Eurographics Symposium on Parallel Graphics and Visualization (2025)
G. Reina, S. Rizzi, C. Gueunet (Editors)

GPU Volume Rendering with Hierarchical Compression Using VDB

Stefan Zellmann[1], Milan Jaros[2], Jefferson Amstutz[3], and Ingo Wald[3]

[1]University of Cologne [2]IT4Innovations, VSB–Technical University of Ostrava [3]NVIDIA

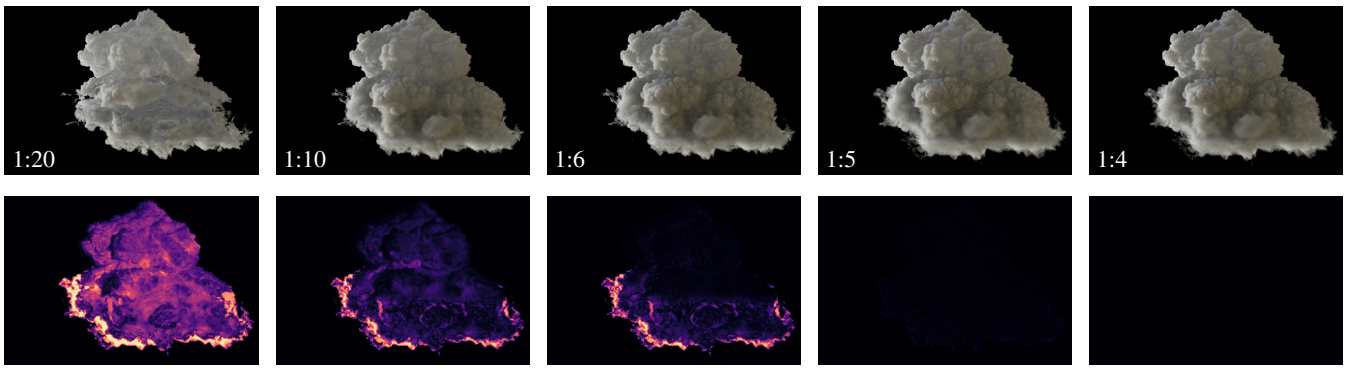


Figure 1: *WDAS cloud data set compressed at different rates. Top row: path traced images of the 2K data set compressed with our proposed method. Bottom row: difference images generated with ΔLIP comparing the compressed ones with renderings of the uncompressed data set.*

Abstract

We propose a compression-based approach to GPU rendering of large volumetric data using OpenVDB and NanoVDB. We use OpenVDB to create a lossy, fixed-rate compressed representation of the volume on the host, and use NanoVDB to perform fast, low-overhead, and on-the-fly decompression during rendering. We show that this approach is fast, works well even in a (incoherent) Monte Carlo path tracing context, can significantly reduce the memory requirements of volume rendering, and can be used as an almost drop-in replacement into existing 3D texture-based renderers.

1. Introduction

Ray-marched volume rendering in scientific visualization is a well-established shading style that virtually every 3D visualization application implements. One of the research challenges in scientific visualization that remains is handling the ever-growing size of the data that is visualized. GPUs are typically chosen for volume rendering due to their superior memory bandwidth and texture sampling routines, but this comes at the cost of limiting (often severely) the size of the data sets that can be rendered. Possible solutions to that are data parallelism and multi-GPU rendering [WJZ23], or out-of-core rendering [SCRL20].

Another option to handle the size increase is to use compression. To make use of standard software for this, one could reach for ZFP [Lin14] with its fixed-rate compression scheme. As ZFP's fixed-rate algorithm is block-based, implementing a GPU renderer would require that when blocks are loaded from memory, many samples are taken across multiple threads to amortize the decompression costs. For Monte Carlo volume rendering with scattering and incoherent memory access patterns, this requires using ray wavefronts, barrier synchronization between bounces, sorting rays for coherency, and other strategies that result in overly complex control flow. From an engineering perspective, what is desirable though is random access as known from using 3D dense textures, so that renderers can devote one GPU thread to each light path.

In this paper we propose to hierarchically encode—and by doing so compress—the volume data, allowing for per-thread decompression in a GPU compute kernel. We use the sparse voxel representation (VDB) for that. VDBs, so named because they represent volumetric grids that share similarities with B+trees [Mus13], are popular in production rendering and industry-strength libraries exist that are optimized for construction and sampling on NVIDIA GPUs. As such, VDBs are often used for data sets like the one shown in Fig. 1.

Our goal is to evaluate if the VDB representation and tools are generally useful for volume rendering in scientific visualization (sci-vis). In sci-vis, volumetric representations beyond structured-regular grids have become more important over the years [SZD*23]. In practice, this means that volume rendering and sampling libraries need to maintain multiple data structures for sampling volumes, including AMR [WZU*21], unstructured

DOI: 10.2312/pgv.20251152

grids [WMZ22] or particle representations [MZS*24]. An overarching question we seek to answer in our research, and which goes beyond the scope of this paper, is if a data structure like VDB with its efficient and convenient to use GPU implementation can be used to replace these data types by resampling the data. In this paper, we explore this question for structured-regular volume representations.

Our main contributions are as follows:

- A fixed-rate compression algorithm with hierarchical encoding implemented with OpenVDB [Mus13],
- an interactive volume path tracer decoding such data on-the-fly on the GPU, and
- an evaluation of the framework and algorithm comparing against dense textures and compression with ZFP.

We also integrated our path tracer, which is realized with CUDA and OptiX, with ANARI. ANARI is an emerging standard for 3D scientific visualization in C++. This allows us to test the framework using VTK and ParaView.

2. Related Work

In this section we review prior work on volume rendering with compression. Our focus lies on block-based compression, which is often used for direct volume rendering (DVR). We are specifically interested in the subset of volume compression algorithms that are used to accelerate renderers, while other (volume) compression algorithms, e.g., ones that require one to decode the whole data before sampling, are of minor interest to us. We also look at related works on hierarchical volume encoding in general, although the majority of these papers does not focus on compression, but on providing spatial indices for fast 3D data retrieval or space skipping. We conclude the section with an overview of VDB grid representations and the OpenVDB and NanoVDB frameworks.

2.1. Block-Based Volume Compression

As 3D uniform grids do not adapt to the topology of the underlying data, their memory footprint increases proportionally when increasing the grid resolution. Hence, compression has traditionally been the focus of volume rendering-related works. One way to compress the data is using wavelet transformed blocks in combination with run-length encoding as proposed by Kim and Shin [KS99]. Back then, volume renderings could not be produced in real-time, but renderers still required a caching data structure to amortize decompression costs. Block-based compression remained popular when volume rendering became interactive due to GPU texture sampling. Schneider and Westermann [SW03], e.g., proposed a block-based algorithm quantizing the volume across different frequency bands and hierarchically within each block, and decompressing it in a fragment shader.

