Interactive Rendering of Large-Scale Volumes on Multi-Core CPUs

Feng Wang*
SCI Institute, University of Utah

Ingo Wald†
Intel Corporation, Now at NVIDIA

Chris R. Johnson ‡
SCI Institute, University of Utah

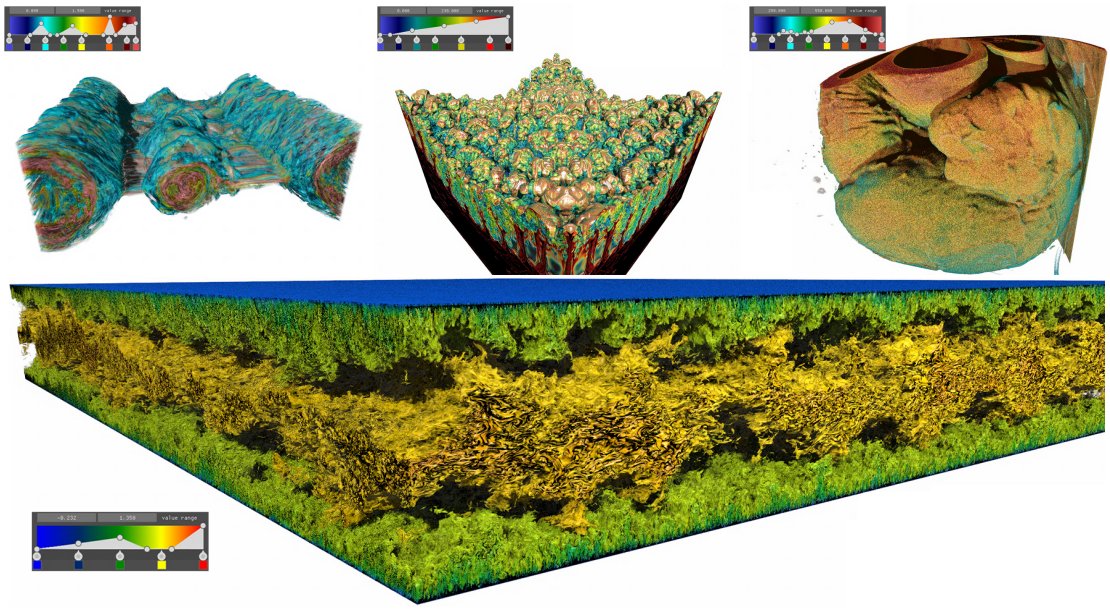


Figure 1: *Screenshots from our high-fidelity interactive volume visualization renderer. Top left: volume rendering of a* 512^3 *magnetic reconnection dataset [15]. Top middle: volume rendering of the* $2048 \times 2048 \times 1920$ *Richtmyer-Meshkov instability (RMI) [6]. Top right: visualization of a* $2048 \times 2048 \times 2612$ *cardiac volume [22]. Bottom: visualization of the* $10240 \times 7680 \times 1356$ *DNS dataset [31]. All images are rendered with surface shading.*

Abstract

Recent advances in large-scale simulations have resulted in volume data of increasing size that stress the capabilities of off-the-shelf visualization tools. Users suffer from long data loading times, because large data must be read from disk into memory prior to rendering the first frame. In this work, we present a volume renderer that enables high-fidelity interactive visualization of large volumes on multi-core CPU architectures. Compared to existing CPU-based visualization frameworks, which take minutes or hours for data loading, our renderer allows users to get a data overview in seconds. Using a hierarchical representation of raw volumes and ray-guided streaming, we reduce the data loading time dramatically and improve the user's interactivity experience. We also examine system design choices with respect to performance and scalability. Specifically, we evaluate the hierarchy generation time, which has been ignored in most prior work, but which can become a significant bottleneck as data scales. Finally, we create a module on top of the OSPRay ray tracing framework that is ready to be integrated into general-purpose visualization frameworks such as Paraview.

Index Terms: Large-scale visualization—Visualization techniques—Progressive Rendering—Visualization design and evaluation methods;

*e-mail: feng@sci.utah.edu
†e-mail: iwald@nvidia.com
‡e-mail: crj@sci.utah.edu

1 Introduction

Interactive visualization of large-scale volumetric data, produced by simulations, astronomical instruments and high-resolution sensors, allows research scientists to explore scientific data, validate hypotheses and discover new knowledge [21]. However, the increasing data resolution in current high-performance computing (HPC) simulations can easily surpass the capabilities of host systems, making interactive visualization of such data challenging. First, loading the full-resolution data into main memory is impractical due to memory limitations. Second, for large datasets, the IO latency incurred by the data loading process is prohibitive in the existing volume renderer, even if all the data fit into memory. The long IO waiting time for large dataset degrades the user experience. Therefore, research on novel techniques for data visualization, processing, storage and IO that scale to extreme-scale data is required to transcend the limitations of current hardware [1].

Current solutions for scalable volume data visualization mostly employ GPU architecture, since it has been shown to be effective for interactive visualization. Prior studies [8, 10, 18] have applied out-of-core approaches, level-of-detail (LOD) techniques, progressive rendering and data compression schemes to overcome GPU memory limitations. However, these approaches inevitably employ extra data structures (e.g., page tables). Meanwhile, they are likely to incur a CPU-GPU data transfer bottleneck in the extreme case that massive missing data are loaded during interactive rendering [17]. Some studies also consider architectures employing CPUs, since the amount of memory directly accessible by a CPU often dwarfs the amount of VRAM available on even the most powerful GPUs. Previous studies have shown that an optimized CPU volume renderer can outperform a GPU renderer for sufficiently large volumes

[23, 36, 39]. However, several studies [3, 20, 32] have focused on distributed parallel rendering on supercomputers, but very few have addressed interactive visualization of large-scale volumes on a single workstation.

In this work, we present an interactive visualization solution for large-scale volumes on multi-core CPU architectures. Our solution allows users to get an overview of the large data in seconds rather than minutes or hours using existing visualization frameworks. We build our approach on a hierarchical data structure – *Bricktree* – that allows for a hierarchical representation of the volume. During the rendering, we stream the necessary data on demand with separate threads and employ ray-guided progressive rendering for data refinement. As an extension of the OSPRay ray tracing framework, which already contains various techniques for visualizing scientific data [19, 39, 43], we build a Bricktree module along with an efficient hierarchy generation tool to support large-scale data visualization on multi-core workstations. Given that OSPRay has been integrated into Paraview and VisIt, our module is ready to be integrated into general-purpose visualization frameworks. Specifically, our contributions in this paper are:

- An interactive visualization solution for large-scale volume visualization, which decouples the data loading and rendering process on multi-core architectures and dramatically reduces the amount of time the user has to wait before being able to explore the data.
- The Bricktree, an efficient and low-overhead hierarchical structure that allows for encoding a large volume into a multi-resolution representation. We also evaluate the structure with several choices of parameters.
- An OSPRay module for large data visualization and a parallel hierarchy generation tool that are ready to be "dropped into" a general-purpose visualization pipeline.

2 Previous Work

Although widely used for visualization of 3D scalar fields, volume rendering remains a computation-, memory- and I/O-intensive task [45]. Several studies have focused on improving the rendering performance by introducing efficient packet BVH traversal [23, 39], empty space skipping [16] and early ray termination [25, 28]. Although efficient for visualizing moderate-size volumes on consumer desktops, these methods struggle to scale to petascale or exascale datasets, since they assume that the entire volume is present in memory. Previous work on large-scale volume rendering can be categorized as:

1) Parallel/distributed rendering on distributed memory systems. These approaches parallelize data processing over many nodes [4]. Research has demonstrated strong scalability on both CPU and GPU clusters [2, 3, 9, 12, 20, 32]. For example, Howison et al. [20] demonstrated that MPI-hybrid parallelism achieves a sublinear raycasting speed-up and is more efficient in terms of overall speed and memory than the MPI-only parallelism. However, previous work [45] has found that the main performance bottleneck of distributed visualization lies in final image compositing rather than in the volume rendering process.

