to create an image. Each pass occurs in parallel on the GPU using compute shaders.

One-point-to-one-pixel rendering often creates aliasing artifacts. To counter it, Schütz et al. [2021] implement a high-quality-shading (*HQS*) variant of their rasterizer (based on a simplified high-quality-splatting method by Botsch et al. [2005]) which uses three render passes: (1) the first iterates over points, projects them to the pixels and stores the depth, (2) the second iterates over points touching each pixel and sums colors of those that lie within 1% of the projected minimum depth, (3) the third blends the colors by dividing the sum of colors by the number of points in that 1% interval. In our implementation, we extend our point compression solution to this *HQS* variant to measure the rendering quality in Sec. 5.2.

Compute-based pipelines like those proposed by previous work have been shown to scale well for large point clouds, rendering at a throughput equivalent to 2 billion points at 60fps. However, accurate rendering of large point clouds takes a toll on the amount of GPU VRAM being used since the previous methods rely on trivial, uncompressed memory layouts to achieve such high



Codeword
000
001
010
011
100
101
110
111

Symbol	CW Length
A	3
X	-3
D	2
D	2
E	3
X	-3
I	2
I	2

Fig. 2. *Huffman tree and table.* Example of our Clipped Huffman tree, clipped at level $L = 3$. The deleted nodes are marked in red, dummy nodes in blue, and the collision nodes with crosses. The level $L = 3$ is mapped to a decoder table that we use for decompression. The collision nodes are re-purposed to signify that a non-frequent value has occurred and the separate stream should be looked-up. They are indicated by a negative codeword length. *Encoding Example:* The sequence *IBA* is encoded to <11 001 000, *B*>. The symbol *I*, having a high probability, has only a 2-bit code 11. Since *B* is a deleted note, its code is represented using its ancestor node at level $L = 3$ and is stored in the second buffer in its raw form. *A* has a 3-bit code 000. *Decoding Example:* <11 001 000, *B*> gets decoded to *IBA*. Since we know the maximum word length $L = 3$, we decode the stream using a 3-bit window. The first 3-bits 110 point to row 7. It decodes to *I* and indicates that the CW length was 2, implying that the remaining 1 bit must be a part of the next code. Thus, the next iteration decodes the 3-bit code 001. The negative length indicates a collision node; the symbol *B* is read from the separate data buffer. Finally, the next 3-bit code is read and decoded as *A*. More details in Sec. 4.1.

throughputs. In this paper, we propose to tackle this problem while maintaining rendering accuracy and real-time rates for massive point clouds.

4 METHOD

We start this section by explaining our variation of Huffman encoding and how it maps to a thread-block on the GPU using a small toy example. Next, we describe how our scheme can efficiently encode point cloud geometry. Next, we cover what problems are faced and how they are solved during real-time decompression. We also propose a level-of-detail method that fits well to Huffman encoded bitstreams. Finally, we describe the requirements of a good color compression scheme and how we meet them.

4.1 GPU-Friendly *Clipped* Huffman Streams

We illustrate our en-/decoding scheme with the help of an example with 9 symbols and their probabilities as follows: A: 0.02, B: 0.01, C: 0.01, D: 0.46, E: 0.04, F: 0.02, G: 0.01, H: 0.01, I: 0.42. The symbols *D* and *I* cover 88% of the data. The resulting Huffman Tree for this example is shown in Fig. 2. The supplementary video includes an animation for Clipped Huffman using this example.

Clipped Huffman. A Huffman tree can get arbitrarily deep, creating variable length codes that are GPU-unfriendly. We thus limit the maximum codeword length to L and clip the Huffman tree there. Leaf nodes in the first L levels are assigned codes based on the paths from the root. Leaf nodes beyond level L are deleted from the tree. For the less frequent symbols, we can afford to store them uncompressed in a separate buffer, keeping the Huffman tree compact and optimized for decompressing the frequent values. Fig. 2 shows the clipped Huffman tree for our example.

Decoder Table. To avoid the expensive tree traversal for decoding the encoded bitstream, we use a lookup table for decoding [Weißenberger and Schmidt 2018]. In the Huffman Tree, all leaf nodes outside of level L are *projected* to it as follows:

- *Dummy nodes.* Leaf nodes with a level lower than L are extended downward to create dummy nodes at level L. (These are D, I in the example.)
- *Collision nodes.* Leaf nodes with a level higher than L are back-tracked upward to create collision nodes at level L. (These are B, C, F, G, H in the example.)