In general, block-based compression using the discrete wavelet transform (DWT) or quantization have been very popular for direct volume rendering. According to the state-of-the-art report by Rodríguez et al. [RGG*14], a generic GPU-based compressed direct volume rendering architecture is centered around encoding blocks during preprocessing, and streaming and decoding those compressed blocks while rendering (cf. Fig. 2 of that report). We also refer the reader to the state-of-the-art report for a general overview on related work on compressed volume rendering that goes beyond the scope of our paper.

Block-based data compression for volumetric rendering is still very popular, which is in parts attributed to the compression framework ZFP [Lin14]. ZFP has become a de facto standard in the HPC community to compress 3D volume and other simulation data. We note that ZFP is a floating-point compression algorithm well applicable beyond 3D volume rendering, and is in fact able to compress higher dimensional data, while the scope of our paper is on 3D volume rendering specifically. ZFP's fixed-rate compression algorithm encodes blocks of size $4 \times 4 \times 4$ containing floating-point values by first converting them to a common fixed-point format. This is achieved by factoring out the largest exponent of floating-point values within each block. The resulting normalized values are converted to a two's complement fixed-point format. Blocks are then spatially decorrelated by converting to a different basis based on a tensor product transformation. Doing so allows the implementation to incorporate different types of transforms, including DWT, discrete-cosine transform (DCT), and others. The resulting transform coefficients are then encoded per bit plane. As with the other block-based compression algorithms, decompression is only efficient when using caches.

Recent work by Usher et al. [UDK23] ported ZFP to WebGPU. Their framework decompressed blocks on-the-fly to extract ISO surfaces using marching cubes. Follow-up work by Dyken et al. [DUK24] extracted ISO surfaces on-the-fly using ray marching and progressive streaming of blocks. We are not aware of any other work that specifically uses a scheme like that applying ZFP to direct volume rendering (DVR); the closest match we found is the reference renderer that Rapp et al. [RPD22] use for their evaluation—this renderer compresses a whole set of volume *samples* for a given perspective frame, though. Implementing a DVR ray marcher would be conceptually similar to what Dyken et al. proposed for ISO surfaces. We are not aware of any multi-scattering volume renderer using ZFP or other block-based compression algorithms, but note that caching would be fundamentally different because of the incoherent memory access patterns used.

2.2. Hierarchical Volume Representations

Hierarchical encodings for structured-regular volume data on Cartesian grids are not necessarily targeted at compression specifically—they usually focus on aspects such as level-of-detail (LOD) composition, spatial indexing, or to skip over empty space. In the context of 3D rendering the purpose of that is often to accelerate the computation, be it by reducing the number of samples taken, or by representing far away objects with coarser ones. The transition between hierarchical encoding and compression in general is fluid. We refer the reader to [RGG*14] for works on hierarchical compression for volume rendering. We also note that hierarchical encoding is an important ingredient to compression in general, e.g., through the use of Huffman codes, prefix tries, etc., but here specifically focus on hierarchies that are *spatial* indices.

One popular representation are sparse voxel octrees (SVO) [LK10] that are often combined with techniques like voxel

cone tracing [CNS*11] for rendering. Such encodings are usually used to represent surface data as voxels. GigaVoxels [CNLE09] is a framework that implements this paradigm.

Hierarchical encodings for volume data often focus on out-of-core rendering or empty space skipping. A comprehensive review on works related to that would be out of scope for this paper. Instead we refer the reader to the state-of-the-art report by Sarton et al. [SZD*23]. The encoding closest to VDBs recently proposed by the literature is BrickTree by Wang et al. [WWJ19], which uses a "wide Octree" topology built over massive volume data sets allowing for out-of-core streaming. BrickTree's inner node encoding uses integer IDs to represent the tree structure, which is similar to OpenVDB's inner node representation presented below.

2.3. VDB Grid Representation

We chose to use the industry standard VDB to implement our method. VDBs were developed by Museth [Mus13], who at that time worked for DreamWorks Animation. Museth's VDBs are 3D spatial indices for sparse voxel data with fast, and on average, $O(1)$ random access on modification and retrieval operations. The VDB data structure is well-suited for visual effects rendering of volumetric data from fluid simulation. VDBs are shallow trees with four levels. In the following we describe the tree layout implemented in OpenVDB, the open source library resulting from Museth's paper.

In OpenVDB, leaf nodes store a fixed number of voxels (defaulting to $8 \times 8 \times 8$), and inner nodes have a fixed number of child nodes. The root level has a variable number of children. At each of the levels the VDB data structure stores a *direct access bit mask* that provides direct random access to a binary representation of the local topology. These 64-bit values encode different information depending on the tree level they are on. Internal nodes, also referred to as tiles, e.g., encode child ID or other representative values for the whole tile in the bit mask. Leaf nodes use the bit mask to encode the leaf origin, with 3×20 bits; other bits encode additional information, e.g., if the voxels are compressed or quantized. Leaf nodes also contain a bit mask indicating which voxels are active, and finally a (raw) pointer to the voxel data itself. This representation allows for fast access and modification and also out-of-core streaming, yet the data prevails in address spaces visible to the CPU. OpenVDB provides numerous authoring tools to modify VDBs at runtime and is well-established because of its integration into production renderers like Cycles [Ble24], and into visual effects software such as Houdini [Sid25] and Cinema4D [Koe01].

NanoVDB [Mus21] is NVIDIA's linear VDB implementation that does not depend on CPU address spaces. The whole VDB can be compactly copied using `memcpy` operations and is optimized for retrieving voxel data on the GPU. Support for modification is rudimentary and essentially limited to altering the values of voxels that already in the VDB and are active. Museth [Mus21] describes NanoVDB as a "linear snapshot of an OpenVDB data structure" that "explicitly avoids memory pointers". Tools included with NanoVDB focus on retrieval more so than on modification, such as 0th to 3rd order interpolation, gradient computation, as well as converters to and from OpenVDB. The 32-byte aligned NanoVDB representation is compatible with numerous GPGPU and shading languages, including CUDA that is used by our implementation.

NeuralVDB by Kim et al. [KLM24] is a recent addition to the VDB family of frameworks that aim at lossy compression of the voxel data using multi-layer perceptrons (MLPs). These are trained on the sparse voxel data and are stored at the lower nodes of the VDB that is otherwise equivalent to OpenVDB. The paper notes that online random access via inference is too slow for real-time applications, so the recommended approach of using NeuralVDB is to decode the neural representation into a regular VDB first.

3. Dense to Sparse Texture Conversion

In the following, we present an algorithm to hierarchically encode and compress volumetric data in memory. Our main objectives are efficient decompression with random access on GPUs, support for single-threaded access from GPU compute kernels without the requirement to cache any data that is shared with other threads, and fixed-rate encoding to give us control over the size of the compressed data. We construct that compressed representation with OpenVDB [Mus13]. As OpenVDB is optimized for editing volumes, but not for random access on the GPU, we then convert the compressed volume with NanoVDB [Mus21], which gives us an efficient GPU representation that is linear in memory and optimized for sampling.