2) Visualization of large-scale volumes on a stand-alone workstation. Much effort has been devoted to the implementation of GPU renderers due to the great hardware interpolation capabilities [5, 44]. Beyer et al. [1] conducted a detailed survey on this topic. Most prior work focuses on overcoming the GPU's memory limitation and tackling this issue by loading only the visible part of the volume into GPU memory [29]. The GigaVoxels system [7, 8], the first to employ this idea, determines the visibility of small blocks "on the fly". Although capable of rendering several billion voxels in real-time, it mainly focuses on entertainment applications and targets sparse volume datasets. CERA-TVR [10] then extended the GigaVoxel paper, targeting scientific visualization user cases. In contrast to GigaVoxel, CERA-TVR is capable of rendering dense volumes and can progressively refine parts of a framebuffer if the size of the visible data exceeds the size of the brick pool. Hadwiger et al. [17] proposed a virtual memory scheme that avoids explicit tree traversal and supports interactive visualization and streaming of petascale 2D image data. However, this approach exhibits IO latency during 3D block construction due to cache misses and requires that all visible data fit into cache.

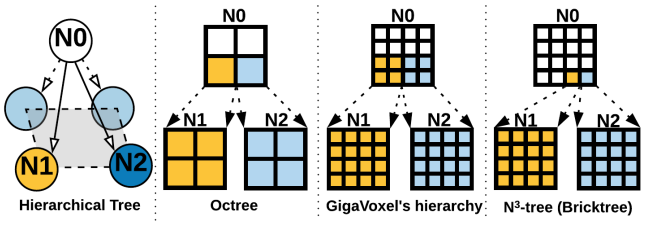


Figure 2: *Comparison of the data structure with an octree, the GigaVoxel's hierarchy and an N^3-tree (Bricktree) in a 2D representation. In a 3D scenario, an octree has a branching factor of 2^3 and allows for decomposing a node into 2^3 cells; the hierarchy used in GigaVoxel [8] has a branching factor of 2^3 and allows for decomposing a node into N^3 cells; an N^3-tree and a Bricktree share the same data structure but have different layouts. Both have a branching factor of N^3 and allow for decomposing a node into N^3 cells.*

In the context of the CPU-based renderers, very little research has focused on large-scale volume rendering on multi-core workstations. As a state-of-art CPU ray-tracing framework for scientific visualization, OSPRay works well for interactive visualization of moderate-size volume data and can potentially be extended to large-scale data. Wu et al. [45] proposed the VisIt-OSPRay system, which can scale to large-scale datasets. To achieve interactive visualization, they integrated OSPRay into VisIt as a backend visualization toolkit and coupled it with PIDX (parallel IO library) [26] for scalable IO. This study achieved interactive visualization, but it took more than 30 minutes to load the DNS dataset into memory with 64 processors on Stampede2.

To address these challenges, we decompose large volumes into a hierarchical representation, with each node representing a small cubic brick. Within the rendering pipeline, we leverage ray-guided streaming and progressive rendering to reduce IO latency.

3 Bricktrees

Adaptive space subdivision and hierarchical data representation are key to interactive visualization of large-scale volumetric datasets [8]. These techniques enable us to load and update the working set on demand to address IO latency and memory limitations. As a popular hierarchical data structure for 3D space subdivision, the octree has been well studied for direct volume rendering [14]. Hierarchical grids feature a theoretically optimal number of traversal steps, and thus have been adopted as general-purpose ray-tracing acceleration structures [24]. Lefebvre et al. [27] proposed the N^3-tree, which is capable of decomposing a node by an arbitrary number N^3 and has a branching factor of N^3. The GigaVoxel system [8] also uses a hierarchical data structure that is similar to the N^3-tree, but with a smaller branching factor of 2^3 (as shown in Figure 2). Other structures, such as kd-trees, have also been used in isosurface rendering with the trend of coherent ray tracing [34, 38, 40].

In this work, we define a hierarchical data structure –"Bricktree"– that allows us to represent a structured volume in a hierarchical, multi-resolution and compressed way (as shown in Figure 3). In order to guarantee that we can safely index all bricks with a 32-bit

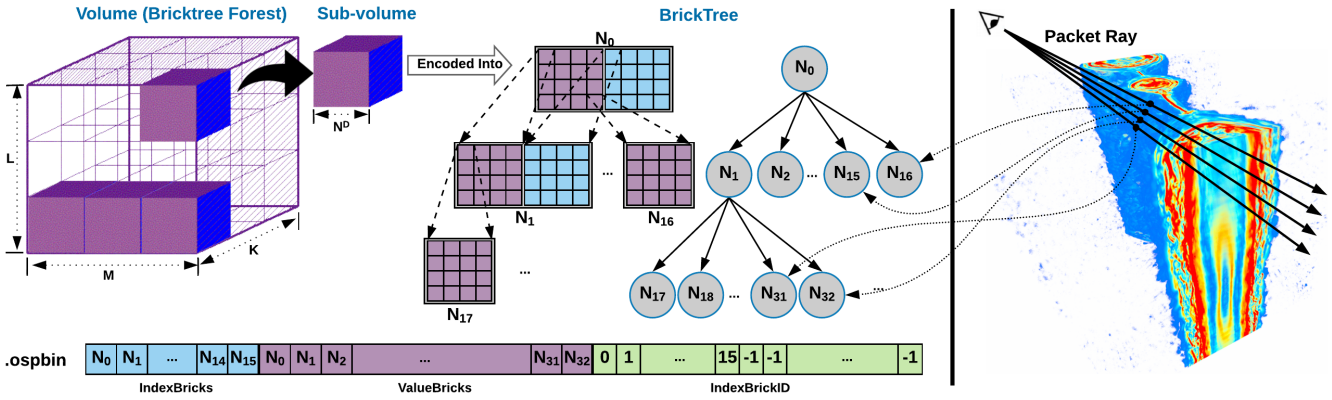


Figure 3: *An illustration of the layout of the Bricktree structure. Each brick (e.g., $N_0, N_1 \ldots$) is represented with a Valuebrick (colored in purple), an indexBrickID and an optional Indexbrick (colored in light blue). The indexBrickID stores the index of the Indexbrick if a node has one. Otherwise, -1 is stored. Both Valuebricks and Indexbricks contain N^3 cells. Each cell encodes a float value in a Valuebrick or an int32 reference in an Indexbrick.*

int, we tile the volume into a "Bricktree Forest", which consists of a list of Bricktrees, rather than encoding the large volume into a single deep Bricktree. Furthermore, this design not only is conducive to parallel tree traversal on multi-core CPU architectures, but also avoids encoding a large empty space when the structured volume is not a perfect cube ($M \neq L \neq K$). Overall, a Bricktree is a generalization of an octree where we subdivide a node by N^3 rather than by 2^3. Furthermore, a Bricktree is similar to the N^3-tree in terms of data structure, but different with respect to data layout in memory. In an N^3-tree node, each cell stores either a data value (for a leaf node) or a reference to the children (for an inner node). In this case, if we expect to store data values for inner nodes to apply progressive rendering, we have to save another index buffer in the large texture and assign an invalid index (-1) for each cell in a brick when no cell has children. The number of those indices adds up to N^3 for an N^3 brick. However, in a Bricktree, we represent bricks with two separate buffers. This choice allows us to save memory and store only one invalid index for an N^3 brick if no cell of this brick has children. Hence, a Bricktree is more suitable for progressive rendering. In the next section, we will introduce the Bricktree layout in detail.

