Distributed Terascale Volume Visualization Using Distributed Shared Virtual Memory

Johanna Beyer* KAUST
Markus Hadwiger KAUST
Jens Schneider KAUST
Won-Ki Jeong Harvard University
Hanspeter Pfister Harvard University

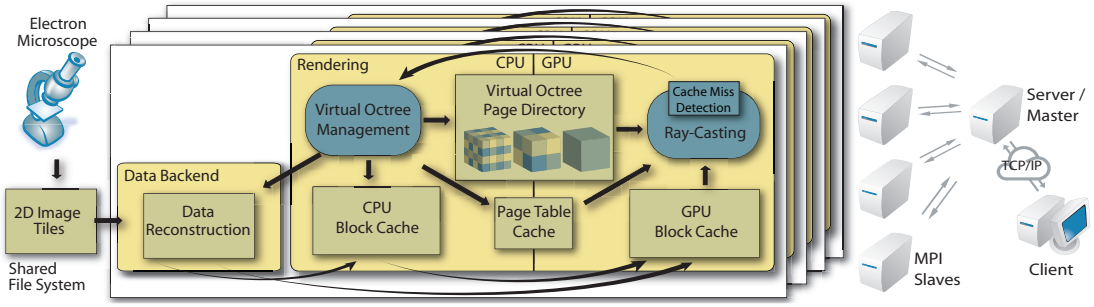

Figure 1: Our terascale volume rendering system builds on the concept of *distributed shared virtual memory*, which facilitates the distribution of the rendering process to multiple nodes and GPUs. Each node refers to the same shared 3D virtual memory space that represents the entire volume, which can be accessed independently on each GPU. Ray-casting on each GPU independently determines small 3D blocks that are actually accessed, which correspond to memory pages. Virtual memory is managed via a two-level hierarchy of page tables, and physical memory is only mapped for the current working set of pages. Memory management and ray-casting operate independently on each node.

1 Introduction

Recent advances in data acquisition techniques such as high-resolution electron microscopy (EM) result in volume data of extremely large size. For example, in neuroscience current EM data sets of brain tissue have pixel resolutions of 3-5nm, resulting in volumes of hundreds of terabytes, rapidly approaching the petascale [3]. Volume rendering such data presents new challenges and usage patterns. EM data is extremely dense and neurobiologists tend to explore the volume at close range most of the time, while looking at the entire volume only in the beginning. This usually results in rather uniformly filled view frustums, in which nevertheless only a small percentage of the whole volume is visible. Additionally, the resolutions of EM data are so high that multi-resolution approaches are necessary to avoid aliasing due to undersampling in zoomed-out views, and to bound the maximum working set size.

Our volume rendering system employs *distributed shared virtual memory*, which comprises a single virtual memory space that is shared by all rendering nodes [4]. We combine this with a sort-last (object-space decomposition) approach for volume rendering on a GPU cluster. We employ GPU-based ray-casting on each node, but in contrast to single-GPU out-of-core ray-casters [1], our system performs efficient multi-GPU volume ray-casting that allows to leverage much more overall physical cache memory. On each node, we allow the ray-caster to sample the shared virtual volume space at arbitrary positions. Only the memory pages that are actually accessed are mapped to physical memory on the respective node. At a higher conceptual level, object space decomposition is done in large *virtual distribution units*, each of which comprises many much smaller virtual memory pages. We refer to these pages also as *blocks*, which are small 3D sub-volumes of 32^3 voxels that allow each node to perform efficient culling, streaming, and memory management. In contrast to many existing systems for distributed volume rendering [2], our system targets GPU clusters with only a few nodes and does not require an explicit load balancing scheme [5]. Our main motivation for using a cluster is allowing a larger total working set size, and implicitly balancing the data load between nodes, as opposed to mainly targeting rendering performance. Overall, a distributed shared virtual memory space greatly simplifies the distribution of rendering and volume data.