3.1. Fixed-Rate Compression Algorithm

OpenVDB provides tools and C++ functions to create sparse from dense volumes, these are however not easily applicable to compression. The integrated tools require one to identify which value represents *background* (i.e., empty space). When converting from dense to sparse, only those voxels that are not background are set and activated and all the others are not. It is easy to adjust those tools to use thresholding by providing the converter with a tolerance value. The behavior is hard to control though because the achieved compression rate is a monotonous, yet not polynomial function of the tolerance value; increasing the tolerance by just a little might accidentally cull a majority of voxels of interest.

We propose to use a compression algorithm using histogram and local frequency analysis of the volume and by that enabling fixed-rate compression. The user provides a quality parameter in $[0:1]$. As we rely on local frequency response the algorithm is also well-suited for homogeneous regions with noise or gradients.

Algorithm 1 provides a high-level overview. The input consists of the volume itself, its spatial extent represented by W, H, and D (in voxels), and the user-provided quality value. We define point sampling routines on the volume itself—without loss of generality, we only sample the volume at exact (integer) voxel positions though (`volume.value`); as well as routines to activate and set individual voxels on the VDB (`Activate`). The algorithm consists of multiple phases. We assume that a VDB is present and can be manipulated; the VDB is a 4-level tree (created with OpenVDB) using the standard $\{5,4,3\}$ tree size configuration. With this configuration, leaf nodes are of size 2^3 in each dimension (i.e., $8 \times 8 \times 8$ voxels); the inner node level above has 2^4 children per dimension (i.e., size $16 \times 16 \times 16$), and so on. The root node level contains as many 2^5-sized inner nodes as necessary to cover the volume extent.

In the first phase we compute the histogram for the volume to

Algorithm 1 Fixed-rate compression and conversion to VDB.

```
1:  function COMPRESS(volume, W, H, D, quality)
2:      hist = COMPUTEHISTOGRAM(volume)
3:      I = ARGMAX(hist)
4:      B = volume.valueAt(I)                          ▷ background value
5:
6:      brickSize = int3(2**5)
7:      numBricks = int3(W,H,D/brickSize)
8:
9:                                                     ▷ Compute brick ranges:
10:     for brickID ∈ numBricks do
11:         lo = brickID * brickSize
12:         hi = (brickID + 1) * brickSize
13:         valueRanges[brickID] = volume.range(lo,hi)
14:     end for
15:     brickRefs = ENUMBRICKS()
16:     SORTBY(brickRefs, valueRanges, SimilarityFunc(B))
17:
18:                                                    ▷ Activate important voxels:
19:     bricksToActivate = numBricks * quality
20:     for Brick ∈ brickRefs do
21:         for Voxel ∈ Brick do
22:             ACTIVATE(Voxel, volume.value(Voxel))
23:         end for
24:         if bricksActive++ > bricksToActivate then
25:             break
26:         end if
27:     end for
28: end function
```

determine the background value. As we assume sparsity the value associated with most of the voxels becomes background; we assume that this value is a good representative of empty space, i.e., other empty voxels have a value close to this one.

In the second phase we form bricks of size $32 \times 32 \times 32$ to cover the original volume, so that each brick covers 64 leaf nodes of the VDB. We compute value ranges for each brick using the original volume.

In the third phase we sort those bricks using a list of references, using the value ranges and a similarity metric. We sort the bricks in descending order so those whose value contribution is most similar to the background value come first. We finally iterate over the sorted brick reference list and activate all the voxels covered by the brick until a predetermined number of bricks was consumed. By linearly mapping the number of bricks to process to the user-provided quality value we achieve fixed-rate compression.

To classify voxels as important, we compare their values—and the values in a local neighborhood—to the background value. We do this based on the brick decomposition from before, as classifying individual voxels would have an impractical memory footprint.

We classify bricks as similar to the background value using their value range; for example, given background value B, a brick with value range $[B:B]$ is most similar. Comparing a scalar to a range, we have to heuristically pick a representative point of that range to compute the distance of that point to the background value. We propose and evaluate the following functions:

$$f1 = \min(|lo-B|,|hi-B|) \quad (1)$$

$$f2 = \max(|lo-B|,|hi-B|) \quad (2)$$

$$f3 = |(lo+hi)/2-B|. \quad (3)$$

$f1$ and $f2$ compute the closest and the farthest distance from the scalar to the range given by $[lo:hi]$, respectively, while $f3$ computes the distance to the median of the range. We refer to the metrics as the closest, farthest and median point-in-range metrics and evaluate their impact on compression in Section 5.

3.2. Implementation with OpenVDB and NanoVDB

To implement the above algorithm we use the C++ library OpenVDB performing the compression on the CPU. We then convert the result to NanoVDB, which provides a linear (in memory) representation of the VDB that can be sampled in a shader or compute kernel but does not allow for arbitrary modification.

3.2.1. Compression

We use OpenVDB to implement the fixed-rate compression algorithm. We start from an empty `openvdb::FloatGrid`. In OpenVDB, grids associate trees, the internal representation of the VDB in memory, with transforms (e.g., voxel to object space). This internal representation is a `Tree4<float, 5, 4, 3>`, i.e., the tree has four levels using the $\{5,4,3\}$ layout as described above. The tree provides a function `addTile(level,Coord(),value)` to set and activate cells on each tree level.

We start by always activating the extreme (minimum and maximum) corners of the VDB by just setting the respective voxels to their original value. If we would not do that because these voxels were identified as background, OpenVDB would never include them in the tree and the aspect ratio (world bound size) of the VDB tree would be different from the original volume.

We then create bricks of size 2^5, compute their scalar ranges, and sort them by the similarity metric. Given a user-provided compression rate in percent we can determine the number of bricks we want to activate. We then iterate over the sorted brick list starting at the one closest to the background value and activate as many bricks as desired. This is done by just activating all the voxels (level-0 cells) of that brick. We finally call `tree.prune()` to allow OpenVDB to perform memory optimizations.

We note that either activating all the cells, or no cells at all, of a brick can lead to distinct block patterns. Another option would be to decide the number of voxels we want to activate from the brick based on similarity, too, e.g. that starting at a certain threshold we activate half the number of voxels but twice the number of bricks. We leave such optimizations as future work as we would require a (possibly perceptual) metric telling us which voxels to activate.

Regarding compression performance we note a $1:1$ relationship between activated voxels and final output size. We also note that activating a whole tile (`addTile()`, see above) on a level higher than leaf node level 0 results in OpenVDB just setting all the voxels in that tile to the same value, so there is no additional compression to gain from using hierarchical or level-of-detail encoding here.

