Multi-GPU Volume Rendering using MapReduce (2010)

Authors:

Jeff A. Stuart, Cheng-Kai Chen, Kwan-Liu Ma and John D. Owens

Presented by:

Kumar Kanishk Singh and Ajitesh Shree

What is this paper about?

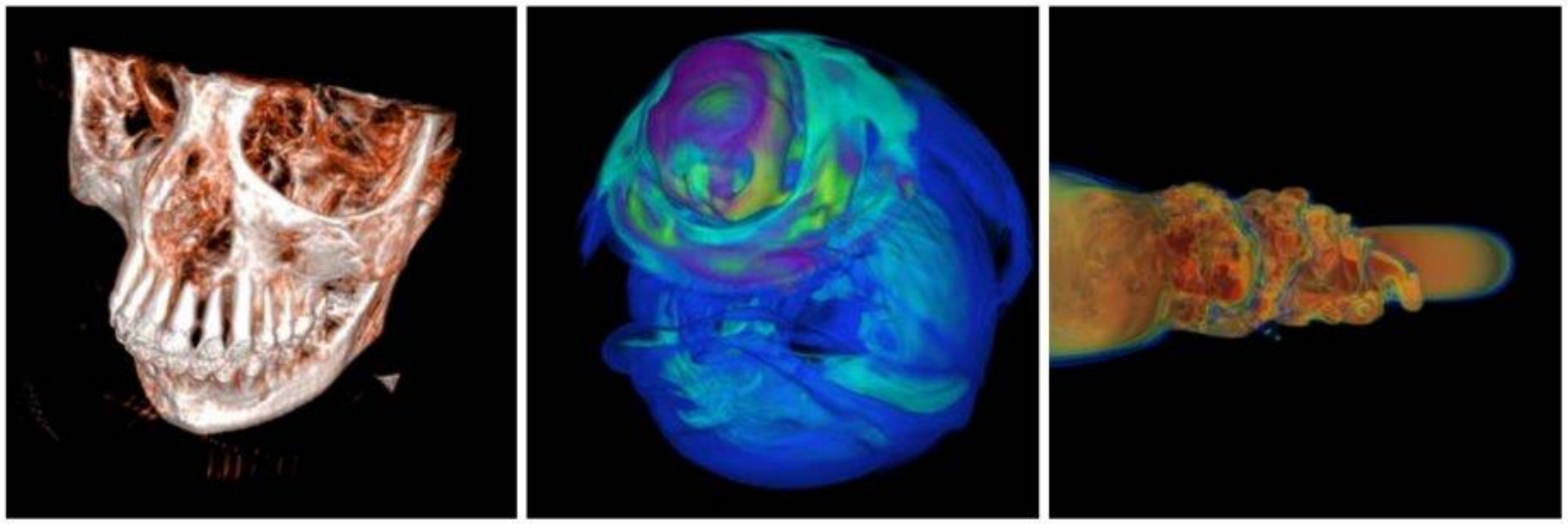
- A library made for volume visualization using Multiple GPUs and MapReduce model.
  - What is Volume Visualization?
    - A process of viewing volumetric datasets by creating graphical representations on a 2D screen.
  - Why use GPUs?
    - Provides great computational power.
    - Helps in interactive ray-casting visualization.
    - Are highly – parallel machines and volume visualization requires great level of parallelism.
    - With increasing data, GPU provides a cost effective alternative for rendering using thousands of CPUs with no decrease in performance.
  - Why MapReduce?
    - Well- known programming model that has an easy to use API.
    - Handles I/O communication, allowing to focus on computation more.
    - Allows a highly modular design.

Benefits of the created library

- Asynchronous Streaming Interface:
    - instead of storing intermediate key-value pairs and final reduced values to disk, it streams these values to the appropriate processes.
    - Benefits:
        - increases efficiency by allowing network communication, CPU data transfers, disk access and GPU kernel execution to all happen simultaneously.
        - allows the library to scale well, both with higher data-processing demands and as the number of GPUs increases.
- Easy-to-use Application Programming Interface:
    - Protocols that allows one software program to interact with and use the capabilities of another software. Inherent in every MapReduce package.
    - Benefits:
        - Enables the use of the GPU during both the Map and Reduce phases.