After projection, all nodes at level L are stored in a lookup table of size 2^L, which is used to decode the encoded bitstreams. Codewords that terminate at collision nodes are marked using negative lengths: they signify non-frequent symbols and are looked up in a separate stream. The right side of Fig. 2 shows how the level $L = 3$ is mapped to a decoder table. **??** outlines how a single Huffman stream can be decoded at runtime. We note that this *Clipped Huffman* representation can be further optimized by using a level L node as a collision node in the dummy sub-tree. We do not exploit this, since the additional complexity in program logic outweighs the efficiency benefits.

For a batch of data, we create multiple such Huffman bitstreams with a common Huffman tree/table backbone. This allows individual thread blocks to decode these bitstreams independently in parallel. Clipping the Huffman tree at level L fixes the size of the decoder table, allowing us to store it in shared GPU memory for faster lookups. In practice, we use $L = 12$ and a table of size 4096. When applied to the point coordinate attributes in our tested datasets, we observed less than 2% of the symbols being stored separately, i.e., over 98% undergo compression (Tab. 1). An ablation for varying clipping levels and corresponding effect is provided in the supplementary material.

4.2 Compressing Point Cloud Geometry for Rendering

LIDAR data is commonly stored as LAS files where the geometry of points is represented as three 32-bit integers X, Y, Z. We first sort the points in Morton order and then divide the entire point cloud

Algorithm 1: Decoding a Clipped Huffman bitstream
N : Number of codewords to decode
EncodedBits: Huffman-encoded data
SeparateData : Separately stored data
DecoderTable : Huffman decoder table

```
idx1, idx2 ← 0, 0                                  // Pointers to the 2 Buffers
for i ← 0 to N do
    code ← EncodedBits[idx1 : idx1 + L]            // Read next L bits
    symbol, length ← DecoderTable[code]            // Lookup
    if length < 0 then
        symbol ← SeparateData[idx2]                // Second Buffer lookup
        idx1 ← idx1 + |length|
        idx2 ← idx2 + 1
    else
        idx1 ← idx1 + length                       // Only length bits used
    end
end
```

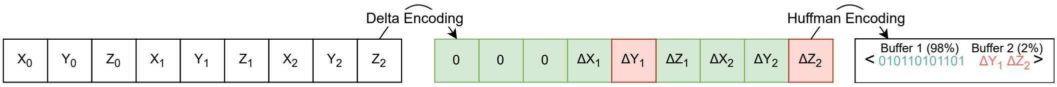


Fig. 3. *Coordinate Compression.* A sequence of point cloud coordinates is delta-encoded independently in X, Y, and Z dimensions. The delta values are Huffman-encoded using our scheme. This is effective since their distribution is skewed. Green values lie within the first L levels and are Huffman-compressed. Red values are stored directly in a separate buffer. On average, more than 98% of coordinates will be Huffman-encoded.

into moderate-sized *batches*. Each batch will be processed by an independent block of GPU threads. In this work, a batch comprises of 64K points. Points of a batch undergo *sequential delta-encoding (w.r.t. previous coordinates)* in X, Y, Z dimensions independently (Fig. 3). These delta values have a skewed distribution favourable for Huffman encoding. A Huffman tree is calculated for this set of delta values. The batch of points is then partitioned into equal-sized *segments* which share this Huffman tree. Each segment is decompressed and rendered independently by a thread of the block.

4.3 Parallel Decompression and Access-Aware Ordering

A batch of points is assigned to a thread block, with each decoding a segment. To maximize cache efficiency for Clipped Huffman on the GPU, we propose an *access-aware ordering* for bitstreams.

Due to variable-length codes, threads consume their bitstreams at different speeds. This can quickly lead to very uncoalesced memory accesses (please refer to Fig. 4 for a visual explanation using an example). However, given the configuration of GPU compute jobs and the rules in our decoding scheme, the order in which data blocks are requested can be simulated ahead of time; we thus re-arrange the bitstream data in that order. This imparts spatial locality[1] to the data and ensures that threads traversing memory at different speeds read from the same 128-bit cache line every single time. This is done by assigning every memory-block a 2-tuple key: $\langle i, t\rangle$ where i is the index of the first codeword in the memory block and t is the thread index. Arranging the memory blocks in increasing order of these keys produces an optimal memory layout as shown in Fig. 4. We refer readers to the supplementary video to explain this using animations. Furthermore, the supplementary document contains the decoding algorithm, which leverages this idea.

