Compression and Interactive Visualization of Terabyte Scale Volumetric RGBA Data with Voxel-scale Details

Mehmet Oguz Derin
Morgenrot, Inc.
Türkiye
oguz@morgenrot.net

Takahiro Harada
Morgenrot, Inc.
, Advanced Micro Devices, Inc.
USA
takahiro.harada@amd.com

Yusuke Takeda
Hokkaido Univ.
Japan
ytakeda@sci.hokudai.ac.jp

Yasuhiro Iba
Hokkaido Univ.
Japan
iba@sci.hokudai.ac.jp

ABSTRACT

We present a compressed volumetric data structure and traversal algorithm that interactively visualizes complete terabyte-scale scientific data. Previous methods rely on heavy approximation and do not provide individual sample-level representation when going beyond gigabytes. We develop an extensible pipeline that makes the data streamable on GPU using compact pointers and a compression algorithm based on wavelet transform. The resulting approach renders high-resolution captures under varying sampling characteristics in real-time.

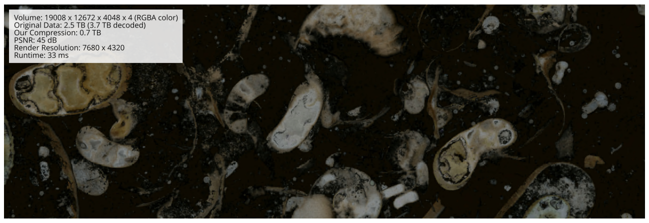
CCS CONCEPTS

• **Computing methodologies** → **Rendering**; **Ray tracing**.

KEYWORDS

compression, sparse volumes, data structures, ray tracing, rendering, visualization

ACM Reference Format:
Mehmet Oguz Derin, Takahiro Harada, Yusuke Takeda, and Yasuhiro Iba. 2022. Compression and Interactive Visualization of Terabyte Scale Volumetric RGBA Data with Voxel-scale Details. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Posters (SIGGRAPH '22 Posters), August 07-11, 2022.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3532719.3543256



Volume: 19008 x 12672 x 4048 x 4 (RGBA color)
Original Data: 2.5 TB (3.7 TB decoded)
Our Compression: 0.7 TB
PSNR: 45 dB
Render Resolution: 7680 x 4320
Runtime: 33 ms

Figure 1: Render of a high-resolution volumetric capture of a rock sample with annotation of statistics and performance.

1 INTRODUCTION

In scientific visualization, capturing and rendering high-resolution volumetric data is an active area of research for both Earth and extraterrestrial material samples. Although there are developments in compression and rendering [Graciano et al. 2021] [Aleksandrov et al. 2021] [Wald et al. 2017], no work has been there which enables the display of RGBA volumes with sizes exceeding 19,000 x 12,500 x 4,000 on 8K screens, where the preservation of voxel-level structures is crucial for findings through interactive exploration. We propose a method that scales to terabyte-scale volumetric data, where our main contributions are a data structure that captures sparsity and compresses, a construction approach that parallelizes the demanding aspects of the building of our data structure, and finally, a rendering approach that reuses stack and hierarchical DDA [Museth 2021] to accelerate content on GPU while streaming.

SIGGRAPH '22 Posters, August 07-11, 2022, Vancouver, BC, Canada
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9361-4/22/08.
https://doi.org/10.1145/3532719.3543256

2 METHOD

For interactive visualization at this scale with high unpredictability of fetch patterns, we process our data into an efficient format.

2.1 Data Structure

Our data structure is a tree where each internal node contains, depending on the build-time configuration, $N^3(2^3$ to $6^3)$ children, which can be null, and minimum color, average color, and maximum color attributes for our compression, besides offsets for children (see Fig. 2). Since we compress the node offset to 8-bits, the maximum value we can use for N is 6. These pointers are offsets to a 32-bit address stored within the node, introducing the primary challenge of parallelization with children exporting to a linear buffer at each level needing to be sequential, besides the secondary challenge with the need for recursive color attributes to achieve wavelet compression. At the bottom level, we contain similarly sized arrays to obtain voxel color values, enabling storage of values starting from 4-bit channels. Reconstruction of a value at a single level is either $x_r^i = (max^{i+1} - ave^{i+1})x_c^i + ave^{i-1}$ or $x_r^i = (ave^{i+1} - min^{i+1})x_c^i + min^{i-1}$, where x_r^i, x_c^i are reconstructed and compressed values respectively. We select one equation depending on the sign bit we store. This is executed recursively from the root to the bottom to reconstruct voxel values.