- Highly-pluggable code:
  - A programming approach designed to make adding, removing, or replacing modules in the code easy.
  - Benefits:
    - We can swap different volume resampling and compositing algorithms easily in and out.
- Parallelism allows it to scale well with increase in dataset size.
- Allows use of more nodes in a cluster for large datasets without affecting efficiency.
- Single GPU efficiently renders small in-core volumes, and only a minimal number of GPUs is required to efficiently render a volume out-of core.

A brief history of Volume Rendering



- Technique to visualize volumetric data.
- Most popular -> Ray Casting. Why? Generates high quality images with internal structure of complicated data.
- How does ray-casting work?
  - A ray is traversed through the volume for each pixel on 2D screen.
  - A transfer function is applied at each point of the volume to map a scalar values in to the optical properties like color and opacity.
  - These optical properties are accumulated along to composite a final color on the 2D screen.
- CPU- Cluster-based volume rendering
  - Way to visualize large data by distributing both data and rendering calculations to multiple compute nodes.
  - Benefits: High quality interactive rendering achieved.

- Previous works:
    - 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.

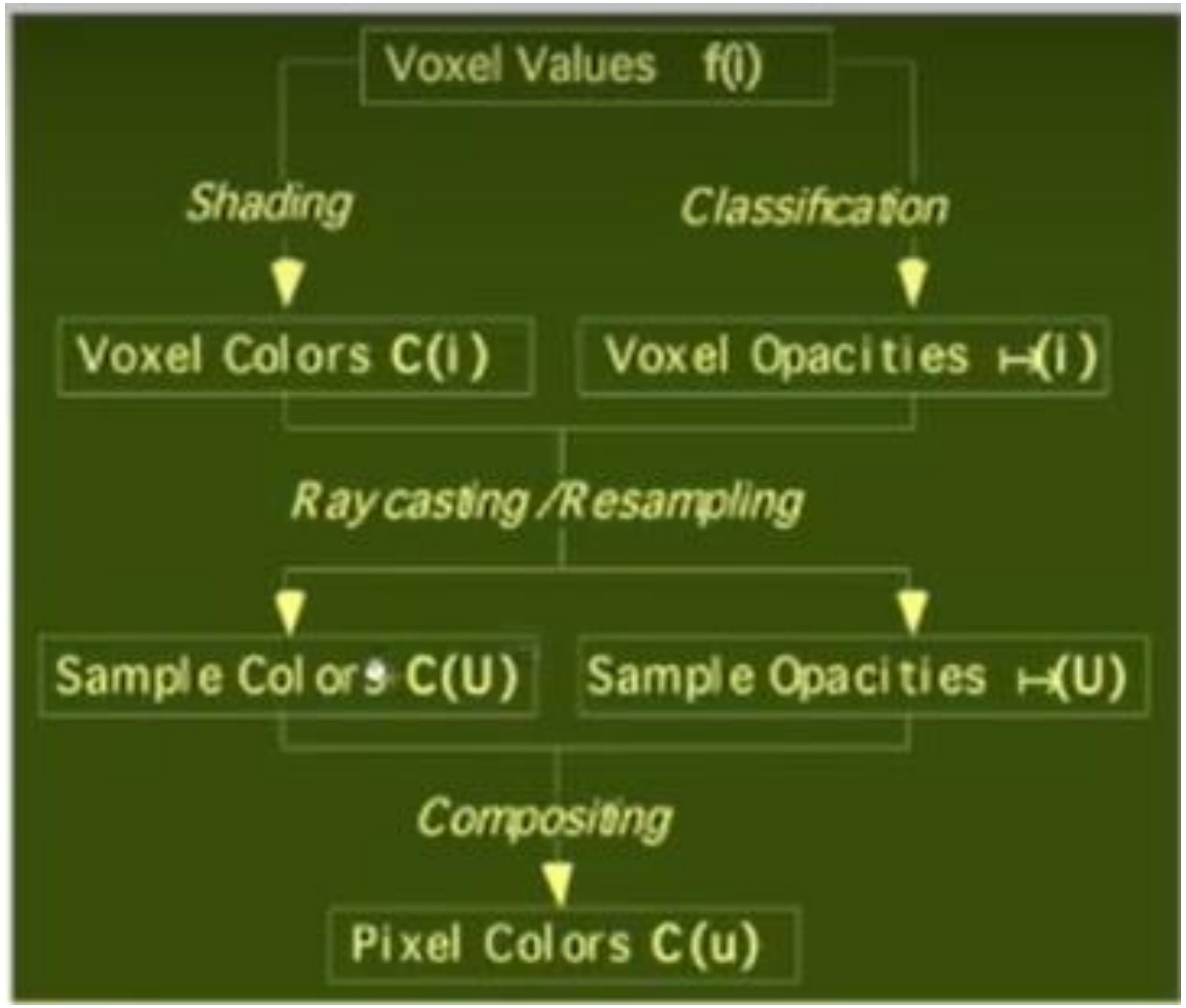
- History of Volumetric Ray Casting:
  - Kruger and Westermann introduced hardware-accelerated ray-casting rendering for structured volumetric data, leveraging programmable graphics hardware and 3D textures for interactive rendering.
  - Weiler developed a GPU-enhanced ray-casting renderer for unstructured tetrahedral meshes using 2D textures to encode mesh data.
  - Other researchers, including Kahler, Vollrath, and Gosink, explored various techniques for rendering adaptive mesh refinement datasets.