In practice, a memory-block is 4 bytes, and we pre-compute the memory layout for each warp (32 threads) independently. During decoding, for a thread to read the correct memory index, it needs to book-keep how fast the other threads in the warp are proceeding. This is done very cheaply using `__ballot_sync()`, a warp-level primitive. To maximize shared memory availability per block, each uses 1024 threads, or 32 warps. Please refer to supplementary for pseudocode. After re-arranging the memory layout, we profile and observe an increase in the L1-cache hit rate from 10% to 70%. We note that this cache-friendly ordering strategy could be applied to any other method with variable-sized data streams, as long as the consumption process is known to follow a pre-determined access pattern.

4.4 Subset Level of Detail

Schütz et al. [2022] enable on-the-fly LODs of point clouds using adaptive-precision data fetching. Unfortunately, their solution is incompatible with the delta encoding in our compression. To still enable users to balance quality and interactivity, we propose a batch-based LOD mechanism that preserves our compression, as well as point coordinate losslessness. When viewed from far away, it is not uncommon for thousands of points with similar relative depth to be projected to one pixel. Since points within a batch and its segments are proximate, they are more likely to project to the

[1] Spatial locality in GPU memory. Not to be confused with spatial locality in 3D space.

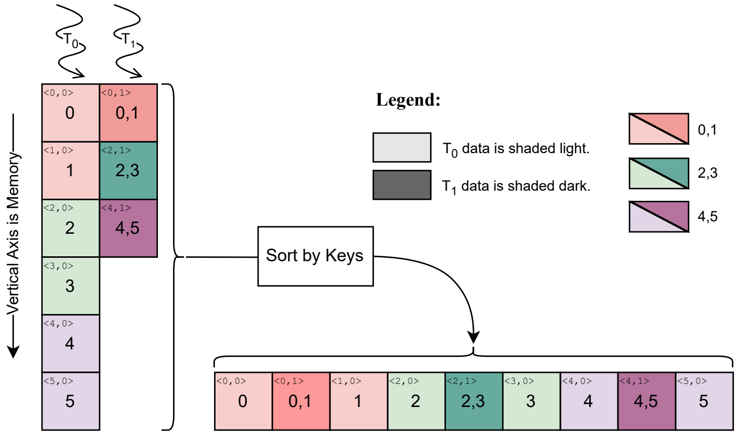


Fig. 4. *Access-aware memory ordering.* Consider the above example. There are only two threads (T_0, T_1) that decode 6 codewords each $\{0..5\}$. Due to variable-length codes, T_0 codewords are twice as long as T_1, i.e., T_0 encodes 1 value per memory block and T_1 is able to encode 2 codewords per memory block. But on the GPU, the threads have to decode the same value simultaneously in a for-loop. This causes T_0 to traverse memory blocks faster than T_1, leading to uncoalesced memory access. Sorting the memory blocks by the assigned keys imparts spatial locality, leading to simultaneously consumed codewords side-by-side (bottom row).

same pixel the farther away they are from the camera. Thus, only a fraction of points in a segment need to be decoded and rendered to achieve sufficient visual fidelity for them.

For each batch of points, we propose a subset level-of-detail method as described below:

- Calculate the pixel footprint of the batch by projecting the bounding box to the pixel coordinates.
- Determine an adequate fraction of points to be rendered based on the pixel footprint, employing a linear drop-off function.
- Allow a thread to decode and rasterize only the specified fraction of points in its segment.

The coordinates of the bounding box are pre-computed and stored during the pre-processing step. The linear drop-off function has a very lenient slope and offset, allowing 100% of points to be rendered for close-by batches, as shown in Fig. 5. In order to avoid holes, we have reduced the length of a segment greatly (64 points) such that compression ratios are not degraded. The minimum fraction of points is clamped from below at 10%. Notably, The fraction of points adjusts itself smoothly as batches move closer or farther away from the camera, improving the efficiency of our method. This helps us skip less important points, resulting in smoother framerates. However, skipping points necessarily introduces a small amount of error: We quantify this in Sec. 5.2.

4.5 Color Compression

Our renderer follows a deferred shading approach and colors each pixels in the second pass. As a result, spatial locality in the lookup of color values cannot be guaranteed. This demands a scheme that compresses color without a large number of neighboring data points. We use the BC1 [Iourcha et al. 1999] compression on colors, a common scheme for compressing textures in chunks of 4×4 blocks. We map a running length of 16 points to a 4×4 block, which is compressed to 4 bytes (a compression ratio of 6). To shade a pixel, these 4 bytes are read and decompressed on the fly.

