Multi-GPU Volume Rendering using MapReduce

Jeff A. Stuart, Cheng-Kai Chen, Kwan-Liu Ma, John D. Owens (2010)

Ajitesh Shree
210079
Chemical Engineering
Indian Institute of Technology Kanpur
ajitesh21@iitk.ac.in

Kumar Kanishk Singh
210544
Statistics and Data Science
Indian Institute of Technology Kanpur
kksingh21@iitk.ac.in

September 12, 2023

1 Problem Statement

This paper aims to create a method for volume rendering that is scalable and efficient, particularly for large datasets, using multiple GPUs. It also seeks to reduce computation bottlenecks with the goal to provide a practical and high-performance solution for interactive volume visualization.

2 Motivation

- **Addressing Complexity**: As data sets grow in complexity, there's a pressing need to enhance our ability to visualize them effectively.
- **Adapting to Data Growth**: Traditional visualization techniques are struggling to keep up with the ever-expanding size of data sets.
- **GPU's Performance Promise**: Leveraging the high performance and cost-efficiency of GPUs can potentially revolutionize volume rendering.
- **Breaking Specificity Barriers**: Current volume rendering solutions are often overly specialized, limiting their widespread use across scientific domains.
- **Scaling GPU Solutions**: The scalability of GPU-based rendering needs improvement, as most are designed for single-GPU or small multi-GPU setups.
- **Efficiency via MapReduce**: Applying MapReduce principles to volume rendering offers efficient parallelism and holds the potential to address I/O communication challanges.

3 Background

3.1 Volume Rendering

Volume Rendering is widely used across various fields to visualize volumetric data, with the primary method being ray casting for structured data with regular grids, in which a transfer function maps scalar values to optical properties, accumulating them to compose the final image.
Here are some previous contributions on Volume Rendering:

- K.L. Ma introduced a parallel adaptive rendering algorithm to visualize massive datasets, aiming for scalability and high-fidelity visualization.
- Wang Gao, and colleagues proposed a parallel multi-resolution volume rendering framework designed for large-scale data.

- Yu presented a parallel visualization pipeline for tera-scale datasets, featuring adaptive rendering and an effective parallel-I/O strategy.
- Strengert introduced parallel ray-casting volume rendering methods on a cluster of GPU-equipped machines, combining hardware-accelerated ray casting with parallel volume rendering and I/O strategies.

3.2 MapReduce

MapReduce originated as separate functions, 'map' and 'reduce,' in functional programming, and was extended by Google for large-scale data processing. Various packages have been developed over time, notably Phoenix (C++ implementation) and Hadoop MapReduce (Java implementation).
Here are some other implementations and contributions done using this model:

- **i-MapReduce and Data Streams**: Innovated by using data streams instead of hard disk access, streaming intermediate values directly to new mapper and reducer nodes, enabling efficient, iterative algorithms.
- **Porting to Parallel Processing**: Recent efforts focus on adapting MapReduce to parallel-processing resources like GPUs and IBM's Cell processor.
- **GPU-Based MapReduce**: Catanzaro invented a GPU-based MapReduce library, primarily for small-scale tasks, with a focus on efficient sorting of mapping outputs on the GPU.
- **Mars**: The first large-scale GPU-based MapReduce system, designed to work on a single GPU within a single node, but limited to in-core datasets.
- **CellMR**: Single-node MapReduce system for Cell Engine, addressing in-core data limitations by streaming map data in small pieces and performing partial reductions on resident data.

4 Methodology

4.1 Workflow

There are four main stages to the MapReduce workflow: Map, Partition, Sort, and Reduce, aiming to complete the steps in parallel volume rendering, viz. partial ray casting on chunks and compositing ray fragments into final pixels.
Before these stages, we divide the data into chunks(also called bricks) and load them into the GPU memory for processing.

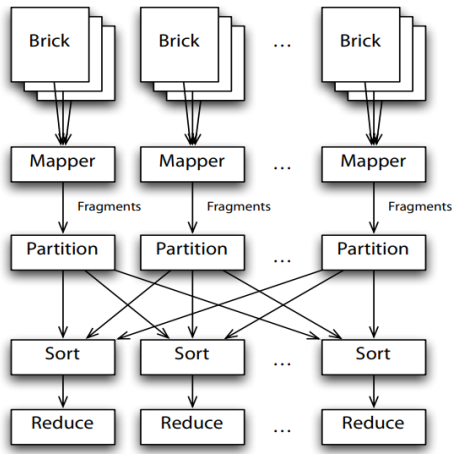


Figure 1: Workflow

Here is the description of the workflow:

1. **Mapper**: It is the first worker unit of our MapReduce library which executes ray-casting on the data to generate ray-fragments. Each mapper has an initialization function that allocates memory on GPU for loading the chunks and consecutively executing the ray-casting kernel on them.

2. **Partition:** In this stage, a key is assigned to the ray fragments using pixel index of the ray *(y(pixel)*width(finalImage) + x (pixel))* as the key and distributing them in a per-pixel round-robin fashion to the values.