Background on MapReduce

- A programming model used for large-scale data processing.
- Began as two separate higher-order functions in functional-programming languages, map and reduce (also known as fold).
- Two Popular Packages:
    - Phoenix, a C++ implementation from Stanford.
    - Hadoop MapReduce, a Java implementation from the Apache foundation.
- i-MapReduce is the first to use data streams instead of hard disk access.

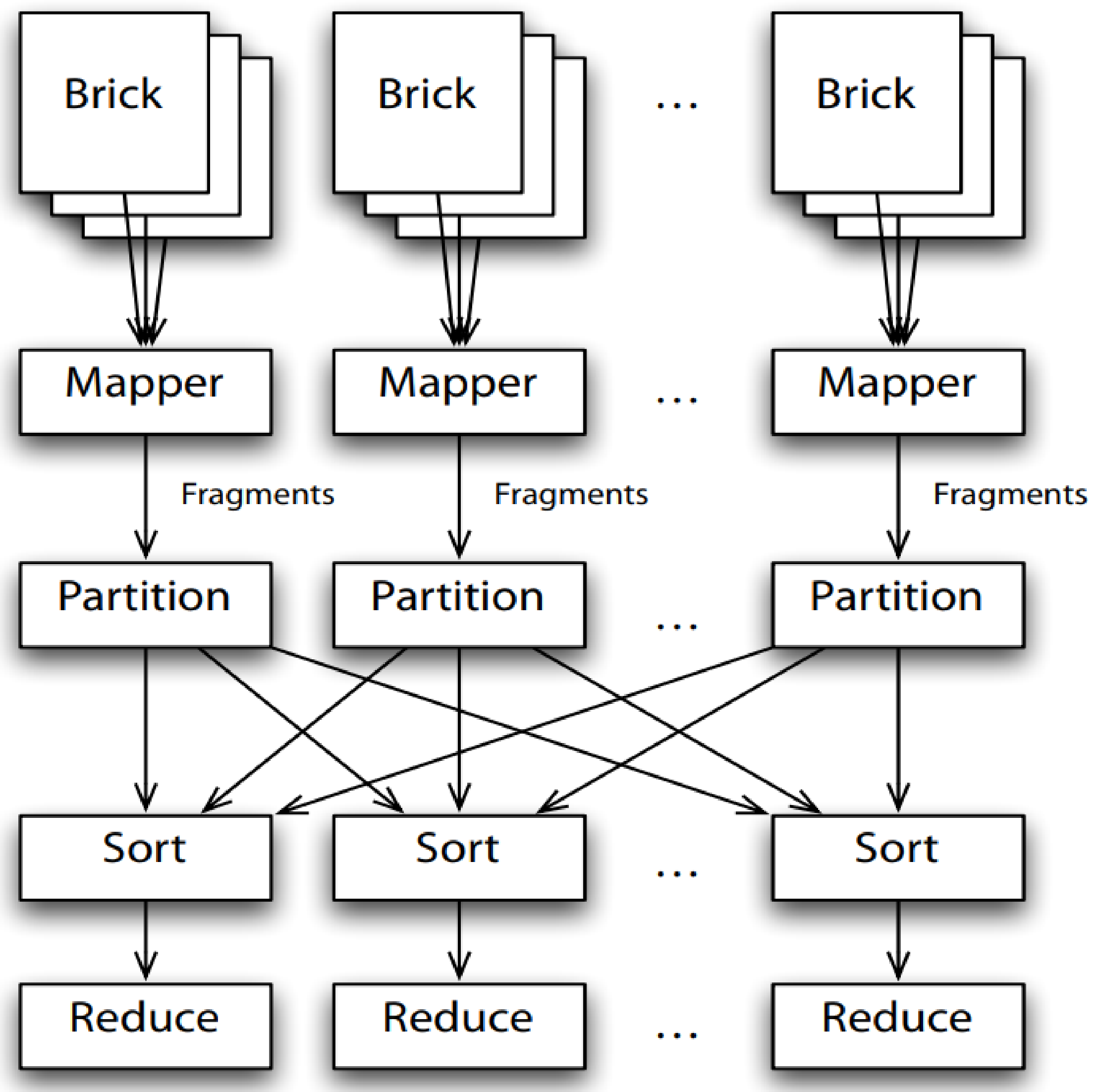
- Recent efforts: porting MapReduce to parallel-processing resources such as GPUs.
- Contributions by:
  - Catanzaro:
    - found an efficient way to sort small mapping outputs on the GPU.
  - Mars:
    - first large-scale, GPU- based MapReduce system.
    - works with a single GPU on a single node, but only on in-core datasets.
  - CellMR:
    - a single-node implementation of MapReduce on the Cell Engine that alleviates the in-core dilemma of Mars by streaming map data onto the compute devices in small pieces and performing partial reductions on resident data.