*e-mail:johanna.beyer@kaust.edu.sa

IEEE Symposium on Large Data Analysis and Visualization
October 23 - 24, Providence, Rhode Island, USA
978-1-4673-0155-8/11/$26.00 ©2011 IEEE

2 Virtual Memory-Based GPU Volume Rendering

The overall structure of our system is illustrated in Figure 1. Each cluster node is running its own pipeline for ray-casting, virtual volume and memory management, and paging. Ray-casting is performed on a huge *virtual 3D volume*, which is conceptually given on a regular grid in 3D. However, this volume is virtual because the data that are sampled by the ray-caster are only created on demand (i.e., read from disk or reconstructed on-the-fly), as determined by the actual visibility of small cubical *blocks* or memory pages of 32^3 voxels each. In order to be able to adapt the data resolution used for ray-casting to the output screen resolution, we represent the whole virtual volume as a *virtual octree* multi-resolution hierarchy, which is mapped to a large *shared virtual memory* address space. Each octree node is solely comprised of the block of 32^3 voxels that it represents, which conceptually resides in virtual memory. This small block size allows for a fine granularity of visibility detection and culling, as well as fine-grained paging of blocks into physical memory with low latency. Ray-casting traverses the volume in virtual memory, and performs on-the-fly address translation from virtual to physical memory addresses for each sample. The ray-caster computes a level-of-detail (LOD) value for each sample individually, which determines the octree level and thus resolution to be sampled. The ray-caster detects, in front-to-back order, *cache misses* of virtual octree nodes whose block of voxel data is not resident in the *GPU block cache*. If a cache miss also cannot be satisfied from the larger *CPU block cache*, the physical data of matching resolution are read in the background, while rendering proceeds at interactive rates in the foreground. Determining the octree nodes in this display-aware fashion significantly bounds the actual cache working set required for volume rendering, because even high display resolutions are much smaller than our data sets.

For distributed rendering on multiple GPU nodes, the whole rendering and memory management pipeline is duplicated on each

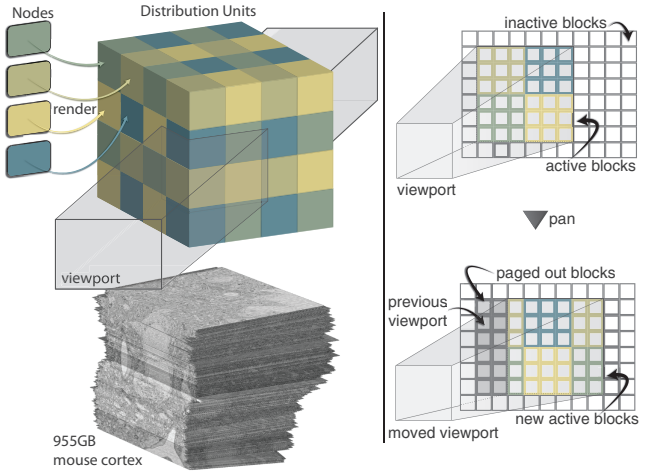

Figure 2: Spatial arrangement of *virtual distribution units*. Units depicted in the same color are assigned to the same GPU node. This spatial assignment, together with further subdivision into blocks of 32^3 voxels, achieves a roughly equal maximum working set size for each node with respect to visible blocks inside the view frustum.

GPU node, where each node references the same shared virtual memory space and operates largely independent of the other nodes, leading to a very small amount of communication between nodes.

For distributed rendering, we employ a sort-last scheme with direct-send compositing. However, we do not explicitly split up the volume data into bricks assigned to each render node. Instead, each node is simply assigned responsibility for different cubical regions in the global virtual memory space. We call one such region a *virtual distribution unit* and specify the size of this unit such that it corresponds to a relatively large cubical region of voxels, such as 1024^3. This fixed set of virtual distribution units for each node is arranged in an interleaved, repeating spatial layout (see Figure 2). For each frame, only the distribution units intersecting the view frustum are set to active, and thus need to be rendered and composited.

The main motivation behind this approach is that while roaming the volume, old distribution units leave the current view frustum, and new distribution units enter it, in a roughly equally-distributed fashion. Note, however, that this approach is only feasible because the distribution units are further subdivided into small blocks of 32^3 voxels that can be paged in and out of cache texture memory individually with fine granularity. The size of the distribution units is chosen small enough to ensure data load balancing when roaming the volume and to fit comfortably into GPU memory, and at the same time large enough to avoid splitting the volume into too many different parts that need to be composited later on. Due to our shared virtual memory approach the assignment of virtual distribution units to GPU nodes and subsequent rendering is straightforward. Before rendering a frame, each node culls the distribution units in this list given their geometric position, and the current view frustum, and sorts the units into a front-to-back order. Next, the node loops over each of these distribution units indivudially, and renders, reads back and directly sends out the images for parallel compositing. Furthermore, we use a distribution unit's alpha image to employ occlusion culling for the remaining distribution units. For this we send out a low-resolution occlusion map of the rendered distribution unit to all other nodes, which blend it with their local occlusion information. This leads to skipping of occluded distribution units, and significantly reduces the communication overhead between nodes and overall render time.