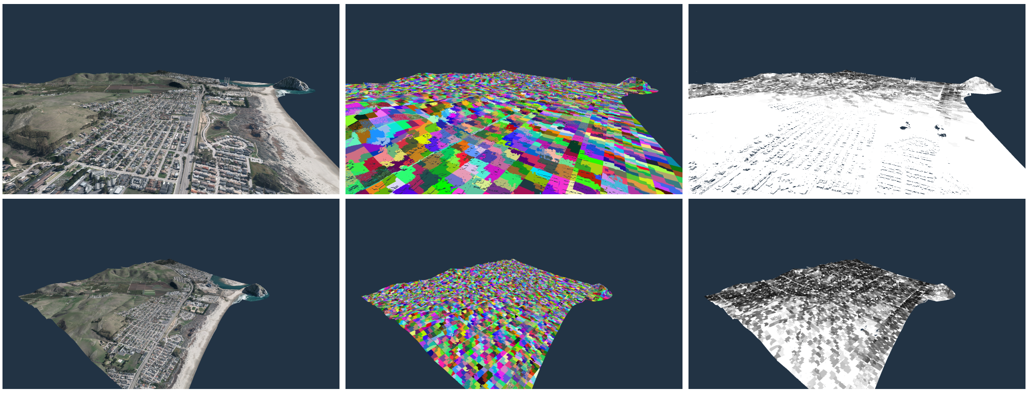


Fig. 5. *Subset Level of Detail.* In this figure, we show renders from two camera positions on the Morro Bay dataset (350M points). Col. 1 shows the reference, full render of the scene. Col. 2 shows the different batches of size 64K, where each batch is given a different color. Col. 3 shows the *fraction of points* being rendered using a grayscale gradient. White indicates all points being rendered, darker shades indicate lower percentages.

Table 1. *Evaluated datasets and compression Ratio (CR).* We test our method on a various datasets of different sizes. Col. 3 reports the percentage of data that lies in the clipped Huffman tree and undergoes compression. Col. 4 reports the lossless geometry compression ratio and Col. 5 reports the total compression ratio after accounting for BC1 color compression. On average, we see a 4× compression.

Dataset	Uncompressed Size	# Points	% Encoded	Geometry CR	Total Compression
Morro Bay	5.2 GB	350 M	98.34%	3.52×	3.84×
Banyunibo	7.9 GB	529 M	98.86%	4.26×	4.52×
Salt Creek	24 GB	1.62 B	98.72%	4.04×	4.32×
Ferrum	32 GB	2.16 B	98.91%	3.98×	4.27×
Neuchatel	90 GB	6.03 B	99.09%	4.47×	4.71×

It is to be noted that BC1 compression is lossy. However, the error in the color is negligible as shown in Sec. 5.2. We also experimented with using BC7 compression for colors; this provided a similar quality as BC1 while needing twice as much memory.

5 EVALUATION

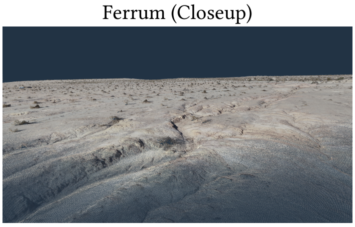
In this section, we explain our evaluation methods and the results we obtained. Our approach achieves high visual quality using significantly reduced memory and operates at real-time speeds. Following the evaluation scheme of Schütz et al. [2022], level-of-detail optimizations (subset LOD for ours and adaptive precision for theirs) are enabled during all experiments, except for the measurement of raw point throughput.