Implementation



- The two primary steps in parallel volume rendering are:
  - Partial ray casting against bricks of the volume to generate ray fragments.
  - Compositing sets of previously unsorted ray fragments into final pixels.
- Cluster of GPUs are used to execute these steps parallely which increases performance, by reducing ray-casting time.
  - **How?** Because transferring a brick of the volume to the GPU (<0.2ms) takes less than 1% of the time it takes to load a small chunk from the disk (20ms).
- There are four main stages to the MapReduce workflow we used : Map, Partition, Sort, and Reduce.

Workflow


Brick
Brick
...
Brick
Mapper
Mapper
...
Mapper
Fragments
Fragments
Fragments
Partition
Partition
...
Partition
Sort
Sort
...
Sort
Reduce
Reduce
...
Reduce

Workflow

- **Chunk:** A Chunk represents a collection of work to be mapped, in our case, it is a brick of a volume. Each chunk requests a certain amount of GPU memory. The library allocates this memory. The volume data is then copied to the GPU immediately before kernel execution.
- **Mapper:** Once a chunk is loaded into GPU memory, the library calls the execution function of the Mapper, which triggers the ray-casting kernel.

Workflow

- The goal of the Partition task along with Sort Task is to distribute the intermediate key-value pairs generated by the Mappers to the appropriate Reducer for further processing.
- **Partition:** We use the pixel index of the ray fragment as the key, and distribute the keys in a per pixel round-robin fashion. It is empirically, the highest-performing method
- **Sort:** We use a specialized counting sort to arrange the key-value pairs. We can do this either on the CPU or GPU, depending on the amount of data.
- **Reducer:** The reducer composites and blends the ray fragments generated by mapper.

Volume Renderer Implementation

- The ray caster is implemented in CUDA.
- The volume data is stored in a 3D texture with floating-point samples.
- The kernel is implemented with a 2D grid of 2D blocks. Each block is 16×16. The grid is made to match the size of the sub-image onto which a chunk projects.
- A non-adaptive trilinear sampling approach was employed.