2.2 Construction

2.2.1 First Pass. The first pass is a bottom-up process that starts with building nodes at level 0. To achieve parallelization, which is necessary due to the scale of the data, we first run a pass over the data to build nodes at level 0. Since our input data measures in terabytes, for which it is difficult to allocate all the nodes in the tree on the RAM, we use a hashmap to keep track of nodes we

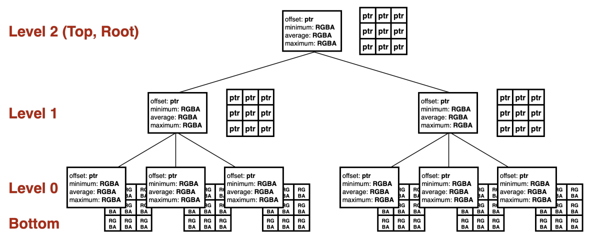


Figure 2: Our data structure with N=3.

created to store them efficiently. The hash key is computed using level, index.x, index.y, index.z as the inputs. Note that the node data we write here uses 32-bit or 64-bit types, which is different from the node data we store in the end. It is compressed using a single bit for a node existence rather than an expanded offset. Depending on the system configuration, our preprocessor can store the node data in RAM or disk. We use a single thread to process N images at the very first pass so it can conclude the build of a level 0 node by itself. It computes the maximum, minimum, and average from the input data and child masks. Then it stores the node data to the location looked up by the hash map. After processing level 0, we build level 1, where we launch the number of jobs equal to the number of nodes we may get if we use a dense grid. A thread goes through the nodes of its children to compute max, min, and average child masks. The hash of a child can be calculated from the level index and index in the 3D coordinate. This process is repeated until we reach the root node.

2.2.2 Second Pass. After the first pass is complete, the second pass parallelizes over the nodes at any level to perform the compression of the max, min, and average. The reason why we can take any node at a time to process in the second pass is using the nature of our compression algorithm, which is local. In order to compress the max, min, and average value of a single node, we only need to know one level above, where the preprocessor quickly seeks the file handle to the determined file positions and recursively calculates the wavelet compressed value. After all the nodes are compressed, we can reconstruct the node value by simply traversing the nodes from the root to the leaf.

2.3 Traversal

Our compact representation stores data in a way where we can load specific voxels on-demand without complete block-wise decompression, which is necessary for the exposed parameters of radius and user-determined custom path direction functions. Two important aspects that enable our rendering approach are fast ray traversal and fast host memory access for individual voxel values in this setting. Accessing a specific voxel on GPU is trivial where the desired voxel position needs to be scaled down and modulated accordingly at each level while descending, but since our data structure captures sparsity and then represents both color and memory location recursively, it is essential to skip empty space and reuse value already fetched for higher levels when traversing along a ray. To achieve this, we implement a hierarchical DDA for skipping empty space. Also, we avoid traversing the tree from the root as much as possible in our novel traversal algorithm. Specifically, we keep a stack of the values we computed when we traversed at the very first time to the bottom. When we access other voxel data, we backtrack the structure to find the node where we can start descending. In this way, we primarily traverse the entire data structure from the root once at the very beginning. At runtime, we store all data except for the bottom level directly on the GPU memory. We expose the bottom level by registering host memory.

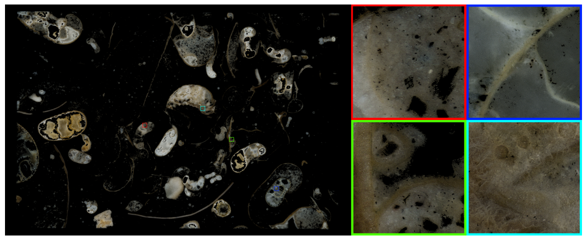


Figure 3: A $19,008 \times 12,672$ pixel image is reconstructed from our data structure. The tiles on the right are 256×256 pixels.

3 RESULTS

We show in Fig. 1 that our work manages to build a compressed tree of 0.7 TB from 2.5 TB of data (3.7 TB decoded) at 45 dB PSNR, interactively visualize 8K at 33 ms per frame, and allow exploration of voxel-scale details through 3D navigation system that displays structures in all dimensions. We use an AMD Radeon PRO ™W6800 GPU on 21.Q4 driver for the evaluation of our implementation. The system has an AMD Ryzen Threadripper PRO 3955WX, 1536 GBs of RAM, and runs Windows 10, Vulkan 1.2, and HIP through Orochi.

We demonstrate that our approach enables visualization of volumes that are: larger in grid size, denser in content, and more varying in modes of render retrieval. Our method builds a more tight-fit tree structure that captures sparsity and compresses through wavelet transform, resulting in a compact representation. We find that it works interactively with recent hardware while relying on generic system memory streaming methods to render on GPU. We also show that it is possible to exploit the hierarchical nature of this data structure to accelerate traversal and that hardware-specific utilization of register spills is a differentiator in performance.

In future, we would like to explore ray-tracing hardware in higher levels of the tree for traversal, subgroups and workgroup shared memory for data movement, directed acyclic graphs, frequency-domain transformations, and neural methods for compression, and new web APIs with alternative streaming methods for mobile.

REFERENCES

Mitko Aleksandrov, Sisi Zlatanova, and David J. Heslop. 2021. Voxelisation Algorithms and Data Structures: A Review. *Sensors (Basel, Switzerland)* 21 (2021).
Alejandro Graciano, Antonio J. Rueda-Ruiz, Adam Pospísil, Jiří Bittner, and Bedrich Benes. 2021. QuadStack: An Efficient Representation and Direct Rendering of Layered Datasets. *IEEE Transactions on Visualization and Computer Graphics* 27 (2021), 3733–3744.
Ken Museth. 2021. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. *ACM SIGGRAPH 2021 Talks* (2021).
Ingo Wald, Gregory P. Johnson, Jefferson Amstutz, Carson Brownlee, Aaron Knoll, J. Jeffers, Johannes Günther, and Paul A. Navrátil. 2017. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* 23 (2017), 931–940.