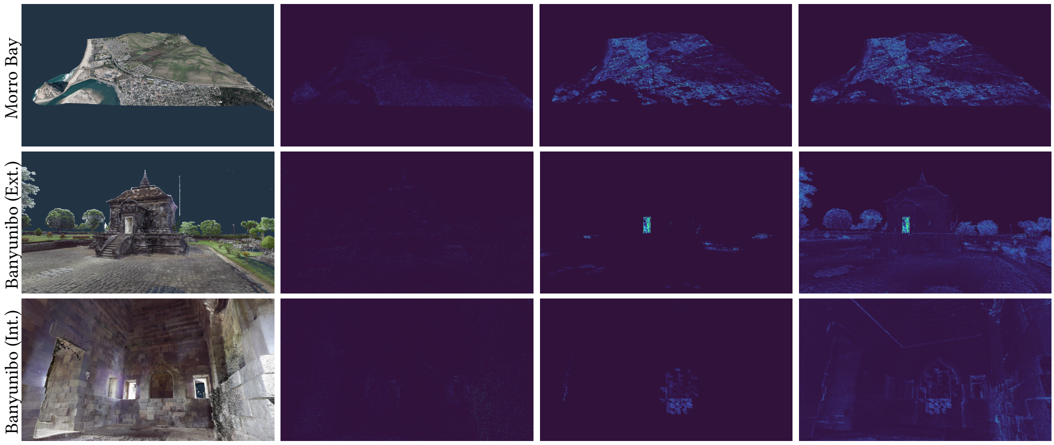
5.1 Compression Ratio

We benchmark our compression method on a variety of aerial and terrestrial scans. In Tab. 1, we list the suite of datasets we used for our evaluation, along with achieved geometry compression and total compression. Our compression ratio accounts for all the in-core buffers used for on-the-fly decompression, i.e., encoded bit-stream, separate data stream, Huffman decoder table, and the warp-level metadata required for tight-packing as explained in Sec. 4.3. Optionally, colors can be compressed using BC1, which gives a constant compression ratio of 6×. We account for this and

Table 2. Left: To evaluate the quality of our renders, we evaluate its PSNR with the reference image. We also compare it to Schütz et al. [2022] wherever their method allows it. – indicates the GPU running out of memory. Right: Closeup view for our second-largest scene, Ferrum.

Dataset	Schütz et al. [2022]	Ours	Ours + BC1
Morro Bay (Overview)	47.22	39.94	39.86
Banyunibo (Outside)	42.51	46.78	41.94
Banyunibo (Inside)	34.42	55.68	45.87
Salt Creek (Overview)	-	44.05	44.07
Salt Creek (Closeup)	-	43.90	35.26
Ferrum (Overview)	-	44.71	44.71
Ferrum (Closeup)	-	44.40	41.29





(a) Reference (b) Schütz et al. [2022] (c) Ours (d) Ours + BC1

Fig. 6. In this figure, we visualize the absolute error in our renderings. Col. 1 shows the reference image, which is generated with double precision and without any compression or LOD. Col. 2 shows the error in the images rendered by [Schütz et al. 2022] that quantize coordinates to 30-bits. Col. 3 shows errors in our rendering due to level-of-detail. Col. 4 shows additional error created due to BC1 color compression. As shown in Fig. 1, the error introduced by our method is visually imperceptible. In this figure, we have scaled it by 10× for better visibility. Please refer to the supplementary video for animated renderings of all scenes.

report the achievable compression ratio on the full dataset as well. On average, our method reduces the GPU memory consumption by more than 4×. In practice, this means that our approach enables users to visualize 4× larger datasets than previous methods on the same hardware.

5.2 Error Metrics

Since our subset LOD method (Sec. 4.4) skips points, the rendered image may not exactly match the reference image. To better understand this trade-off, we measure the error among the rendered images. Shading one pixel with exactly one point leads to a noisy aliased image. For an accurate comparison of renderings, we use the high-quality-shading (HQS) variant of these methods. The HQS method blends points in a 1% range of the projected depth [Schütz et al. 2021, 2022].

We calculate the PSNR of our renderings and compare it with the state-of-the-art [Schütz et al. 2022] in Tab. 2. For visualization, we also calculate the absolute error in the renderings and show

Table 3. In this table, we compare the time taken and memory consumed by our method to prior art on a wide range of datasets. We calculate the time taken on the 2-pass basic and 3-pass *HQS* variants. For each variant, we report results on a 24GB RTX 4090 GPU (left) and a 8GB RTX 4060 Laptop GPU (right). We also consider the different loads of overviews/exteriors ("Far"/"Ext.") and closeups/interiors ("Near"/"Int."). What other methods cannot render on a 24GB GPU, we can with just 8GB. Note that the memory used by Schütz et al. [2022] is equivalent to the figures listed in Tab. 1. – marks GPUs running out of memory.