Volume Renderer Implementation

- We implement early ray termination and front-to-back compositing, utilizing a 1D transfer function based on textures to determine the final color and opacity for each ray fragment.
- The same type of front-to-back compositing was used during the reduce phase. All ray fragments for a given pixel were ascending-depth sorted, composited, and blended against the background color

Alternate Approaches

- **Volume sampling:** we had three feasible options: ray casting, splatting, and slicing. We opted for ray casting as opposed to splatting and slicing due to the inherent characteristics of the GPU. With ray casting, each GPU thread could operate independently of the others, resulting in a consistent number of outputs.
- **Compositing:** We had two options, either direct-send compositing or swap compositing. We chose direct-send compositing because it allows an overlap of communication and computation, and also because it fits within the MapReduce model.

Results

Cluster Configuration

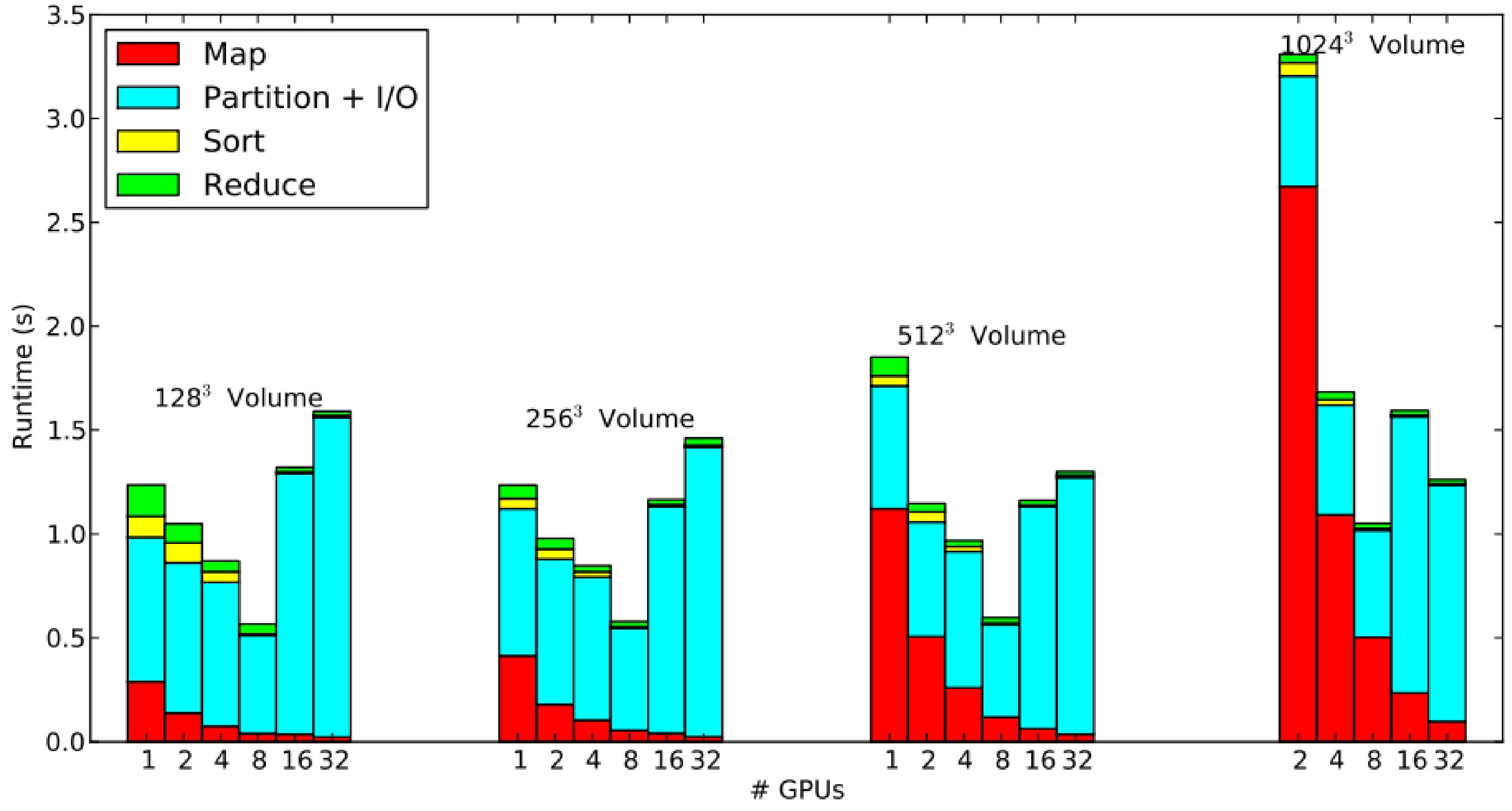
- We utilized the **Accelerator Cluster (AC)** at the National Center for Supercomputing Applications (NCSA), USA.
- The tests involved configurations of upto **32 GPUs**.
- Each node within the cluster has **quad-core CPU**, **8 GB SDRAM**, and a **Tesla C1090** with **four logical GPUs**.
- The cluster is interconnected via **QDR (Quad Data Rate) Infiniband**, providing high-speed communication between nodes
- Each node is running the **Linux 2.6 kernel** with the **CUDA 3.0** toolkit and the **NVIDIA 195**.