3.1 Data Layout

In our design, a "brick" is an N^3 set of cells. Obviously, each cell stores a data value. We denote a brick that stores data values as a "Valuebrick". Bricks can be thought of as nodes of a tree. We call such a tree a "Bricktree". Each brick can have up to N^3 children, where each child brick is associated with exactly one cell in the parent brick. If a brick does have children, the child indices are stored in an "Indexbrick". Thus, we represent a brick as a Valuebrick and an optional Indexbrick in memory, along with an indexBrickID to correlate the Valuebrick and Indexbrick.

Each cell of a Valuebrick corresponds to a set of voxels in the volume. For a "leaf brick" (e.g., N_{16}), one cell corresponds to exactly one voxel, and the cell's value equals the voxel's value. By contrast, each cell of an "inner brick" (e.g., N_0) typically represents a set of voxels. Accordingly, we can assign a value to each cell. How to calculate this value is the user's choice. In our implementation, we compute this value by averaging over all voxels in this set. Each cell of an Indexbrick refers to a child brick if the current brick has children. A "leaf brick" has no children and thus no Indexbrick. In this case, the indexBrickID of this brick is set to invalid (-1), whereas for an "inner brick", its indexbrickID refers to the right Indexbrick, and the cells in this Indexbrick point to the children. In Figure 3, a Valuebrick is shown in purple and an Indexbrick in light blue.

Multiple Bricktrees can be combined to form a "Bricktree Forest". Recall that this allows us to tile a nonsquare domain efficiently. Figure 3 (left) illustrates a structured volume with dimension $M \times L \times K$, which is organized into a Bricktree Forest. Each Bricktree is specified through a brick size (N), a data type (T) and a tree depth (D), and represents a set of $N^D \times N^D \times N^D$ voxels in the volume.

In memory, each $BrickTree < N, T > (D)$ is represented as three linear arrays:

1. One linear array of Valuebricks, where each Valuebrick contains $N \times N \times N$ values of type T.

2. One linear array of int32 "indexbrickIDs", with exactly one such int32 per Valuebrick. If a given Valuebrick's indexbrickID is -1, it does not refer to an Indexbrick and thus no cell in the brick has children; otherwise, this ID refers to an Indexbrick in the Indexbrick array.

3. One linear array of "Indexbricks", where each Indexbrick contains $N \times N \times N$ int32 indices. Each such index can be -1 (meaning the corresponding cell does not have a child); if the index is greater than or equal to 0, it refers to a brick in the Valuebrick array.

On a file system, the volume is represented by three types of files: 1) one high-level XML file represents the Bricktree Forest, which contains meta-data (N, T, M, L, K etc.) of the input volume; 2) one XML file for each Bricktree that gives high-level information (N, T, D, etc.) of the current bricktree; 3) one binary file for each Bricktree that stores the three arrays (Valuebricks, Indexbricks and indexbrickIDs).

3.2 Bricktree Overhead

Due to the specific layout, our Bricktree is easy to index and the overhead is low. Given a Bricktree with a brick size of 4, a data type of float and a depth of 4 ($Bricktree < 4, float > (4)$), a certain number of 4^3 Valuebricks and 4^3 Indexbricks will be generated when building the tree. In practice, leaf Valuebricks have zero overhead, since each costs exactly 256 bytes for 64 float-type cells. The only overhead lies in the Indexbricks and inner Valuebricks. However, few of these bricks exist relative to the number of leaf bricks.

In a more ideal case, where many of the inner nodes all point to the leaf Valuebrick, we store only about one 32-bit int (4 byte) for a complete leaf Valuebrick. Hence, 4 bytes are used to index 256 "payload" bytes for float data. A less ideal case is that some children of inner nodes are leaves, whereas other children are inner

nodes. Recall that for each inner node, we need to store both a Valuebrick and an Indexbrick, which will result in more memory consumption than in the ideal case. However, the overhead will still most likely be considerably less than that of an octree. Benchmarks in Table 1 show that the in-memory size of a Bricktree with $N = 4$ is much smaller than a Bricktree with $N = 2$, which is an octree. In addition, few bricks are located in the upper levels of the tree. Assuming a brick size of 4, the number of inner nodes goes down by a factor of 64 each time we go up one level in the tree. This holds in practice - the overhead of Bricktree with $N = 4$ is 8.0% on the magnetic reconnection dataset and the DNS dataset. The details of the benchmarks can be found in Section 6. By contrast, the overhead of an octree (Bricktree with $N = 2$) that is built on top of those data is 43% and 72%, respectively.

3.3 Hierarchy Generation in Parallel

As an offline process that is usually run in advance, the performance of reorganizing the volume into multi-resolution bricks is mostly ignored in the volume rendering literature [13]. However, performance becomes a significant bottleneck for large-scale datasets. Fogal et al. reported that the runtime for building a hierarchical structure for RMI (8.1 GB, $2048 \times 2048 \times 1920$) is up to 1.5 hours in the worst case and 13 minutes in the best case [13]. Petascale datasets might take hours or days. The virtual memory architecture of [17] alleviates this problem by constructing the brick at runtime. However, the latency of constructing bricks at runtime will dramatically influence the framerate, especially when the visible data are missing in memory.

Our design also makes hierarchy generation more efficient. It is straightforward to construct the Bricktree Forest in parallel. Rather than parallelizing the process using *tbb :: parallel_for*, our "*ospRawToBricks*" tool employs the *GNU make* command to meet this goal. This choice allows us to specify a customized number of trees to build in parallel in case we consume all the host memory. Normally, *make* will execute only one recipe at a time. By specifying the "*-j*" option, it is possible to execute many recipes simultaneously. Once parameters N, T and D are set, we first generate an index file over the Bricktrees as well as a makefile. The makefile is used to build each Bricktree in parallel. Algorithm 1 shows the algorithm for recursively constructing a Bricktree. Measurements of the performance improvement from parallelization are shown in Figure 9.

Algorithm 1 The recursive function for constructing a Bricktree. N, T is defined as template parameters. $Threshold$ is a customized parameter for "compression".

Input: *llC* - left lower coord; *lvl* - tree level; *lvlWidth* - level width.
Output: *avgValue* - average value of a brick; *vRange* - value range.

```
1:  function BUILDREC(avgValue, llC, lvl, lvlWidth)
2:      cellSize ← lvlWidth/N
3:      brick, vRange
4:      if levelWidth == N then
5:          brick.value[i][j][k] ← input.get(N * llC + offset)
6:          vRange.extend(brick.value[i][j][k])
7:      else
8:          lowerLeft ← N * llC + offset
9:          vRange ← BuildRec(avg, lowerLeft, lvl + 1, cellSize))
10:         brick.value[i][j][k] ← (T)avg
11:     avgValue ← brick.ComputeWeightedAverage()
12:     if vRange ≤ threshold then
13:         This is a brick with each cell of the same value.
14:         Kill this brick (value has been saved in the parent node).
15:     else
16:         Set this brick into the brick buffer.
        return vRange
```

In addition, our tree construction approach also supports data format conversion and compression. We can easily convert an input data format to an interval format, such as double to float. Furthermore, we allow the user to set a threshold (t) that determines which input regions can be safely collapsed into a single node. For instance, consider a cell C with a child brick B. Let $Value(b)$ be a function that obtains the value at cell $b \in B$ and t be the threshold:

$$\left(\max_{b \in B} Value(b)\right) - \left(\min_{b \in B} Value(b)\right) \leq t \tag{1}$$

If Equation 1 holds, then B and its children are "collapsed" into C, meaning that C no longer points to a child brick, and is instead assigned the average value of all cells in B. Our default value of t is 0, meaning that by default, our implementation losslessly eliminates equal-value regions.

4 Volume Integration

In this section, we illustrate the rendering pipeline with Bricktrees in a simple case where all data have been read into memory. Raycasting-based volume rendering consists of marching rays through the volume with a step size and accumulating color and opacity along the ray. With a hierarchical structure, rays need to traverse the structure until reaching a leaf node, an inner node with an appropriate level-of-detail in current viewport or an unmapped node.