		Schütz et al. [2022]				Ours					Ours + BC1				
Dataset	View	Basic (ms)		HQS (ms)		Basic (ms)		HQS (ms)		Mem.	Basic (ms)		HQS (ms)		Mem.
		24GB	8GB	24GB	8GB	24GB	8GB	24GB	8GB	used	24GB	8GB	24GB	8GB	used
Morro Bay	Far	1.9	7.4	6.8	31.3	2.6	11.1	8.6	28.4	2.5GB	2.5	11.1	7.6	27.0	1.3GB
	Near	1.1	4.0	3.8	13.9	2.5	12.0	5.2	25.7		2.5	12.0	5.2	25.8	
Banyunibo	Int.	4.0	-	11.9	-	4.9	24.5	11.1	53.8	3.4GB	4.8	24.3	10.6	53.7	1.7GB
	Ext.	3.4	-	13.5	-	4.8	25.6	10.3	56.2		4.7	25.6	10.2	56.6	
Salt Creek	Far	-	-	-	-	3.6	-	16.1	-	11GB	3.6	16.5	14.7	45.4	5.3GB
	Near	-	-	-	-	4.9	-	10.4	-		4.9	24.8	10.3	53.8	
Ferrum	Far	-	-	-	-	4.3	-	18.1	-	15GB	4.3	20.2	16.6	53.8	7.1GB
	Near	-	-	-	-	5.8	-	12.8	-		5.8	28.1	12.5	62.1	
Neuchatel	Far	-	-	-	-	-	-	-	-	32GB	11.8	-	47.3	-	18GB
	Near	-	-	-	-	-	-	-	-		14.3	-	28.9	-	

it in Fig. 6. We evaluate the effect of BC1 compression and observe no practical difference. For generating the reference images, we rasterized the point clouds without any compression, with double precision, and without any level-of-detail mechanism.

Tab. 2 and Fig. 6 demonstrate that our method does not cause significant degradations in visual quality. Moreover, in some scenes we even achieve better quality than Schütz et al. [2022]; this is due to completely lossless coordinates, which results in more coherent pixel depths and occlusion. We also evaluate the error on depth maps as shown in supplementary document. Note that our intention behind evaluating image quality on the HQS variant is to ensure minimum visual degradation due to our LOD. Raising the visual quality of point cloud renderings is not a primary focus of this work.

5.3 Performance

We compare the time and memory used by our method to prior art. We report performance metrics for two versions of our method: with and without color compression (Tab. 3). For each method, we measure two variants: the base method (2 passes) and the high-quality-shading (HQS) method (3 passes). We benchmark on a workstation with an NVIDIA RTX4090 and a gaming laptop with an NVIDIA RTX4060 Laptop GPU. For each, the timings are calculated to render images of size 1920 × 1080 for both closeup/interior and overview/exterior cameras. In accordance with the reported compression ratios, our used memory in Tab. 3 is a fraction of [Schütz et al. 2022] which directly uses the uncompressed data (c.f. Tab. 1).

In Tab. 4, we further report *point throughput*: it measures the raw point processing power of an algorithm. It differs from rendering since we disable the frustum culling and any form of LOD to process all the points at all times. The throughput of rendering Huffman-compressed data reaches up to 124M points per millisecond, compared to 184M points per millisecond that can be achieved with uncompressed data sets. Note that our method's LOD reduces the number of processed points. In contrast, [Schütz et al. 2022] maintains the same number but fetches their attributes with adaptive precision. Hence, our LOD mechanism helps to compensate for lower raw point throughput. Unsurprisingly, LOD does not play a major role in close-up views for our method, but for distant shots it can reduce the GPU workload immensely, depending on the number of points in the dataset (Morro Bay: 1.8×, Candi Banyunibo: 1.2×, Salt Creek: 4.9×, Ferrum: 5.1×).

Table 4. Left: We report the point throughput (in million points per ms) of each method on RTX 4090. Adding BC1 color compression has a negligible effect on throughput. Right: the largest evaluated scene, Neuchatel.

Dataset	Points	Schütz et al. [2022]	Ours
Morro Bay	350 M	184.4	81.5
Banyunibo	529 M	147.0	76.7
Salt Creek	1.62 B	-	108.9
Ferrum	2.16 B	-	110.7
Neuchatel	6.03 B	-	124.4

Neuchatel (Closeup)

Due to the different mechanics and resource usage of our method, frame times are not consistently higher or lower than Schütz et al. [2022] (e.g., note that they are higher for Banyunibo Basic, but lower for Banyunibo HQS). On average, our method tends to produce slightly lower framerates, depending on the scene. Overall, however, our solution maintains the prominently high performance of compute-based point sample rendering solutions. With our method, we can visualize scenes with more than 6 billion points at 60+ FPS on a consumer-grade GPU, without compromising quality, while previous work runs out of memory around the 1.5 billion mark.