Evaluation Metrics

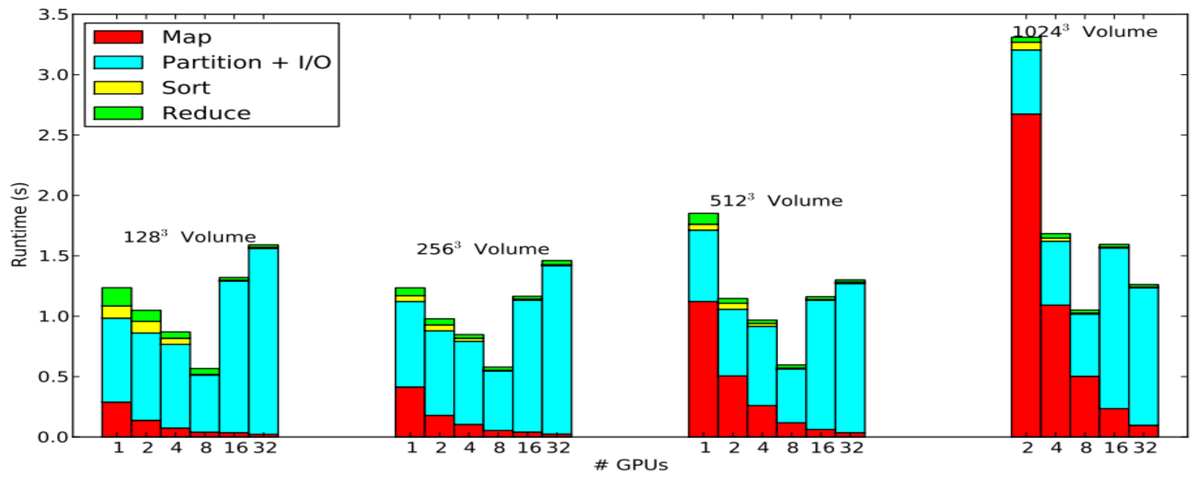
To evaluate the performance of the library and the renderer, we assess three key metrics:

- Voxels per Second (VPS): VPS 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.
- 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.
- Parallel Efficiency: Parallel efficiency gauges how well the system can scale its performance as the data size and the number of GPUs increase. Being able to double the number of GPUs and achieve almost twice the performance is a desirable goal