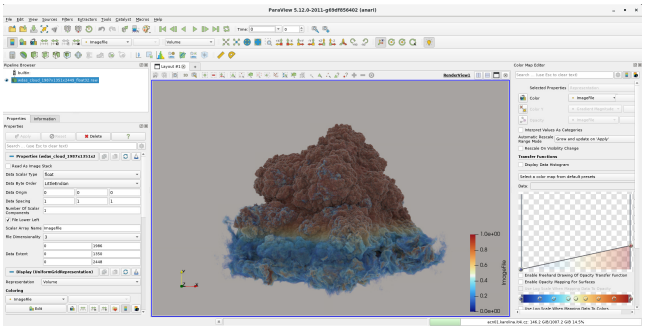


Figure 2: *WDAS cloud loaded in ParaView using the* `raw` *format encoding. Our renderer is integrated using the ANARI interface, structured-regular grids are automatically compressed and converted to NanoVDB.*

After the VDB was created we convert the OpenVDB grid to NanoVDB using builtin tools. The NanoVDB representation is linear in memory, can no longer be modify, but can be copied to the GPU with `cudaMemcpy()`.

3.2.2. On-the-Fly Decompression

The NanoVDB representation is amenable to sampling on the GPU. NanoVDB provides C++ utility functions that can be called from host as well as CUDA device code. This is achieved through accessor and sampler classes. The samplers support zeroth and first-order interpolation similar to dense textures; samplers using cubic or higher order interpolation are also available. The samplers use nearest neighbor or trilinear interpolation on the voxel level (level-0 of the sparse tree) and substitute the background value where no voxels are available.

4. ANARI Renderer

Given our conversion algorithm plus texture sampling routines on the GPU we can implement a renderer. We chose to give the renderer an ANARI API. ANARI support has recently been added to ParaView [WZA*24] so our renderer can be evaluated using standard visualization tools and be compared with other renderers that also use the ANARI standard (cf. Fig. 2).

ANARI standardizes direct volume rendering with structured-regular spatial fields—the exact type we target with our optimization—but only describes what data is rendered and not how. We integrate the VDB optimization into the ANARI renderer Barney [WZA*24], which is a multi-GPU wavefront path tracer implementing volume rendering using Monte Carlo free-flight distance sampling with Woodcock tracking. Multi-GPU rendering is implemented using Wald et al.'s ray queue cycling method [WJZ23].

We hide our compression algorithm behind an ANSI-C API whose input is the structured-regular field representation of ANARI and that outputs a NanoVDB grid. This allows us to hide all the details related to compression behind the API. The use of VDB is transparent to the user of ANARI, who sets up the data as though it were an ordinary structured-regular field, with only a ANARI parameter indicating the data is sparse, and another parameter setting the desired compression rate. Internally, we replace the device texture object with a NanoVDB accessor and sampler that we use during sampling from inside Barney's volume shader.

Barney uses volumetric scattering for shading computations. By default, rays inside the volume scatter until canceled by Russian Roulette, and radiance is contributed via image-based lighting through an HDRI light source, or by an ambient light source constantly illuminating the scene from all directions. ANARI and Barney provide a render graph so that more than just a single volume can be present in the scene at a time, and mixed surface and volumetric scenes are supported. For rendering volumes, Barney implements two operations: traversal using majorant densities for accelerated free-flight distance tracking, and random access sampling into the spatial field itself.

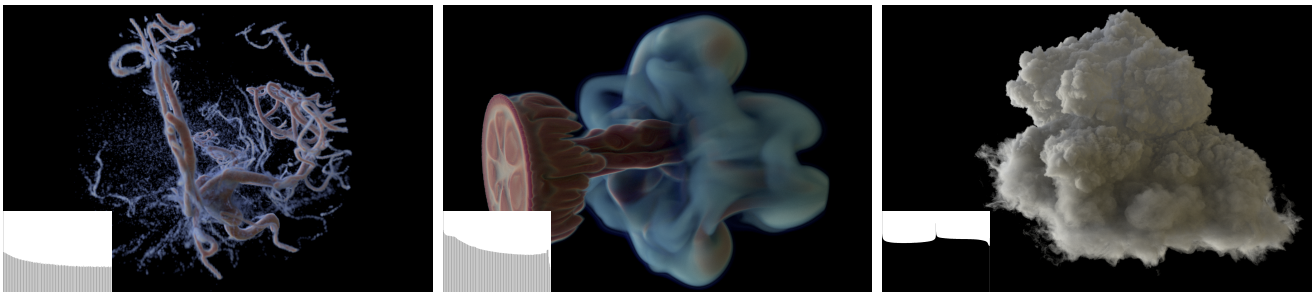
Traversal is realized using macrocell grids that store min/max ranges of the density the cell represents. These min/max ranges are used as lookups into an RGBA transfer function; the α component of the transfer function serves as extinction coefficient and is also used to provide majorants. These majorants are recomputed whenever the transfer function changes, based on the min/max ranges.

By traversing the majorant grid, the path tracer can adapt the sampling rate to how homogeneous the density is inside the macrocells. Cells that are empty can be skipped over; in cells that are homogeneous, the majorant density is closer to the reconstructed density, which allows the Woodcock tracking algorithm to take bigger steps so that fewer samples are taken. The grid is traversed using the 3D digital differential analyzer (3D-DDA) algorithm as in [SKTM11]. When a non-empty cell is encountered, Woodcock tracking is performed inside that cell. If a valid free-flight distance was found, traversal stops; otherwise, it continues to the next cell or until the ray leaves the grid.

With VDB, it is possible to implement a hierarchical version of DDA. Some first experiments with that showed diminishing returns for that approach. Diminishing returns of using hierarchical space skipping data structures for volumetric path tracing in the presence of RGBA transfer functions were also reported by Zellmann et al. [ZWS*24]. Using a uniform grid for traversal also allows us to compare the sampling performance with structured-regular volumes using dense textures. Therefore, we opted to use a uniform grid with non-hierarchical traversal also for the VDB texture representation.

5. Evaluation

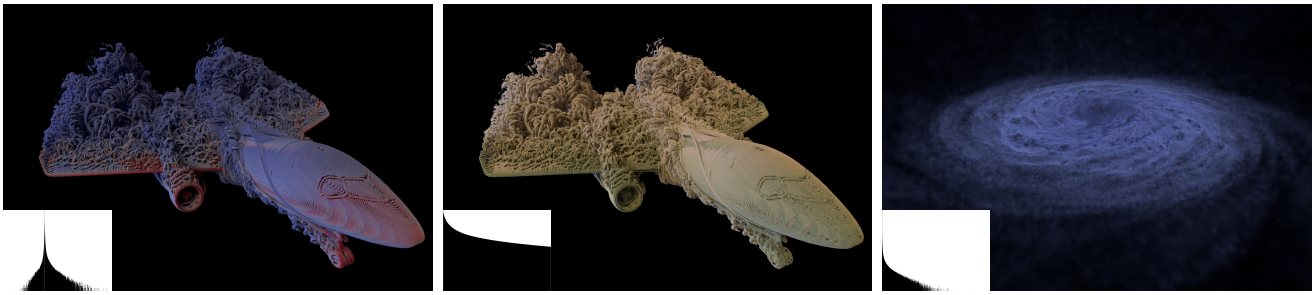
We evaluate the quality and performance of our compressed representation for direct volume rendering. For that we use the structured-regular volume data sets from Table 1. The `aneurism` data set is particularly sparse. The `WDAS cloud` data set by Walt Disney Animation Studios [Wal18] is originally available as a VDB and was resampled to a structured-regular representation retaining the original 2K resolution. As we also have access to the original VDB, this gives us the possibility to compare back-and-forth conversion using our algorithm. The `airplane` data set was simulated with OpenFOAM; we use the Q-criterion vorticity field which is usually sparse compared to other variables. The data was voxelized using the MESIO library [MvJB22], as detailed in [FMB*25]. We have access to two versions of that data set: one that represents the