3. **Sort:** In this stage, we use a specialized counting sort, which can be done on either the CPU or GPU, depending on the amount of data. This runs in θ(n) as the library knows the minimum and maximum keys for each node. This stage is required to create a global sort of the key- value pairs of the ray fragments before passing them to the reducer.

4. **Reducer:** The reducer composites the ray fragments generated in the previous stages to give final sub-images. The GPUs are good for compositing but it is quicker to do it on CPU due to the required sort of the ray fragments of each pixel.

4.2 Volume Renderer Implementation

The ray caster, implemented in CUDA, processes floating-point volume data stored in a 3D texture. Using a 2D grid of 16×16 blocks tailored to sub-image size, it intersects rays with a bounding box, employing non-adaptive trilinear sampling, early ray termination, and front-to-back compositing via a 1D transfer function for ray fragment color and opacity determination.
During the reduction phase, the same front-to-back compositing approach is used, with ray fragments for each pixel sorted by ascending depth, composited, and blended with the background color for the final output.

4.2.1 Alternate Approaches

For volume sampling there were two choices other than ray casting: splatting and slicing.

1. **Splatting:** In splatting, each voxel in the volume is projected onto the 2D screen as a textured polygon.

2. **Slicing:** In slicing, the 3D volume is virtually "sliced" along one or more planes to create 2D images.

We chose ray casting because of the nature of the GPU. With ray casting, each GPU thread was able to work independently of all others, and produced a uniform number of outputs.

For compositing, there two choices: Direct-Send Compositing and Swap Compositing.

1. **Direct-send Compositing:** It is a technique where each rendering node independently generates a portion of the final image and directly sends its contribution to a master node.

2. **Swap Compositing:** It is a technique where multiple rendering nodes work together to render the entire image in parallel, with each node responsible for a subset of pixels.

We chose direct-send compositing because it allows an overlap of communication and computation, and also because it is compatible with the MapReduce model.

4.3 Cluster Configurations

The authors used the Accelerator Cluster (AC) at the National Center for Supercomputing Applications (NCSA), USA, and tested with up to 32 GPUs. Each node has quad-core CPU, 8 GB SDRAM, and a Tesla C1090 with four logical GPUs each. The AC is connected via QDR Infiniband. Each node is running the Linux 2.6 kernel with the CUDA 3.0 toolkit and the NVIDIA 195

5 Results

The authors tested three datasets, Skull, Supernova, and Plume. The first two datasets are in resolutions of 128^3, 256^3, 512^3, and 1024^3 . The Plume dataset is in a resolution of $512 \times 512 \times 2048$.

5.1 Evaluation Metrics

To analyze the performance of this library and renderer, we consider three metrics: voxels per second (VPS), runtime, and parallel efficiency

1. **Voxels per Second (VPS):** VPS is the number of voxels rendered each second. It is significant because volumes grow larger cubicly while the renderings grow quadratically. It illustrates the performance of the software, and also the PCI-e bus and memory system of the GPU.
2. **Runtime:** Runtime is the total time taken time taken for the rendering. Frames per Second (FPS) is inversely proportional to runtime. A fast runtime is needed for interactive-rate visualization.
3. **Parallel Efficiency:** Parallel efficiency gauges how well the system can scale its performance as the data size and the number of GPUs increase.

It is to be noted that the runtime do not include the time taken to brick the volumes, nor the time taken to stitch the composited pixels. Neither of these tasks use this library, and both can be implemented in many different ways.