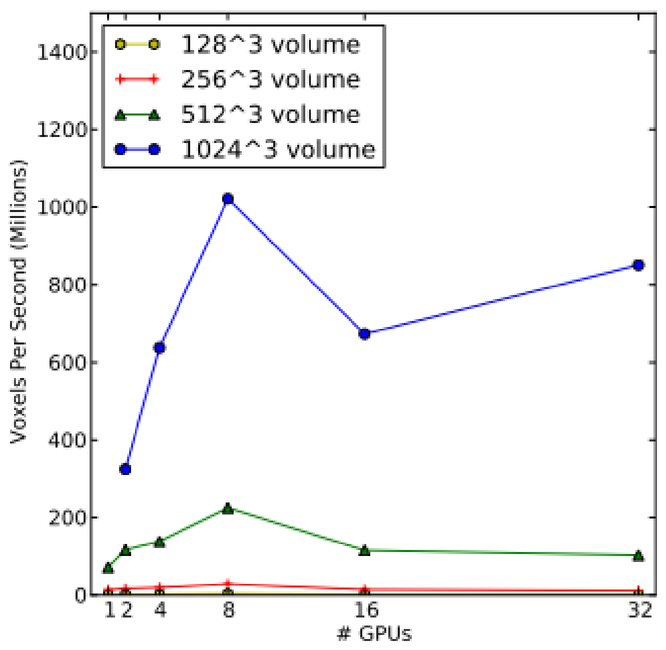
- We tested three datasets, Skull, Supernova, and Plume. The first two datasets are in resolutions of 128^3, 256^3, 512^3, and 1024^3. The Plume dataset is stored at a resolution of 512 × 512 × 2048.

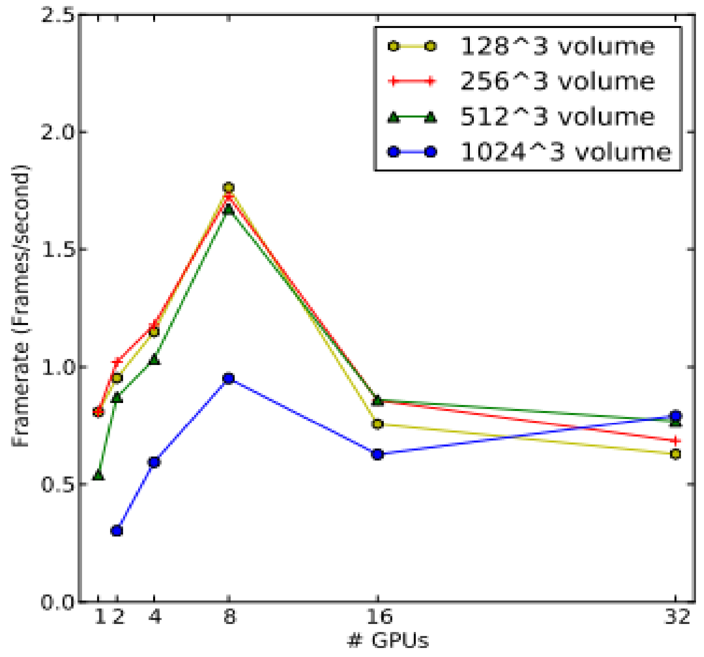


- The time taken for ray casting decreases linearly with the number of GPUs.
- The non-linear decrease or in some cases increase in the runtime can be attributed to the extra communication required with the large number of GPUs.
- As we can observe, the best runtime configuration is of 8 GPUs, as there is a good balance between splitting work and minimizing communication.
- The extra communication by using more GPUs is outweighed by the saving in compute time for 1024^3 volume.
- With sufficiently large volumes, we believe that performance increases should be seen beyond eight GPUs.



Results





- As we can see the maximum VPS and FPS are achieved for 8 GPUs.
- There is a slight increase for 1024^3 volume, even after the increase in GPUs used.

- Our 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.
- However, even when this isn't the case, the rendering speed remains quite satisfactory.

Bottlenecks

- 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.
- Computation time from ray casting is reduced as we are no longer using CPU.
- Reading bricks from disk can take several orders of magnitude more time than the entire MapReduce process.

Future Possibilities

- Extending our MapReduce implementation by making it more robust, more pluggable and thus more efficient for any kind of MapReduce task.
- Investigating the speed tradeoffs of using asynchronous memory transfers combined with manually filtering the volume samples in shared memory, as opposed to using the synchronous memory transfer functions and hardware filtering units.
- Exploring the benefits of direct access for the GPU to system memory (0-copy memory)
- It could potentially be more efficient to keep the final pixels on the GPU, enabling them to be rendered right after the compositing concludes.

Thank You