Aneurism
256 × 256 × 256

Heptane
302 × 302 × 302

WDAS Cloud
1987 × 1351 × 2449



Airplane Q-Vorticity
2450 × 1162 × 4000

Airplane Q-Vorticity (filtered)
2450 × 1160 × 4000

Galaxy B-field
3000 × 3000 × 3000

Table 1: *Evaluation data sets. We use data sets of different sizes and with different degrees of sparseness; in the bottom-left corner we show log-scale histograms of the original structured-regular data.*

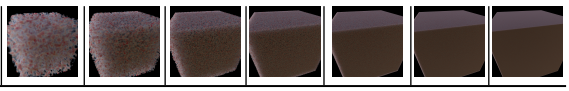
vorticity, and a filtered version of that field where $Q \in [0 : 500K]$, retaining only 4.4 % of the original voxels. We also use a high-resolution, non-cosmological isolated galaxy data set for the evaluation. We utilize an example from [WS23], designed to resemble a Milky Way-type galaxy. The simulation incorporates magnetohydrodynamics (MHD) along with standard subgrid physics commonly employed in galaxy formation studies, including gas dynamics with a magnetic field. The simulation was produces using ChaNga [JGM*08] and subsequently converted to a floating-point volume on a structured-regular grid. To simplify the comparison we assume voxels to be represented with 32-bit floating-point values; we convert `aneurism` and `heptane` to that format whose voxels use 8-bit values.

5.1. Compression Rate

Data Set	Uncompressed	1:8	1:4	1:2	1:1
Aneurism	67.1 MB	9.3 MB	16 MB	16 MB	16 MB
Heptane	110 MB	15 MB	30 MB	35 MB	42 MB
WDAS	26.3 GB	3.5 GB	5.8 GB	5.9 GB	6.2 GB
Airplane	45.6 GB	5.7 GB	12 GB	24 GB	48 GB
Airplane (f)	45.5 GB	4.5 GB	4.5 GB	4.5 GB	4.6 GB
Galaxy	108 GB	14 GB	29 GB	57 GB	91 GB

Table 2: *Actual size in memory of the NanoVDB representation when compressing at different rates.*

We first evaluate if, by using our fixed-rate encoding, the target compression rate is achieved and compress the data set with different ratios from 1:8 to 1:1 (the latter meaning the targeted reduction in size is zero percent). In the latter case the algorithm is guaranteed to be lossless but will only activate voxels that are not considered empty, so the actual size will be lower than the original one. To assess this we report the actual size in memory of the resulting NanoVDBs in Table 2.



	32^3	64^3	128^3	256^3	512^3	1024^3	2048^3
Size							
CUDA	131 KB	1.0 MB	8.4 MB	67 MB	537 MB	4.3 GB	34 GB
VDB	440 KB	1.4 MB	9.1 MB	71 MB	564 MB	4.5 GB	36 GB
FPS							
CUDA	42.0	27.0	22.1	19.9	18.4	17.1	16.3
VDB	31.9	22.0	17.8	16.2	15.2	14.0	13.3

Table 3: *Size in memory and frame rate per second (FPS) when compressing truly dense data from a noise texture. We compare ours (VDB) against a renderer using dense GPU textures (CUDA). Even in this adversarial case the memory overhead for large-scale data sets does not exceed 10 %.*

We are also interested in the worst case we have to expect when losslessly compressing a volume that is not sparse. This is an extreme case that we simulate by generating synthetic volume data sets. We generate noise textures of different sizes to fill the structured-regular grids with voxels so that no local neighborhood is considered empty by our algorithm. We report results for this experiment in Table 3. For large-scale data we observe that the over-

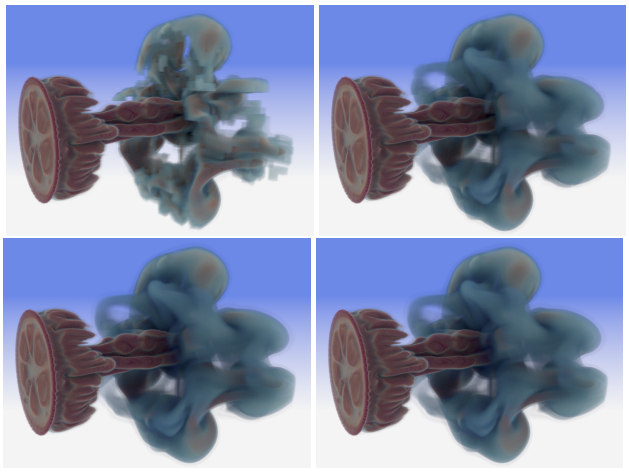


Figure 3: *Our algorithm assigns homogeneous blocks to regions that are considered nearly empty, and voxelizes regions that are not. The compression rates shown are: Top-left: 1:8, top-right: 1:5, bottom-left: 1:4, bottom-right: 1:3. Whether blocks become visible depends on the compression rate and the transfer function.*

head does not exceed 10 % of the original size. We also found, although without a formal evaluation, that there is no correlation between sparseness and render performance and that NanoVDB performs equally well on data that is dense. This suggests that VDBs are a viable replacement even for dense textures.

5.2. Compression Quality

We first evaluate the compression quality using similarity metrics comparing the compressed representation with their dense texture counterparts. These results are summarized in Table 4; we tabulate mean squared error (MSE) and peak signal-to-noise-ratio (PSNR) for the test data sets. As expected, the measured quality is a function of sparseness. While giving a first impression, similarity metrics traditionally have limitations making it hard to draw conclusions relating the metric to the perceived quality. What distinguishes our compression method from other methods is that we encode some values—those that are closer to the VDB background value; i.e., some local features will be perceived as blocky or have missing features, while other volumetric regions will appear as if compressed with a lossless algorithm (cf. Fig. 3). This is particularly hard to assess from just looking at similarity metrics like MSE or PSNR. What we can conclude from those results though is at which compression rate our algorithm achieves lossless encoding. We observe that many of our data sets can be compressed by 25 % and higher without any quality loss at all.

We then evaluate the impact of the closest, farthest, and median point-in-range metrics from Section 3.1. We exemplarily present results for the WDAS cloud in Fig. 4, which indicate superiority of the farthest and median over the closest point-in-range metric. We observed similar outcomes for experiments with the data sets not shown here, leading us to the conclusion that these two metrics seem generally superior.