Algorithm 2 Pseudocode on sampling a given point p and traversal of the Bricktree structure

```
1:  function BT_CPLUS_SAMPLE(p)
2:      vArray[8] ← 0
3:      voxelCoord[8] ← Calculate 8 vertex's coordinates
4:      while i < 8 do
5:          bTreeID ← ComputeBrickTreeID(voxelCoord[i])
6:          bt ← GetBrickTree(bTreeID)
7:          vArray[i] ← Bt_Cplus_GetVoxels(bt, voxelCoord[i])
8:      result ← lerp(vArray[8])
9:      return result
10: function BT_CPLUS_GETVOXELS(bt, coord)
11:     brickID ← 0                                  ▷ top-down traversal
12:     brickStack.push(brickID)
13:     while brickStack is not empty do
14:         cBrickID ← brickStack.pop()
15:         ibID ← bt.brickInfo[cBrickID].indexBrickID
16:         cOffset ← ComputeOffset(coord)
17:         childBrickID ← bt.indexBrick[ibID].child[cOffset]
18:         if childBrickID = −1 then
19:             return bt.valueBrick[cBrickID].child[cOffset]
20:         else
21:             brickStack.push(childBrickID)
```

For a given sample point p along a ray, we need to query the value of eight neighboring voxels for interpolation. For each voxel, a tree traversal is needed to fetch the Valuebrick that the voxel belongs to. We start the Bricktree traversal from the root node. Assume that we operate on brick 1 of a $BrickTree < 4, float >$ and want to know if its cell $(1,1,2)$ has a child. First, we look up the brick's index brick ID ($ibID = indexBrickID[1]$). We know that none of the cells of brick 1 have a child if the *ibID* is invalid or less than 0. If this is the case, cell $(1,1,2)$ certainly does not have a child. However, if $ibID \geq 0$ (e.g., $ibID = 1$ in Figure 3), then we look at $cellChildID = IndexBrick[ibID].child[1][1][2]$. If this value is invalid (-1), then this particular cell of brick 1 does not have any children. Otherwise, $valueBrick[cellChildID]$ is the child brick.

Algorithm 2 describes the general sampling process. Trilinear interpolation of the eight-nodal value is used to determine the value

of an arbitrary point p. *Bt_Cplus_GetVoxels* traverses the Bricktree and queries the value of a given cell.

So far, we have obtained a sample kernel that has access to the Bricktree "forest" and is implemented with C++ code. However, as a loadable module to the OSPRay ray tracing framework, our Bricktree module employs the OSPRay rendering pipeline, which is internally built on top of Embree [41] and ISPC [33]. Embree is a collection of high-performance ray tracing kernels. The Intel SPMD Program Compiler (ISPC) allows a number of program instances to execute in parallel on SIMD hardware. ISPC compiles its own programming language (a variant of C), and this ISPC code can call and be called from the C/C++ application code. In OSPRay, the sampling process maps well to the SIMD paradigm, and is thus implemented with the ISPC code. Under this condition, we can have two approaches to implement our sampling process.

4.1 C/C++ Serial Implementation

A simple but inefficient approach is a serial implementation with the C/C++ code without maintaining a copy of the Bricktree structure on the ISPC side. We can easily implement a C/C++ version of the sample function (see Algorithm 2) that has direct access to the Bricktree structure, and call it serially from the ISPC sample callback function. The ***foreach_active*** construct is utilized to specify a loop that iterates over the active program instances serially. The loop executes once for each active program instance, and with only that program instance executing. Algorithm 3 depicts the process of serially calling the C/C++ implementation of the sampling function from the ISPC code (*programCount*, *programIndex*, *lane* are built-in variables in ISPC).

Algorithm 3 Pseudocode for serially calling the C++ version of sampling function from ISPC code.

```
1: procedure BT_ISPC_SAMPLE(varying vec3f samplePos)
2:     uniform vec3f uPos[programCount]
3:     uniform float uValue[programCount]
4:     uPos[programIndex] ← samplePos
5:     foreach_active(lane) do
6:         uValue[lane] ← Bt_Cplus_Sample(uPos[lane])
7:     return uReturnValue[programIndex]
```

4.2 ISPC Vectorized Implementation

A more efficient method involves maintaining a sibling Bricktree structure and implementing a parallel tree traversal algorithm that is run on multiple vector instruction set architectures. This method is feasible due to the ISPC code and C/C++ code being able to share the same memory. In this work, we implemented an efficient packet-based variant of *Bt_ISPC_GetVoxels* kernel, which is suitable for packet-based ray-tracing on multi-core CPU. We maintained a stack to query the Valuebrick for an entire packet of ray samples. Assuming eight inputs in a packet, the ideal case is all the inputs will traverse to the same Valuebrick, in which case we can achieve a theoretical 8x performance improvement. In practice, not all samples will traverse to the same level, and performance is influenced by Valuebrick size. We achieved a 2x performance improvement for $N = 4$.

4.3 64-bit Addressing

For performance reasons, ISPC uses 32-bit addressing by default, even with a 64-bit compilation target. In this addressing mode, ISPC maps all addresses for **varying** array access to vectors of 32-bit offsets relative to a shared 64-bit base pointer. However, this approach limits the volume size to 4 GB. When our data exceed 4 GB, 32-bit addressing will fail in Algorithm 2, line 19. Hence, we need to treat each array as consisting of smaller segments, and then iterate over all unique segments addressed by a vector using ISPC's ***foreach_unique*** statement. To do so, we can guarantee that each segment can be addressed with a 32-bit offset. More details about the implementation can be found in [37].

With this technique, our implementation can scale to the workstation's available memory. However, the implementation has some performance penalties, because 64-bit addressing is slightly more costly than 32-bit addressing. In an ideal case, memory accesses are mostly coherent, and only a few iterations of the aforementioned loop are necessary. In the worst case, many iterations are needed. However, the performance is still superior to the performance when running our application in the ISPC 64-bit addressing mode.

4.4 Sampling Optimization

Although we implement an efficient traversal kernel for packet sampling along the ray, eight neighboring voxels need to be accessed to perform trilinear interpolation. A naive way to access these voxels is to call the traversal kernel eight times. Given that the neighboring voxels are likely located in the same Valuebrick, duplicated traversal can be avoided if we initialize neighboring voxels in the same Valuebrick.

Ideally, only one traversal of the Bricktree rather than eight is needed for a given sample point. In practice, the speed-up is lower than 8x, since we need to retraverse the tree for the neighboring voxels that are located in neighboring Valuebricks. For large datasets, data overhead with this approach will be high when a small brick size is used. Benefiting from the large branching factor of our data structure, the generated Bricktrees will keep a small tree depth. Hence, the retraversal at the brick boundary will only slightly impact the performance. One would expect to achieve a greater speed-up with a larger brick size, but larger brick sizes incur more possibility that eight voxels are located in the same brick but with inefficient empty space skipping [13]. An analysis of the choice of brick size is conducted in Section 6.2. By performing the optimization mentioned above, we achieve a 4-5x performance improvement for $N = 4$.

5 Ray-Guided Progressive Rendering

Our Bricktree naturally leads to progressive rendering where data streaming and rendering are asynchronous in the visualization pipeline. If the requested data are not yet in memory, the renderer uses the average value stored in the (already loaded) parent node and requests the missing brick from a data loading thread. Consequently, the visualization process can be launched very quickly without waiting for the large data to first be loaded into the memory. In addition, rendering performance remains stable when the camera position is updated.