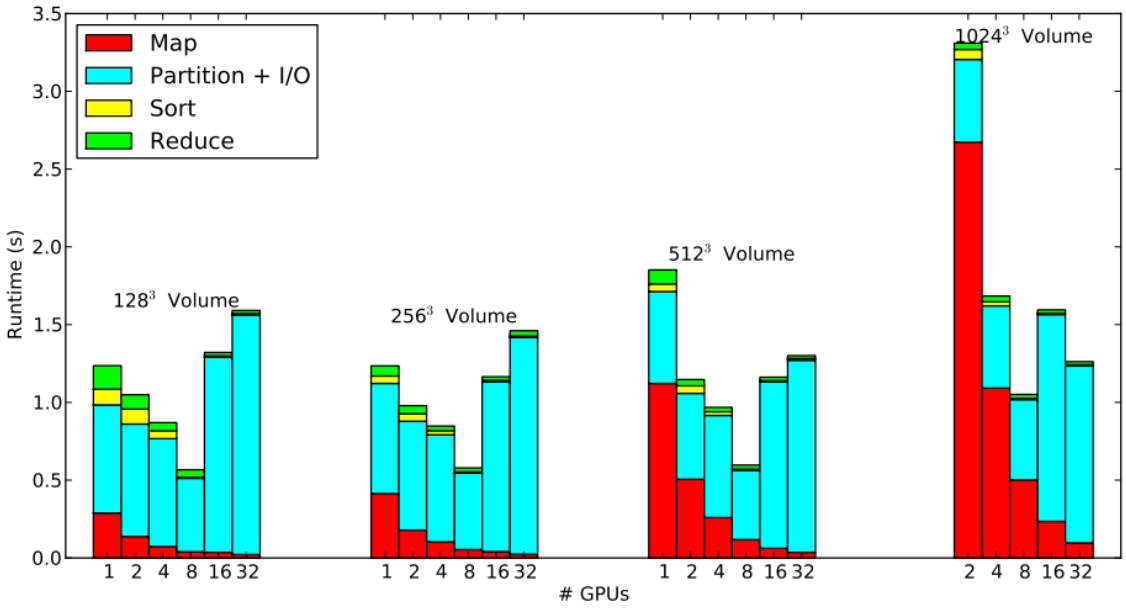
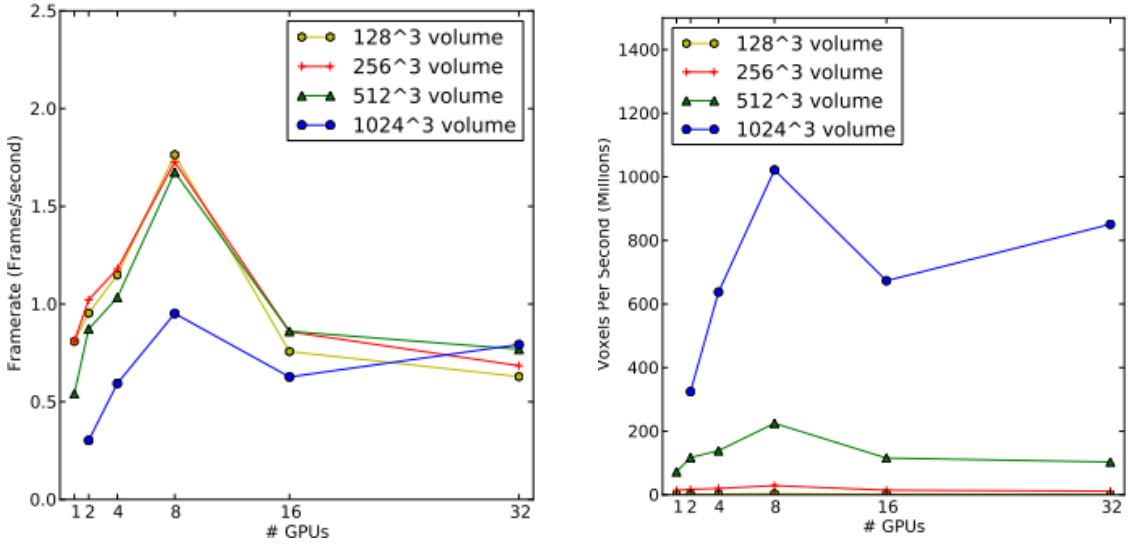


Figure 2: Runtime vs GPU



5.2 Inference

Let us take a look at the Figure 2 for runtimes for different number of GPUs.

- The times for map, sort and reduce tasks are decreasing linearly with the number of GPUs.
- The runtime first decreases then increases. This can be attributed to the extra communication required due to the increase in the number of GPUs.
- The best configuration is of 8 GPUs for all the resolutions. This is because there is a good balance between splitting work and minimizing communication.
- The computation time is increasing with the volume.
- For 1024^3 volume, the runtime for 32 GPUs is less than that for 16 GPUs. This is because the extra communcation is outweighed by the decrease in compute time. It can be inferred that with sufficiently large volumes, the performance increases should be seen beyond 8 GPUs.

We can note similar results in Figure 3 for VPS vs GPU and FPS vs GPU graphs as well.

- Both VPS and FPS are highest for 8 GPUs.
- For the 1024^3 volume, there is a decrease for 16 GPUs and increase for 32 GPUs. This is due to the same reason we talked about earlier.

6 Advantages

Some advantages of this method are:

- A bottleneck eliminated was the large computation time by using GPU, instead of CPU in volume sampling.
- Using MapReduce provided a highly modular design, meaning flexibility in altering volume-sampling technique or composting technique without the need to change both.

7 Limitations

There are currently two bottlenecks in this approach:

- Sometimes, a volume is of such large size that it cannot be accommodated within the system's memory, including GPU VRAM. In such cases, the data must either reside in virtual memory or be streamed in from external storage, such as disk or network resources.

- Reading bricks from disk can take several orders of magnitude more time than the entire MapReduce process. But, this is problem that all applications that rely on hard disks face.

8 Conclusion

- This library works well for configurations where the number of bricks is close to the number of GPUs.
- The renderer can effectively process input of any size.
- We can operate the renderer in either an in-core or out-of-core manner, and reduce bottlenecks as much as possible in both cases.
- When there are sufficient GPUs to fit the entire bricked volume in core memory, the advantages in terms of speed are evident.
- Even when the brick can't fit in the core memory, the rendering speed remains quite satisfactory.

References

[1] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. 2010.

[2] Jeff A. Stuart, Cheng-Kai Chen, Kwan-Liu Ma, and John D. Owens. Multi-gpu volume rendering using mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 841–848, New York, NY, USA, 2010. Association for Computing Machinery.