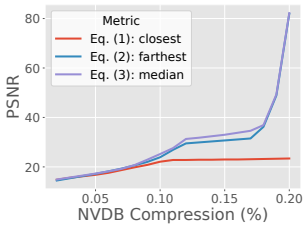


Figure 4: *Similarity metrics and their impact on quality. The results represent PSNR for different compression rates using the WDAS cloud data set. Graphs for the other data sets look similar; we found the farthest and median point-in-range similarity metrics to outperform the closest point-in-range metric.*

5.3. Comparison with Dense Texture Multi-GPU Volume Renderer

We compare the rendering performance of VDBs to that of using random access with dense textures. For that we use the Barney renderer with the extensions described in Section 4 and either activate or deactivate VDB compression. In the former case, Barney will encode the volume data sets using VDB as described above, while in the latter case the structured-regular grid representation is used. Barney represents structured volumes with 3D CUDA textures.

We use an NVIDIA A100 multi-GPU node for the evaluation. Not all our data sets fit into the 40 GB GPU memory, so we have to use two GPUs for the aircraft and three GPUs for the galaxy data sets, respectively. Barney uses data parallel rendering in this case, distributing the data evenly across the available GPUs, and uses ray queue cycling [WJZ23] for volume path tracing.

We present average frame rates with our volume path tracer for the rendered images in Table 5. These frame rate estimates indicate how long it takes to render noisy convergence frames; the images in Table 1 represent converged results over 1K convergence frames. To estimate the framerates we chose the $1:8$ compression ratio across all data sets, but note that framerates are stable within small error margins regardless of the size of the VDB. While not faster, we observe that software trilinear interpolation with NanoVDB achieves competitive frame rates compared to using 3D CUDA textures. As the NanoVDBs always fit into one GPU, our algorithm outperforms dense textures on the galaxy data set that requires rendering on three GPUs.

5.4. Quality Comparison with ZFP

We also compare the quality of our compression with that of ZFP. We do not implement a full ZFP renderer due to the technical debt involved. Instead, we compress and then decompress the volume data sets from Table 1 with ZFP using different compression rates. The decompressed volumes we compare to the original, uncompressed data sets, allowing us to compute MSE and PSNR presented in Fig. 5.

We are also interested in a qualitative comparison of the two compression algorithms given rendered images. For further analysis we implemented an interval-based implicit ISO surface ray marcher to extract the 0.5 ISO surface of the WDAS cloud data set. We show renderings with both our algorithm as well as with

Rate	10 %		20 %		30 %		40 %		50 %		60 %		70 %		80 %		90 %		100 % (1:1)	
Data Set	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR	MSE	PSNR
Aneur.	1e-6	59.0	2e-9	86.7	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞
Heptane	5e-4	32.7	1e-5	49.0	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞
WDAS	4e-3	23.9	6e-9	81.9	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞
Airplane	7e-9	81.8	6e-9	82.3	5e-9	82.9	4e-9	83.6	4e-9	84.4	3e-9	85.4	2e-9	86.7	1e-9	88.5	7e-10	91.7	1e-10	119
Airpl. (f)	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞	0.00	∞
Galaxy	2e-16	156	5e-17	163	2e-18	169	6e-19	175	7e-19	182	7e-20	192	1e-21	209	0.00	∞	0.00	∞	0.00	∞

Table 4: *Compression rate statistics. We report mean squared error (MSE, lower is better) and peak signal-to-noise ratio (PSNR, in dB, higher is better) for the data sets from Table 1. PSNR = ∞ indicates lossless compression.*

Data Set	# GPUs (dense)	CUDA texture	VDB (ours)
Aneurism	1	75.9	55.8
Heptane	1	49.2	42.1
WDAS	1	25.4	24.2
Airplane	2	20.0	18.2
Airplane (f)	2	19.5	17.4
Galaxy	3	18.6	34.8

Table 5: *Frame rates achieved for volumetric path tracing. We compare our VDB encoding to encoding the structured data with dense 3D CUDA textures. In that case, multi-GPU rendering is required for the airplane and galaxy data sets, where we also report the number of GPUs used.*

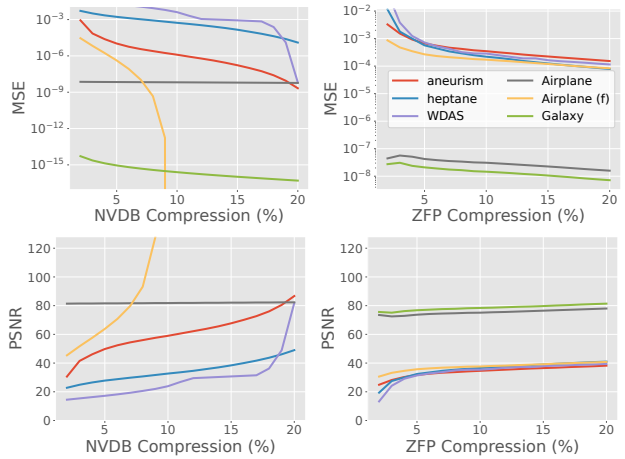


Figure 5: *Quality comparison of our method with ZFP. We report mean squared error (MSE, lower is better) and peak signal-to-noise ratio (PSNR, in dB, higher is better) for the data sets from Table 1, for compression rates between 1-20 %.*

a ZFP back-and-forth compressed and decompressed structured-regular volume, in Fig. 6, as well as difference images using ꟻLIP [ANA*20].

5.5. VDB and Dense Texture Back-and-forth Conversion

Disney's cloud data set [Wal18] was originally published as VDB. For our purposes, we converted it to a structured-regular format matching the exact resolution of the original VDB. This allows us to compare the compressed size as well as the quality achieved of our compression algorithm to that of the original VDB. Conveniently, Disney's data repository contains the data set in different sizes, each downsampling the original data set by a factor ranging from $\frac{1}{2}$ to $\frac{1}{16}$ along each axis. This allows us to run the experiment using different resolutions of the data set. As the original format is (Open-)VDB, we convert the five data sets to NanoVDB first. The NanoVDB file format allows the data to be compressed internally, either using ZIP or BLOSC—non of which is useful to us as these formats do not support random access. We are careful to deactivate internal compression during conversion.

Data Set	Original	Structured	Size MSE=0	MSE (orig. size)
WDAS 1×	4.1 GB	26.3 GB	5.8 GB	0.001
WDAS $\frac{1}{2}$×	590 MB	3.3 GB	605 MB	0.010
WDAS $\frac{1}{4}$×	92.5 MB	413 MB	91.7 MB	0.000
WDAS $\frac{1}{8}$×	17 MB	52.2 MB	15.3 MB	0.000
WDAS $\frac{1}{16}$×	5.1 MB	6.7 MB	2.95 MB	0.000

Table 6: *Back-and-forth conversion of the WDAS cloud's original VDB to dense texture and back to our NanoVDB. We report the size of the original NanoVDB, the size of the dense grid, the size of ours when targeting zero error (MSE=0), and MSE when compressing the structured grid so the size is exactly that of the original.*

In Table 6 we present the results of our experiments. We report the size of the original data converted to NanoVDB and the size of the corresponding structured-regular grid. We further report the smallest size in bytes our algorithm achieves while compressing the data with zero error (MSE=0). For the original resolution, e.g., we observe that compression without error gives a size of 5.8 GB while the original data was 4.1 GB in size. We finally also report the error we realize when compressing the data to the exact size of the original; when compressing the high-resolution data set to match the original size of 4.1 GB, e.g., the MSE we realize is 0.001.