5.1 Valuebrick Streaming

So far, we have illustrated how we sample and traverse the Bricktree assuming that all multi-resolution bricks are in memory, which works well on small- or moderate-size datasets. For large datasets, however, IO bottlenecks and memory size limit performance and rendering quality. In particular, IO latency for loading the dataset into memory before rendering becomes prohibitive when visualizing large data. Loading a terascale dataset into memory on a contemporary workstation might take hours, even with a parallel IO library such as PIDX. Furthermore, the memory might not be adequate to load all bricks.

To solve this problem, we employ ray-guided progressive rendering and stream bricks on demand. Although the idea behind this approach is similar to GigaVoxel [8] and to Hadwiger's work [17], our implementation on multi-core CPUs is more straightforward and avoids the complex data structures needed to store the rendering context. As mentioned in Section 3.1, each Bricktree stores a linear array of Valuebricks. All we need to do is maintain a status for each Valuebrick and update it correctly according to the rendering context. In our solution, two additional bits (*isRequested* and *isLoaded*)

Algorithm 4 Sampling function with progressive rendering on top of our Bricktree structure

```
1:  procedure BT_ISPC_GETVOXELS(bt, coord)
2:      cBrickID ← 0                          ▷ current brick, set to root
3:      pBrickID ← −1                         ▷ parent brick
4:      uniform FindStack stack[16]           ▷ ISPC stack struct
5:      stackPtr ← pushStack(&stack[0], cBrickID, pBrickID)
6:      while stackPtr > stack do
7:          −− stackPtr
8:          if stackPtr is active then
9:              cBrickID ← stackPtr.cBrickID
10:             pBrickID ← stackPtr.pBrickID
11:             if current brick is not requested then
12:                 bt.brickStatus[cBrickID].isRequested ← true
13:             if bt.brickStatus[cBrickID].isLoaded then
14:                 childBrickID ← getChildBrickID()
15:                 cOffset ← ComputeOffset(coord)
16:                 if childBrickID = −1 then
17:                     return value in current brick
18:                 else‘
19:                     pushStack(stackPtr, childBrickID, cBrickID)
20:             else
21:                 return average value in the parent brick
```

are stored for each brick and used to identify the brick's current status. In our sample function, *isRequested* is set when we need to load the Valuebrick into memory, which allows us to implement a visibility-based solution that requests only data that are inside the view frustum.

5.1.1 Progressive Sampling

The sample function needs to be able to handle the equivalent of a cache miss – the case when a Valuebrick is needed but not in the memory – to generate a correct image. One simple and synchronous approach mentioned in [8] is to load the missing Valuebrick immediately and for volume integration to stop until the Valuebrick is loaded. This approach is problematic for interactive visualization when many Valuebricks are missing due to the prohibitive IO latency incurred by Valuebrick loads. Given that the inner node in the Bricktree stores the average values of its child nodes, we adopted an asynchronous approach for volume integration. We use the average value as an approximation for the current sample point when we request the Valuebrick and refine the output image when the Valuebrick is loaded in separate threads. Algorithm 4 depicts the pseudo-code of this process.

5.1.2 Valuebrick Loading Strategy

In this work, multi-threading is employed to decouple the data streaming and rendering. We create independent threads that are responsible for determining the streaming sequence and loading the requested Valuebrick from the file system. These threads are able to access the Valuebrick buffer and update the *isloaded* status when a Valuebrick is loaded. An interactive and smooth visualization is achieved in coordination with the progressive rendering.

Although we achieve a stable framerate by decoupling the data streaming and rendering, the Valuebrick loading sequence impacts the qualitative performance of our progressive refinement strategy. Given the layout of our Bricktree, three choices are considered.

Recall from Section 3.1 that the volume is encoded as a Bricktree Forest. Hence, the naive approach is to map the whole Bricktree into memory if a Valuebrick in the respective Bricktree is requested. We refer to this approach as "streaming by Bricktree". This approach is easy to implement but not efficient, because Bricktrees tend to be large, and because many unrequested Valuebricks will be loaded.

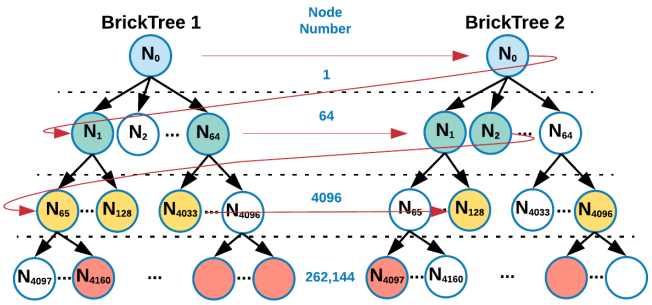
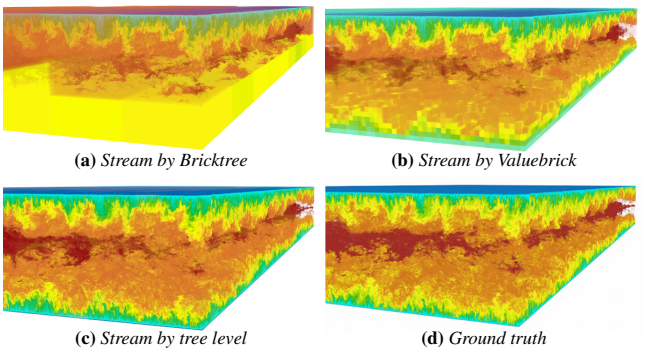


Figure 4: *An illustration of streaming the Valuebrick by level. Colored nodes are requested in current time. The red arrow indicates the loading sequence.*

A finer-granularity approach is to "stream by Valuebrick" rather than by Bricktree. For each Bricktree, the streaming threads will filter and load the requested Valuebricks. This approach yields better performance than the first approach, since loading a Valuebrick is much cheaper than loading a Bricktree. Visually, the user will notice a smoother and finer grained refinement (as shown in Figure 5b). However, for the first few frames, when a large number of Valuebricks are missing, performance is lackluster, since many Valuebricks will be requested at once by the ray sample function. In the worst case, we need to load almost all Valuebricks in the current Bricktree before moving to the next Bricktree, which results in a slight performance improvement over the first approach.

Generally, the data exploration process is "overview first, zoom and filter, then detail-on-demand" [11, 42]. Following this mantra, the ideal approach is to update the data resolution progressively. In this work, we adopt an approach similar to level-order tree traversal and load the requested Valuebricks by level. Figure 4 illustrates this approach. Given a user-defined view frustum, the streaming thread selects the requested Valuebricks (colored) by level and pushes them into a queue to load. Due to the large branching factor (N^3) of our hierarchical structure, only a small portion of the nodes are inner nodes. For example, in Figure 4, using 4^3 bricks to encode a 512^3 volume, only 1.6% of the nodes are inner nodes. Consequently, loading the inner node levels takes relatively little time, and we can achieve smooth visual refinement of our data. Figure 5 compares visual refinement using different streaming strategies.

Another factor that improves progressive refinement performance is Valuebrick load speed, although this is highly dependent on IO throughput. Due to the design of the Bricktree "forest", the levels of the Bricktree can be loaded in parallel, since each process is



(a) *Stream by Bricktree* **(b)** *Stream by Valuebrick*

(c) *Stream by tree level* **(d)** *Ground truth*

Figure 5: *A comparison of rendering images of the DNS dataset with three Valuebrick loading strategies at frame 100. Stream by level shows more detail and smoother data refinement.*

independent. For example, we can load level 2 of tree 1 and tree 2 in parallel using $tbb :: parallel\_for$. Compared to serial streaming, we observe that parallel streaming significantly improves performance.

5.2 Early Tree Traversal Termination

So far, we have discussed a ray-guided visibility culling approach that allows our renderer to load only the visible portion of the volume. The idea is, for a given viewport, that only part of the data contributes to the final image. Similar ideas can be applied to improve rendering performance. In particular, under the right conditions, Bricktree traversal can be stopped at an inner node that reaches the appropriate level of detail (LOD).