Since we use shared memory for our look-up table, threads access the table entries with bank conflicts. These bank conflicts create the biggest bottleneck in our current implementation, limiting the impact of other design optimizations. Hence, to get a better idea of the benefits that *access-aware ordering* may provide when fully optimized, we substitute look-up with (compiler-unoptimized) instructions that generate synthetic point data from the encoded bits. For the Morro Bay scene, we observe a 30% runtime improvement in the render loop due to the 7× increment in cache hits.

While we did not consider optimizing the compression speed in this work, our CPU implementation is reasonably quick, taking ≈44*s* to compress the Morro Bay (350M) dataset on a 13th Gen Intel Core i9-13900F. This speed depends on the size and the compressibility of the data. We leave a massively parallel GPU implementation of our compression preprocess to future investigation.

6 DISCUSSION AND CONCLUSION

In order to facilitate the rendering of large point clouds that struggle to fit in the GPU, we propose a compression-based solution. We apply lossless compression on point geometry and store it in a format that can be simultaneously decompressed and rendered using software rasterization. We achieve a compression ratio of >4× and maintain real-time framerates alongside high-quality rendering. In contrast to previous work, our lossless representation allows us to recover exact point coordinates in massive point clouds. This property can raise the fidelity in close-up renderings and common practical tasks requiring maximum precision (e.g., distance measuring). Our method is designed for real-time visualization of large-scale captures. While our results already suggest strong benefits for practical, real-world applications, there are several directions for further improvement.

Huffman Encoding. We use a lookup table as a proxy for the Huffman tree. This method of storage is redundant due to the repetition of symbols (Fig. 2). Improving this can enable (i) storing a larger Huffman tree/table in shared memory, leading to higher compression, and (ii) eliminating the separate data buffer. We hope to exploit the *self-synchronizing property* of Huffman codes [Weißenberger and Schmidt 2018] to push compression ratios even further by encoding in a single stream.

Lossy Compression. Lossy geometry compression can give better compression, where appropriate. LIDAR scanners often sample in a straight line. The idea of BC1 color compression can directly be applied here: store two end-points of the scan-line and interpolate the in-between points.

Performance Bottlenecks. Bank conflicts in shared memory for Huffman table look-ups cause a performance bottleneck. Minimizing them would result in raised efficiency. Also, the computation of encoded bitstreams during pre-processing takes a few minutes, depending on the size and data compressibility. Online conversion could be a relevant quality-of-life improvement of our method.

With the steadily increasing complexity of captured datasets, our work provides an important step toward addressing the concerns of domain experts, as well as regular users: With the presented approach, we hope to address the need for swift, authentic rendering methods, which are trailing behind the rapidly advancing solutions for reconstruction and generation of large datasets.

A webpage for this paper with all relevant links (supplementary, code and video) is available at: *https://rahul-goel.github.io/pcrhpg24.*

ACKNOWLEDGMENTS

The authors wish to thank *PG&E and Open Topography* [2013] for the *Morro Bay* data set; *Buckley et al. and Open Topography* for the Ferrum [2021] and Salt Creek [2021] data sets; *SITN* [2023] for the *Neuchatel* data set; and the *TU Wien, Institute of History of Art, Building Archaeology and Restoration* [2019] for the *Candi Banyunibo* data set.

This research has been funded by *WWTF* (project *ICT22-055 - Instant Visualization and Interaction for Large Point Clouds*) and *Department of Science and Technology, India.*

REFERENCES

Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. 2005. High-quality surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*

William Buckley and Thomas K. Rockwell. 2021. Structure from Motion data along the sSAF, Salt Creek. https://doi.org/10.5069/G9TX3CK0 Distributed by OpenTopography.

William Buckley, Thomas K. Rockwell, and Allen Gontz. 2021. Structure from Motion data along the sSAF, Ferrum. https://doi.org/10.5069/G93F4MT2 Distributed by OpenTopography.

Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. 2003. Sequential Point Trees. *ACM Trans. Graph.* (2003).

SITN (Canton de Neuchâtel Système d'information du territoire neuchâtelois). 2023. Neuchatel.

Alex Evans. 2015. Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game. In *ACM SIGGRAPH 2015 Courses, Advances in Real-Time Rendering in Games.* http://media.lolrus.mediamolecule.com/AlexEvans_SIGGRAPH-2015.pdf.

Linus Franke, Darius Rückert, Laura Fink, Matthias Innmann, and Marc Stamminger. 2023. VET: Visual Error Tomography for Point Cloud Completion and High-Quality Neural Rendering. In *SIGGRAPH Asia 2023 Conference Papers.*

Linus Franke, Darius Rückert, Laura Fink, and Marc Stamminger. 2024. TRIPS: Trilinear Point Splatting for Real-Time Radiance Field Rendering. *Computer Graphics Forum* (2024).