6. Discussion

We presented a fixed-rate compression algorithm to encode structured-regular volumes with OpenVDB. Our main goal was to evaluate the fitness of VDB for sci-vis data when used as a drop-in replacements for dense 3D textures in a GPU-based volume path tracer. From an engineering perspective this is appealing in many ways, as NanoVDB provides an interface that is very similar to that of dense 3D textures, and is equally easy to use.

It was not the main goal of this work to devise a most efficient hierarchical compression algorithm, in a sense that it is generally

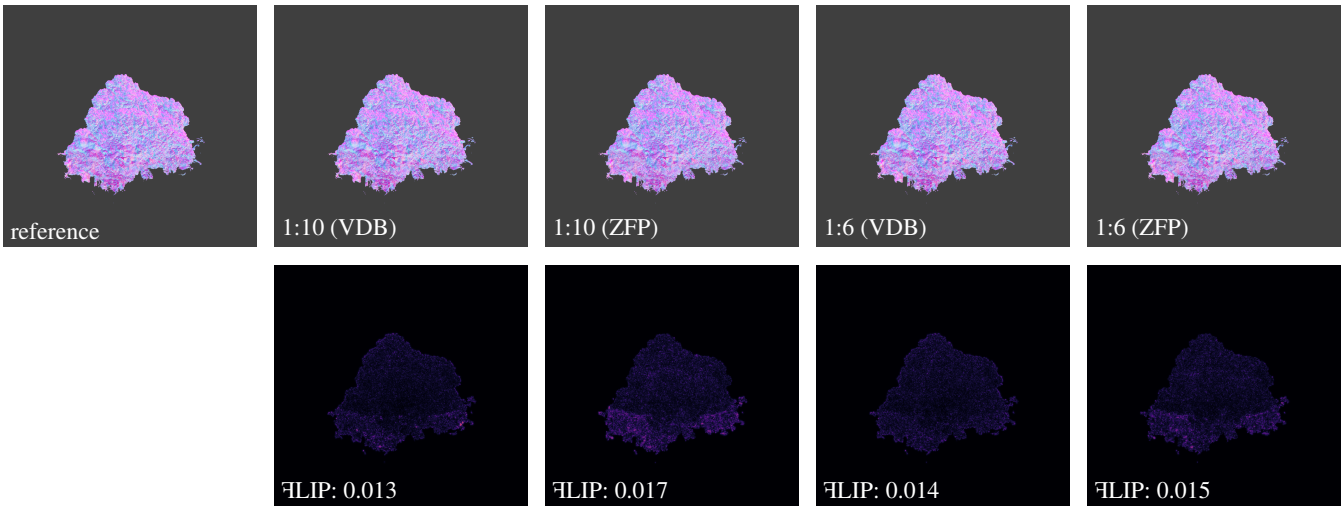


Figure 6: *Direct image comparison between our compression and ZFP (we use* dense *textures to render the latter). The ZFP representation is obtained by compressing and directly decompressing the WDAS cloud, and rendering the resulting structured-regular volume with an implicit ISO surface ray marcher. The color mapping shows reconstructed surface normals. We compare the outcome using ꟻLIP.*

known that hierarchical encoding has beneficial compression properties when applied to volumes [RGG*14]. Much more than that, properties of VDB such as the worst case compression rate when applied to data that is not sparse at all, or their sampling performance when compared to dense textures are much more important to us; and we can conclude that even in adversarial cases, both volume size as well as rendering performance are well within range of the performance of just using dense textures.

With our VDB compression we note that the data should be inherently sparse. But even if it is not, from an engineering perspective, using VDB still has advantages as the size and performance overhead is so small. We however note that sparseness should come from the volume itself and not (only) from the transfer function. Compression is performed in a pre-process and the VDB structure does not automatically adapt to topological changes only induced by the transfer function. The volume must contain a value clearly distinguishable as background, and with high compression rates, adversarial transfer functions can still reveal block artifacts.

It is not our goal in this paper to reduce those artifacts, or to propose the best possible compression algorithm. Optimizations like that are future work and orthogonal to what we propose. Another interesting future work is converting other data structures such as AMR or finite element meshes to VDB using voxelization, to eventually replace all of them with VDB. More research is necessary to determine if this is viable, since especially in the case of AMR, level differences in the hierarchy often span multiple orders of magnitude of space, whereas the standard VDB layouts support a fixed number of consecutive hierarchy levels only.

7. Conclusion

We proposed to use hierarchical encoding with OpenVDB and NanoVDB to compress structured-regular 3D grids for volume rendering with Monte Carlo path tracing. Our main goal was to evaluate the fitness of VDBs for sci-vis volume rendering. We conclude that VDBs are viable alternatives to dense GPU textures that incur only little memory and performance overhead even in adversarial cases. For sparse data as used in our evaluation we can achieve high compression rates at comparably high quality. Qualitatively, our fixed-rate compression algorithm, although much simpler, compares favorably to compression with ZFP if the data is sparse. In the future we want to evaluate if VDBs can also be used to encode other data types typically found in sci-vis, such as finite elements or AMR grids.

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant no. 456842964. This work was also supported by the SPACE project under grant agreement No 101093441. The project is supported by the European High-Performance Computing Joint Undertaking and its members (including top-up funding by the Ministry of Education, Youth and Sports of the Czech Republic ID: MC2304). This work was also supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254).

References

[ANA*20] Andersson P., Nilsson J., Akenine-Möller T., Oskarsson M., Åström K., Fairchild M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–15:23. doi:10.1145/3406183. 8

[Ble24] Blender Project, Cycles Developers: Cycles Open Source Production Rendering. https://www.cycles-renderer.org, 2024. URL: https://www.cycles-renderer.org. 3

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, Association for Computing Machinery, p. 15–22. URL: https://doi.org/10.1145/1507149.1507152, doi:10.1145/1507149.1507152. 3

[CNS*11] Crassin C., Neyret F., Sainz M., Green S., Eisemann E.: Interactive Indirect Illumination Using Voxel Cone

Tracing. *Computer Graphics Forum 30*, 7 (2011), 1921–1930. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2011.02063.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.02063.x, doi:https://doi.org/10.1111/j.1467-8659.2011.02063.x. 3

[DUK24] DYKEN L., USHER W., KUMAR S.: Interactive Isosurface Visualization in Memory Constrained Environments Using Deep Learning and Speculative Raycasting. *IEEE Transactions on Visualization and Computer Graphics* (2024). doi:10.1109/TVCG.2024.3420225. 2