5.2.1 Level-of-Detail Control

Some primitives are smaller than the output device pixels when rendering a large dataset at low magnification [35]. Generally, in this case, it is hard to tell the visual difference in the interactive rendering if we use higher resolution data. One general-purpose approach to reduce loading and rendering time is to store data at a discrete level of detail and select a lower resolution LOD with primitives that more closely matches the display resolution.

In this work, it is easy to apply LOD-based ideas, since an inner node can be interpreted as a coarser representation of its descendants. During traversal, we calculate the projected screen space area of a Valuebrick and stop the traversal if the area is smaller than a user-defined threshold. To simplify computation, we use a sphere as an approximation of a cubic brick. By setting the threshold to one pixel, we achieve a 2.5x speed-up when we zoom out and view the DNS dataset at low magnification.

5.2.2 Culling with Transfer Function

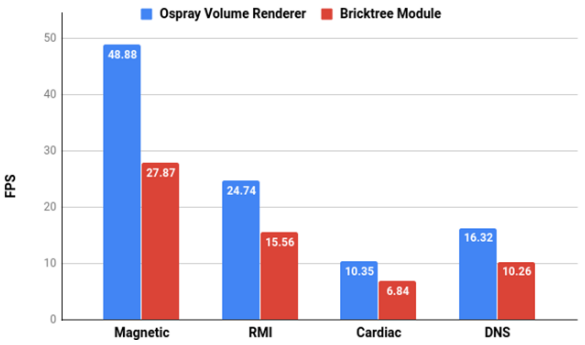
We can stop the traversal when the Valuebrick does not contribute to the final image given the current transfer function. In general, the performance of interactive visualization depends on the user-defined transfer function. For example, the framerate drops to 2-3 fps if we make the interior of the DNS volume transparent and keep the top and bottom boundaries opaque, because rays terminate much later when most of the volume is transparent. Furthermore, tree traversal is still performed on each sample even when its corresponding Valuebrick is transparent. Taking advantage of the value range stored in each node, this can easily be avoided in our implementation. The callback function *getMaxOpacityInRange(cellRange)* allows us to look up the brick's maximum opacity based on the current transfer function. If the maximum opacity equals 0, the tree traversal stops at the current Valuebrick and skips the descendants. Although the speed-up of this optimization depends on the data and transfer function, we achieve an average 2x speed-up for the DNS dataset with a transparent interior.

6 Results

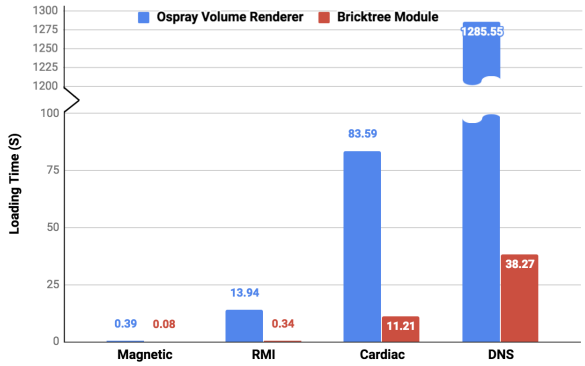
In this section, we evaluate four key aspects of our system: 1) the performance of the existing OSPRay volume renderer and our Bricktree module; 2) the performance of the Bricktree with different brick sizes; 3) data compression using the Bricktree; 4) the performance of Bricktree generation. Unless otherwise noted, benchmarks were performed on a quad-socket workstation (FSM) with four Xeon E7-8890 v3 CPUs, with a total of 72 physical cores at 2.5 Ghz, along with 3 TB RAM. All test datasets are stored on a network-mounted file server over a 1Gb/s network. The maximum I/O throughput of the file system is 110 MB/s. In the benchmark, volumes were rendered to a 1024×768 framebuffer, and the rendering performance was measured by calculating the average framerate over 100 frames. Bricktree read and write bandwidths include effects from the filesystem cache, which may result in a measured bandwidth value that appears to exceed the underlying hardware's capability. Nonetheless, the measured read/write times are the ones that would be observed by a real user.

We tested our renderer with four datasets with sizes ranging from small (e.g., magnetic reconnection volume [15]), to medium (e.g., the Richtmyer-Meshkov instability simulation [6] and cardiac volume [22]) to extremely large (e.g., the DNS dataset [31]).

- The magnetic volume, produced from kinetic simulations, is a 512^3 float-precision dataset (size: 512 MB) that represents magnetic reconnection in relativistic plasmas.
- The Richtmyer-Meshkov instability (RMI) is a simulation of carbon hexafluoride being pushed up through a wire mesh, with a resolution of $2048 \times 2048 \times 1920$ and data type of uint8 (size: 8.1 GB). It is a popular dataset and is also used in [13, 24, 45].
- The cardiac volumes were obtained by way of computed tomography (CT) imaging on excised, postmortem porcine hearts. The resolution of the data is $2048 \times 2048 \times 2612$. With a data type of int16, the size of the data goes up to 21 GB.
- The DNS, produced by the Institute for Computational Engineering and Science (ICES) at the University of Texas-Austin, is a single $10240 \times 7680 \times 1536$ double precision volume (size: 900 GB) from a turbulent flow simulation. In our experiments, we use a float version of the DNS data (450 GB).



(a) *Overall performance*



(b) *Loading time before rendering the first frame*

Figure 6: *A comparison of overall performance and loading time between existing OSPRay volume renderer and our Bricktree module. The evaluation was run on Bricktree with a brick size of 4.*

6.1 Existing OSPRay Renderer vs. Bricktree Module

In this section, we first compare the overall performance and loading time before rendering the first frame between our Bricktree module and the existing OSPRay volume renderer. It is a trade-off. For our Bricktree module, every individual sample requires traversing a

hierarchical data structure, whereas the existing OSPRay volume renderer uses a standard structured volume and allows data to be queried using simple math operations. We would expect each sample to be several times as costly as for the reference renderer. However, given our access-friendly data layout and ISPC-optimized tree traversal, we see in Figure 6 that the rendering performance for fully loaded data is only about 35% slower. Despite some performance loss, Figure 6a indicates that our Bricktree module still runs at interactive framerates. The benchmarks are run from a fixed viewpoint with the volume covers through 3/4 of the screen and with a semi-transparent transfer function. Images of the benchmarks are provided in the supplementary material.

More importantly, our Bricktree module performs much better than the existing OSPRay renderer when it comes to loading time. We measured the time that a user must wait before the renderer draws the first frame (see in Figure 6b). The Bricktree module achieves much better performance, since it needs to load only metadata prior to rendering. With both renderers, loading time increases with data size. However, our Bricktree module slows down much less. For instance, without the Bricktree module, users must wait more than 20 minutes on the test workstation (FSM) before being able to see a "bird's eye view" of the DNS dataset. By contrast, with our Bricktree module, we get an overview of the DNS dataset in under 40 seconds.

6.2 Choice of Brick Size

A volume renderer's domain decomposition method can have a large impact on the performance of the rendering pipeline [13]. Our design and implementation allow the user to easily set a custom brick size. We performed a set of experiments to discover how the brick size affects the tree generation time, tree size, loading time, streaming performance and overall framerate. Two datasets were tested: RMI (8.1 GB, int8), which is a sparse dataset, and DNS(450 GB, float), which is a very dense dataset.