For very large volumes, the geometric size of the virtual distribution units with respect to the normalized volume bounding box cannot be kept constant without increasing the number of distribution units inside the field of view significantly when zooming away from the volume. A high number of distribution units, however, would result in too many ray-casting passes and a significant compositing overhead. Therefore, we switch the size of virtual distribution units according to the overall view zoom factor at which the volume is currently viewed. Switching the size of distribution units is synchronized between cluster nodes, and depends on the current screen resolution, volume size and level-of-detail settings. This, however, is straightforward because the necessary calculations can be performed independently on each node since the same virtual volume space is shared by all nodes. For seamless switching of distribution unit size, prefetching of volume data corresponding to the new size is necessary in order to avoid delays in rendering on the individual nodes. For this, we employ a synchronization flag that signals to all nodes that prefetching has completed, and all nodes switch in sync.

# GPUs	2 GPUs (= 4 GB cache)			4 GPUs (= 8 GB cache)		
screen res.	512^2	768x512	1Kx768	512^2	768x512	1Kx768
dist. unit size	256^3	512^3	1024^3	256^3	512^3	1024^3
ray-cast (ms)	37.0	34.4	58.8	21.0	54.9	60.9
total frame (ms)	95.2	117.6	208.3	67.5	142.8	188.6
# dist. unit	<=10	<=4	<=4	<=5	<=2	<=2

Table 1: Results for the 955GB dataset for different numbers of GPUs, screen resolutions, and distribution unit sizes. We give times in ms for ray-casting and for the total of all stages (ray-casting, image read-back and send, compositing, client transmission). We also give the number of distr. units in view that are actually rendered per node.

The communication between the different render nodes is done with asynchronous MPI messages. Rendering and sending out image tiles and occlusion maps is decoupled from receiving images for compositing and receiving occlusion maps, to avoid stalling individual nodes while they wait on other nodes. For transmitting the final image to the client (i.e., PC or IPad application), we use TCP sockets and support different image compression schemes.

3 Results

Table 1 illustrates the impact of different distribution unit sizes, different screen resolutions, and numbers of GPU nodes. We use two and four GPUs (NVIDIA Quadro 5000 with 2.5 GB memory) and a mouse cortex EM dataset (see Figure 2) of resolution $21,494 \times 25,790 \times 1,850 = 955$GB. The size of the virtual distribution units significantly influences the data distribution between nodes. Small distribution units result in a high depth complexity for compositing. Large distribution units lead to a low utilization of GPUs, because in the worst case only a single distribution unit will be in view, which is rendered by only a single node. The choice of an optimal distribution unit size depends on three major factors: the output screen resolution, the block cache size on each node, and the number of nodes. Currently, we are working on optimizing the compositing step and network communication between nodes.

References

[1] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *Proc. of Symp. on Interactive 3D Graphics and Games*, pages 15–22, 2009.

[2] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. Bergeron, and P. Hatcher. Large Data Visualization on Distributed Memory Multi-GPUs Clusters. In *Proc. of High Performance Graphics*, pages 57–66, 2010.

[3] W.-K. Jeong, J. Beyer, M. Hadwiger, R. Blue, C. Law, A. Vasquez, C. Reid, J. Lichtman, and H. Pfister. SSECRETT and NeuroTrace: Interactive Visualization and Analysis Tools for Large-Scale Neuroscience Datasets. *IEEE CG&A*, 30(3):58–70, 2010.

[4] A. Moerschell and J. D. Owens. Distributed Texture Memory in a Multi-GPU Environment. In *Graphics Hardware*, pages 31–38, 2006.

[5] C. Müller, M. Strengert, and T. Ertl. Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)*, pages 59–66. Eurographics Association, 2006.