Enrico Gobbetti and Fabio Marton. 2004. Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-sampled Models. *Comput. Graph.* 28, 6 (2004), 815–826.

Tim Golla and Reinhard Klein. 2015. Real-time point cloud compression. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*

J. P. Grossman and William J. Dally. 1998. Point Sample Rendering. In *Rendering Techniques '98.*

Christian Günther, Thomas Kanzok, Lars Linsen, and Paul Rosenthal. 2013. A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds. *J. WSCG* (2013).

U. Herbig, L. Stampfer, D. Grandits, I. Mayer, M. Pöchtrager, Ikaputra, and A. Setyastuti. 2019. DEVELOPING A MONITORING WORKFLOW FOR THE TEMPLES OF JAVA. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-2/W15 (2019), 555–562. https://doi.org/10.5194/isprs-archives-XLII-2-W15-555-2019

David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* (1952).

K. I. Iourcha, K. S. Nayak, and Z. Hong. U.S. Patent 5,956,431, 1999. System and method for fixed-rate block-based image compression with inferred pixel values.

Martin Isenburg. 2013. LASzip: lossless compression of LiDAR data. *Photogrammetric Engineering & Remote Sensing* 79 (2013).

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. In *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games, Part 1.* https://advances.realtimerendering.com/s2021/index.html.

Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A High-performance Software Graphics Pipeline Architecture for the GPU. *ACM Trans. Graph.* (2018).

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics* (2023).

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Transactions on Graphics* (2020).

Tzu-Mao Li, Michal Lukáč, Gharbi Michaël, and Jonathan Ragan-Kelley. 2020. Differentiable Vector Graphics Rasterization for Editing and Learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* (2020).

Oscar Martinez-Rubi, Stefan Verhoeven, M. van Meersbergen, Markus Schütz, Peter van Oosterom, Romulo Goncalves, and T. P. M. Tijssen. 2015. Taming the beast: Free and open-source massive point cloud web visualization. Capturing Reality Forum 2015, Salzburg, Austria.

Alistair Moffat, Radford M Neal, and Ian H Witten. 1998. Arithmetic coding revisited. *ACM Transactions on Information Systems (TOIS)* (1998).

Tilo Ochotta and Dietmar Saupe. 2004. 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*. Eurographics Association, Goslar, DEU.

Pacific Gas & Electric Company. 2013. PG&E Diablo Canyon Power Plant (DCPP): San Simeon and Cambria Faults, CA, Airborne Lidar survey. https://doi.org/10.5069/G9CN71V5 Distributed by OpenTopography.

Szymon Rusinkiewicz and Marc Levoy. 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., USA.

Darius Rückert, Linus Franke, and Marc Stamminger. 2022. Adop: Approximate differentiable one-pixel point rendering. *To appear in ACM Transactions on Graphics* (2022).

Claus Scheiblauer and Michael Wimmer. 2011. Out-of-Core Selection and Editing of Huge Point Clouds. *Computers & Graphics* (2011).

Kersten Schuster, Philip Trettner, Patric Schmitz, Julian Schakib, and Leif Kobbelt. 2021. Compression and Rendering of Textured Point Clouds via Sparse Coding. In *High-Performance Graphics - Symposium Papers*, Nikolaus Binder and Tobias Ritschel (Eds.). The Eurographics Association.

Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2021. Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. *Computer Graphics Forum* (2021).

Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2022. Software Rasterization of 2 Billion Points in Real Time. *Proc. ACM Comput. Graph. Interact. Tech.* (2022).

Milan Shah, Xiaodong Yu, Sheng Di, Michela Becchi, and Franck Cappello. 2023. Lightweight Huffman Coding for Efficient GPU Compression. In *Proceedings of the 37th International Conference on Supercomputing*.

Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*.

Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. 2008. Processing and interactive editing of huge point clouds from 3D scanners. *Computers & Graphics* (2008).

André Weißenberger and Bertil Schmidt. 2018. Massively Parallel Huffman Decoding on GPUs. In *Proceedings of the 47th International Conference on Parallel Processing*. Association for Computing Machinery.

Ian H Witten, Radford M Neal, and John G Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* (1987).

Xuanyu Zhou, Charles R. Qi, Yin Zhou, and Dragomir Anguelov. 2022. RIDDLE: Lidar Data Compression With Range Image Deep Delta Encoding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.