RMI / Brick Size	**2**	**4**	**8**	**16**
Tree Build (min)	1.61	1.49	**1.34**	1.96
Tree Size (GB)	7.80	**5.90**	7.10	9.60
Loading Time (s)	1.48	0.34	**0.15**	0.56
Stream Performance (s)	0.21	0.13	**0.11**	0.08
Framerate (fps)	10.56	**18.56**	10.75	7.78
DNS / Brick Size	**2**	**4**	**8**	**16**
Tree Build (min)	177.22	92.93	86.66	**80.25**
Tree Size (GB)	772	486	455	**451**
Loading Time (s)	92.26	38.27	9.3	**7.67**
Stream Performance (s)	0.18	0.16	0.13	**0.08**
Framerate (fps)	15.36	18.38	36.63	**45.62**

Table 1: *An evaluation of tree build time, tree size, loading time, stream performance (s/1000 Valuebricks) and overall performance with different brick sizes on a sparse dataset (RMI) and a dense dataset (DNS). The evaluataion was run on FSM. The tree build process was executed in parallel with eight processors.*

Table 1 shows rendering performance with different brick sizes. As we know, the smaller Valuebrick size is more likely to be uniform in value [13]. In our unbalanced Bricktree, a Valuebrick that contains uniform values will not be further decomposed and stored by adopting a lossless fashion (mentioned in Section 3.3) and setting the threshold to the default value (0). Under this consideration, domain decomposition with a small brick size is more likely to yield a Bricktree with fewer Valuebricks. Therefore, a small brick size is more suitable for storage and probably fast traversal.

However, the results in Table 1 indicate a slightly different conclusion for different datasets. For a sparse dataset (e.g., RMI), a brick size of 4 produces a smaller tree and performs better than a brick size of 8 or 16. A brick size of 2 hampers performance and tree size, possibly due to an increased number of inner nodes resulting from greater tree depth.

For a dense dataset (e.g., DNS), where values vary in almost all cells, the output Bricktree is most likely a balanced tree. Hence, we found that a smaller brick size results in a larger tree size and longer construction time. From the perspective of streaming performance, a large brick size is preferable for disk performance. For instance, the size of a Valuebrick with a brick size 16 is 4 KB, which equals the size of a single memory page on FSM. Therefore, a brick size of 16 gives more a friendly memory access pattern. We observed that the streaming thread will influence rendering performance when many Valuebricks are requested. For performance reasons, we believe that a large Valuebrick size is more appropriate for visualizing dense datasets. Table 1 demonstrates that, for the DNS dataset, the best rendering performance is achieved with a brick size of 16.

Recall that a Bricktree is an octree when the brick size N is set to 2. As shown in Table 1, a Bricktree with an appropriate brick size outperforms an octree in terms of framerate, tree size, tree construction time and loading time.

6.3 Compression Results

In Section 3.3, we described how a threshold could be specified in order to collapse a specific input region of volume during the hierarchy generation. By using the default value 0, we produced a lossless hierarchical representation of the volume. Section 6.2 indicates that this lossless compression is satisfactory for a sparse dataset, since regions with uniform values are collapsed. However, a lossy representation is generated if the threshold is set to a positive value. Figure 7 displays Bricktree build time, the ratio of the Bricktree's size to raw volume and the rendering performance given different thresholds over the DNS dataset. The build time and tree size drop dramatically when the threshold is increased, and the rendering performance improves significantly. A detailed quantitative analysis of the accuracy of the output image over different thresholds is beyond the scope of this paper. However, Figure 8 depicts the output image of the magnetic dataset rendered with four thresholds. Aside from the performance improvement, we observe that only a slight difference can be discerned in the final images if we select an appropriate threshed (e.g., 0.05) for the magnetic dataset. On the other hand, artifacts emerge when the data are visualized with a large threshold (e.g., 0.3).

6.4 Bricktree Generation

Although hierarchy generation performance has drawn less attention than rendering performance, it is a significant bottleneck in real-world usage [13]. In this section, we evaluate the performance of

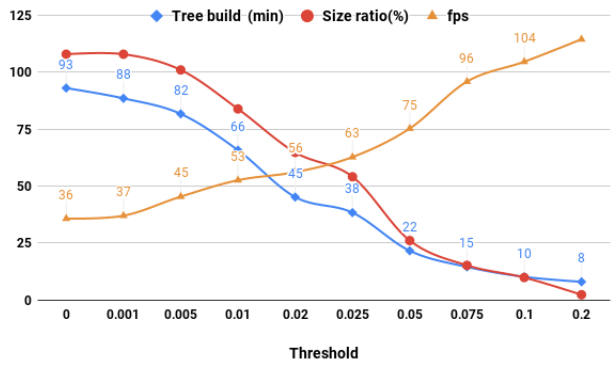


Figure 7: *A comparison of tree build time, size ratio (tree size / volume size) and overall performance with different thresholds on the DNS dataset.*

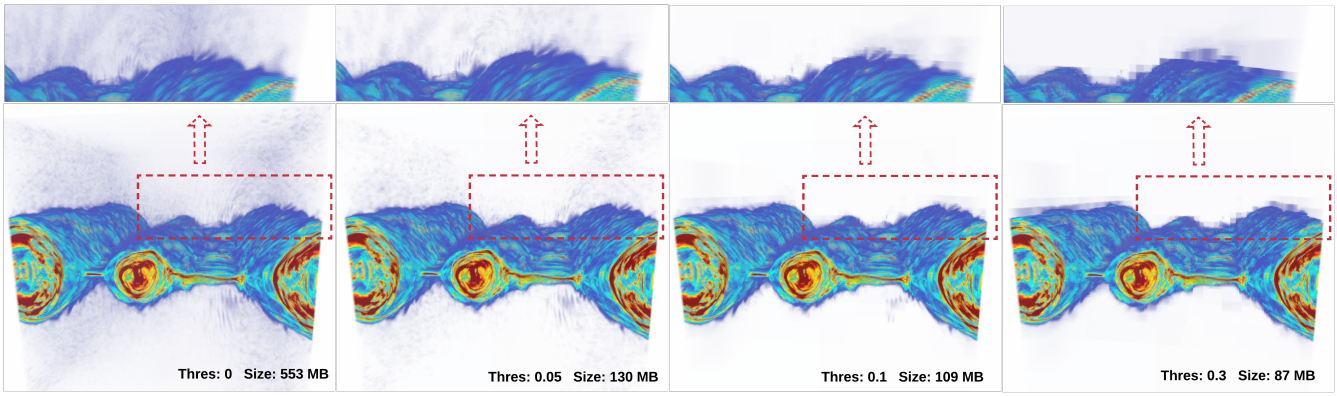
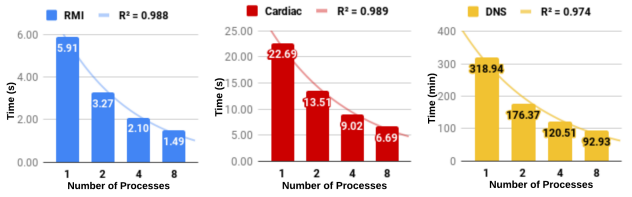


Figure 8: *A comparison of the output image rendered with four thresholds on the magnetic dataset (512 MB). With an appropriate threshold, such as 0.05, we achieve significant performance improvement and produce a final image that is slightly different from ground truth (thres: 0).*



(a) *RMI* **(b)** *Cardiac* **(c)** *DNS*

Figure 9: *Tree build time by running the ospRawToBrick tool with different numbers of processes on three datasets. The brick size of the Bricktree is set to 4 in this evaluation.*

the "*ospRawToBrick*" tool. Tree generation time, which benefits from the parallel build processes, drops dramatically by running multiple jobs simultaneously. Figure 9 shows hierarchy generation time with three datasets. For the RMI dataset, [13] claims that the generation time of their hierarchy structure takes about 13 minutes in the best case. However, with our solution, it only takes 1.49 seconds using eight processes. For the DNS datasets, we reduce the offline construction time from 5 hours with one processor to 1.5 hours with eight processors.