[FMB*25] FALTÝNKOVÁ M., MECA O., BRZOBOHATÝ T., ŘÍHA L., JAROŠ M., STRAKOŠ P.: Workflow for high-quality visualisation of large-scale CFD simulations by volume rendering. *Advances in Engineering Software 200* (2025). doi:10.1016/j.advengsoft.2024.103822. 5

[JGM*08] JETLEY P., GIOACHIN F., MENDES C., KALE L. V., QUINN T.: Massively parallel cosmological simulations with ChaNGa. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (2008), pp. 1–12. doi:10.1109/IPDPS.2008.4536319. 6

[KLM24] KIM D., LEE M., MUSETH K.: NeuralVDB: High-resolution Sparse Volume Representation using Hierarchical Neural Networks. *ACM Trans. Graph. 43*, 2 (Feb. 2024). URL: https://doi.org/10.1145/3641817, doi:10.1145/3641817. 3

[Koe01] KOENIGSMARCK A. V.: *Maxon Cinema 4D 7*. Peachpit Press, USA, 2001. 3

[KS99] KIM T., SHIN Y.: An efficient wavelet-based compression method for volume rendering. In *Proceedings. Seventh Pacific Conference on Computer Graphics and Applications (Cat. No.PR00293)* (1999), pp. 147–156. doi:10.1109/PCCGA.1999.803358. 2

[Lin14] LINDSTROM P.: Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2674–2683. doi:10.1109/TVCG.2014.2346458. 1, 2

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, Association for Computing Machinery, p. 55–63. URL: https://doi.org/10.1145/1730804.1730814, doi:10.1145/1730804.1730814. 2

[Mus13] MUSETH K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (July 2013). URL: https://doi.org/10.1145/2487228.2487235, doi:10.1145/2487228.2487235. 1, 2, 3

[Mus21] MUSETH K.: NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In *ACM SIGGRAPH 2021 Talks* (New York, NY, USA, 2021), SIGGRAPH '21, Association for Computing Machinery. URL: https://doi.org/10.1145/3450623.3464653, doi:10.1145/3450623.3464653. 3

[MvJB22] MECA O., ŘÍHA L., JANSÍK B., BRZOBOHATÝ T.: Toward highly parallel loading of unstructured meshes. *Advances in Engineering Software 166* (2022), 103100. URL: https://www.sciencedirect.com/science/article/pii/S0965997822000138, doi:https://doi.org/10.1016/j.advengsoft.2022.103100. 5

[MZS*24] MORRICAL N., ZELLMANN S., SAHISTAN A., SHRIWISE P., PASCUCCI V.: Attribute-Aware RBFs: Interactive Visualization of Time Series Particle Volumes Using RT Core Range Queries. *IEEE Transactions on Visualization and Computer Graphics 30*, 1 (2024), 1150–1160. doi:10.1109/TVCG.2023.3327366. 2

[RGG*14] RODRÍGUEZ M. B., GOBBETTI E., GUITIÁN J. I., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum* (2014). doi:10.1111/cgf.12280. 2, 9

[RPD22] RAPP T., PETERS C., DACHSBACHER C.: Image-based Visualization of Large Volumetric Data Using Moments. *IEEE Transactions on Visualization and Computer Graphics 28*, 6 (2022), 2314–2325. doi:10.1109/TVCG.2022.3165346. 2

[SCRL20] SARTON J., COURILLEAU N., REMION Y., LUCAS L.: Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-of-Core Approach. *IEEE Transactions on Visualization and Computer Graphics 26*, 10 (2020), 3008–3021. doi:10.1109/TVCG.2019.2912752. 1

[Sid25] SIDEFX: Houdini. https://www.sidefx.com/products/houdini, 2025. URL: https://www.sidefx.com/products/houdini. 3

[SKTM11] SZIRMAY-KALOS L., TÓTH B., MAGDICS M.: Free Path Sampling in High Resolution Inhomogeneous Participating Media. *Computer Graphics Forum 30* (2011). 5

[SW03] SCHNEIDER J., WESTERMANN R.: Compression Domain Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (USA, 2003), VIS '03, IEEE Computer Society, p. 39. URL: https://doi.org/10.1109/VISUAL.2003.1250385, doi:10.1109/VISUAL.2003.1250385. 2

[SZD*23] SARTON J., ZELLMANN S., DEMIRCI S., GÜDÜKBAY U., ALEXANDRE-BARFF W., LUCAS L., DISCHLER J. M., WESNER S., WALD I.: State-of-the-art in Large-Scale Volume Visualization Beyond Structured Data. *Computer Graphics Forum 42*, 3 (2023), 491–515. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14857, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14857, doi:https://doi.org/10.1111/cgf.14857. 1, 3

[UDK23] USHER W., DYKEN L., KUMAR S.: Speculative Progressive Raycasting for Memory Constrained Isosurface Visualization of Massive Volumes. In *13th IEEE Symposium on Large Data Analysis and Visualization* (2023). doi:10.1109/LDAV60332.2023.00007. 2

[Wal18] WALT DISNEY ANIMATION STUDIOS: Cloud Data Set. https://disneyanimation.com/resources/clouds/, 2018. 5, 8

[WJZ23] WALD I., JAROŠ M., ZELLMANN S.: Data Parallel Multi-GPU Path Tracing using Ray Queue Cycling. *Computer Graphics Forum 42*, 8 (2023). doi:https://doi.org/10.1111/cgf.14873. 1, 5, 7

[WMZ22] WALD I., MORRICAL N., ZELLMANN S.: A Memory Efficient Encoding for Ray Tracing Large Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics 28*, 1 (2022), 583–592. doi:10.1109/TVCG.2021.3114869. 2

[WS23] WISSING, ROBERT, SHEN, SIJING: Numerical dependencies of the galactic dynamo in isolated galaxies with SPH. *A&A 673* (2023), A47. URL: https://doi.org/10.1051/0004-6361/202244753, doi:10.1051/0004-6361/202244753. 6

[WWJ19] WANG F., WALD I., JOHNSON C. R.: Interactive Rendering of Large-Scale Volumes on Multi-Core CPUs. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)* (2019), pp. 27–36. doi:10.1109/LDAV48142.2019.8944267. 3

[WZA*24] WALD I., ZELLMANN S., AMSTUTZ J., WU Q., GRIFFIN K., JAROS M., WESNER S.: Standardized Data-Parallel Rendering Using ANARI. In *2024 IEEE 14th Symposium on Large Data Analysis and Visualization (LDAV)* (2024), pp. 23–32. doi:10.1109/LDAV64567.2024.00013. 5

[WZU*21] WALD I., ZELLMANN S., USHER W., MORRICAL N., LANG U., PASCUCCI V.: Ray Tracing Structured AMR Data Using ExaBricks. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021), 625–634. 1

[ZWS*24] ZELLMANN S., WU Q., SAHISTAN A., MA K.-L., WALD I.: Beyond ExaBricks: GPU Volume Path Tracing of AMR Data. *Computer Graphics Forum 43*, 3 (2024), e15095. doi:https://doi.org/10.1111/cgf.15095. 5