7 Conclusion

In this paper, we have presented a solution for interactive visualization of large volumes on multi-core CPU architectures. Our method is based on hierarchical representation and ray-guided progressive rendering that allows the user to view and explore hundreds of gigabytes of data without spending minutes or even hours waiting for data to load. Our solution is a significant improvement compared to the existing OSPRay volume renderer, which usually takes minutes or hours to load the data prior to rendering the first frame. Inspired by many recent renderers, we present a hierarchical data structure – a Bricktree – with a large branching factor and relatively low overhead. We have also evaluated and discussed the tradeoffs of the different parameters. Based on our experimental results, we conclude that sparse datasets are best used with small brick sizes (e.g., 4), and dense datasets are best used with large brick sizes (e.g., 16).

Our data structure naturally facilitates compression. Using the Bricktree to create an easy-to-implement lossless compression scheme, we reduced the size of the RMI dataset from 8.1 GB to 5.9 GB with a brick size of 4 (Table 1). Furthermore, this scheme can easily be extended to support lossy compression.

Finally, we implemented our solution as a module for the OSPRay ray tracing framework. Since OSPRay is already interfaced with tools like Paraview and VisIt, users should have no difficulty putting our results into production.

In the future, we hope to further investigate data compression. For instance, Valuebricks are natural candidates for float-point data compression using ZFP [30]. In addition, a detailed quantitative analysis of how the lossy compression threshold affects the final image quality should be performed. On the other hand, a hierarchical structure naturally leads to adaptive sampling. It would be interesting to further apply the adaptive sampling to our Bricktree module. Eventually, we would like to integrate our Bricktree into general-purpose visualization frameworks. We are also interested in testing the Bricktree structure in GPUs architecture.

Acknowledgments

This project was supported in part by the Intel Graphics and Visualization Institutes of XeLLENCE and the National Institute of General Medical Sciences of the National Institutes of Health under grant number P41 GM103545-18. The authors wish to thank Paul Navrátil from TACC for the use of DNS datasets and the Lawrence Livermore National Laboratory for the use of Richtmyer-Meshkov datasets. We would also like to thank Nathan Marshak for reviewing and polishing the manuscript.

References

[1] J. Beyer, M. Hadwiger, and H. Pfister. A Survey of GPU-Based Large-Scale Volume Visualization. In *EuroVis - STARs*. The Eurographics Association, 2014. doi: 10.2312/eurovisstar.20141175

[2] J. Beyer, M. Hadwiger, J. Schneider, W.-K. Jeong, and H. Pfister. Distributed terascale volume visualization using distributed shared virtual memory. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pp. 127–128. IEEE, 2011.

[3] H. Childs, M. A. Duchaineau, and K.-L. Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *EGPGV*, pp. 153–161, 2006.

[4] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, G. H. Weber, E. W. Bethel, et al. Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications*, 30(3):22–31, 2010.

[5] J. Clyne, P. Mininni, A. Norton, and M. Rast. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics*, 9(8):301, 2007.

[6] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward. Three-dimensional simulation of a richtmyer–meshkov instability with a two-scale initial perturbation. *Physics of Fluids*, 14(10):3692–3709, 2002.

[7] C. Crassin, F. Neyret, and S. Lefebvre. *Interactive gigavoxels*. PhD thesis, INRIA, 2007.

[8] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 15–22. ACM, 2009.

[9] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE transactions on visualization and computer graphics*, 15(3):436–452, 2009.

[10] K. Engel. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pp. 123–124. IEEE, 2011.

[11] B. Ferster. *Interactive visualization: Insight through inquiry*. MIT Press, 2012.

[12] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher. Large data visualization on distributed memory multi-GPU clusters. In *Proceedings of the Conference on High Performance Graphics*, pp. 57–66. Eurographics Association, 2010.

[13] T. Fogal, A. Schiewe, and J. Krüger. An analysis of scalable GPU-based ray-guided volume rendering. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pp. 43–51. IEEE, 2013.

[14] E. Gobbetti, F. Marton, and J. A. I. Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008.

[15] F. Guo, H. Li, W. Daughton, and Y.-H. Liu. Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Physical Review Letters*, 113(15):155005, 2014.

[16] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE transactions on visualization and computer graphics*, 24(1):974–983, 2018.

[17] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, 2012.

[18] M. Hadwiger, F. Laura, C. Rezk-Salama, T. Höllt, G. Geier, and T. Pabel. Interactive volume exploration for feature detection and quantification in industrial ct data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1507–1514, 2008.

[19] M. Han, I. Wald, W. Usher, Q. Wu, F. Wang, V. Pascucci, C. D. Hansen, and C. R. Johnson. Ray tracing generalized tube primitives: Method and applications. In *Computer Graphics Forum*, vol. 38, pp. 467–478. Wiley Online Library, 2019.

[20] M. Howison, E. W. Bethel, and H. Childs. Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):17–29, 2012.

[21] D. Keim, H. Qu, and K.-L. Ma. Big-data visualization. *IEEE Computer Graphics and Applications*, 33(4):20–21, 2013.

[22] P. Klacansky. Open scientific visualization datasets, August 2018.

[23] A. Knoll, S. Thelen, I. Wald, C. D. Hansen, H. Hagen, and M. E. Papka. Full-resolution interactive CPU volume rendering with coherent BVH traversal. In *Visualization Symposium (PacificVis), 2011 IEEE Pacific*, pp. 3–10. IEEE, 2011.

[24] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pp. 115–124. IEEE, 2006.

[25] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 38. IEEE Computer Society, 2003.

[26] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout. PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets. In *2011 IEEE International Conference on Cluster Computing*, pp. 103–111. IEEE, 2011.

[27] S. Lefebvre, S. Hornus, F. Neyret, et al. Octree Textures on the GPU. *GPU gems*, 2:595–613, 2005.

[28] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics (TOG)*, 9(3):245–261, 1990.

[29] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 42. IEEE Computer Society, 2003.

[30] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[31] R. D. Moser, J. Kim, and N. N. Mansour. Direct numerical simulation of turbulent channel flow up to re τ= 590. *Physics of fluids*, 11(4):943–945, 1999.

[32] T. Peterka, H. Yu, R. Ross, and K.-L. Ma. Parallel Volume Rendering on the IBM Blue Gene/P. In *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2008. doi: 10.2312/EGPGV/EGPGV08/073-080

[33] M. Pharr and W. R. Mark. ISPC: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pp. 1–13. IEEE, 2012.

[34] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*, pp. 46–65. Springer, 2003.

[35] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[36] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. M. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, et al. Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. *IEEE transactions on visualization and computer graphics*, 15(6), 2009.

[37] I. Wald. ISPC Bag of Tricks Part 3: Addressing 64-bit arrays in 32-bit addressing mode, Nov 2018.

[38] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005.

[39] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. OSPRay-a CPU ray tracing framework for scientific visualization. *IEEE transactions on visualization and computer graphics*, 23(1):931–940, 2017.

[40] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer graphics forum*, vol. 20, pp. 153–165. Wiley Online Library, 2001.

[41] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4):143, 2014.

[42] F. Wang, W. Li, S. Wang, and C. Johnson. Association rules-based multivariate analysis and visualization of spatiotemporal climate data. *ISPRS International Journal of Geo-Information*, 7(7):266, 2018.

[43] F. Wang, I. Wald, Q. Wu, W. Usher, and C. R. Johnson. Cpu isosurface ray tracing of adaptive mesh refinement data. *IEEE transactions on visualization and computer graphics*, 25(1):1142–1151, 2019.

[44] F. Wang, G. Wang, D. Pan, Y. Liu, Y. Liuzhong, and H. Wang. A parallel algorithm for viewshed analysis in three-dimensional digital earth. *Computers and Geosciences*, (75):57–65, 2015.

[45] Q. Wu, W. Usher, S. Petruzza, S. Kumar, F. Wang, I. Wald, V. Pascucci, and C. D. Hansen. VisIt-OSPRay: Toward an Exascale Volume Visualization System. In *EGPGV*, pp. 13–23, 2018.

Supplementary Material 1