Multi-GPU Volume Rendering using MapReduce

Jeff A. Stuart
UC Davis
stuart@cs.ucdavis.edu

Cheng-Kai Chen
UC Davis
ckchen@ucdavis.edu

Kwan-Liu Ma
UC Davis
ma@cs.ucdavis.edu

John D. Owens
UC Davis
owens@ece.ucdavis.edu

ABSTRACT

In this paper we present a multi-GPU parallel volume rendering implemention built using the MapReduce programming model. We give implementation details of the library, including specific optimizations made for our rendering and compositing design. We analyze the theoretical peak performance and bottlenecks for all tasks required and show that our system significantly reduces computation as a bottleneck in the ray-casting phase. We demonstrate that our rendering speeds are adequate for interactive visualization (our system is capable of rendering a 1024^3 floating-point sampled volume in under one second using 8 GPUs), and that our system is capable of delivering both in-core and out-of-core visualizations. We argue that a multi-GPU MapReduce library is a good fit for parallel volume renderering because it is easy to program for, scales well, and eliminates the need to focus on I/O algorithms thus allowing the focus to be on visualization algorithms instead. We show that our system scales with respect to the size of the volume, and (given enough work) the number of GPUs.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; I.3.2 [**Computer Graphics**]: Graphics processors

General Terms

Design

Keywords

GPU, MapReduce, Volume Rendering

1. INTRO

Volume visualization is an important tool used by scientists to better understand complex data sets. These data sets are becoming ever larger while the visualization techniques tend to stay the same. Large clusters, of machines sometimes thousands of CPUs, are used to render data. These clusters have mostly retained their relevance in the past decade. However, GPU visualization techniques are a viable alternative because of their performance, as well as the cost-performance advantage of GPUs over CPUs.

A common problem with many volume renderers is the level of specificity in their code base. Most renderers are specific to one application domain. The programmers of such packages often write special optimizations based on the types of data to be explored and the desires of the application scientists. Because of the very-specific nature of such renderers, they are not widely used within the scientific community. Even if they were shared, the code is of little use to scientists outside of the application domain. GPU renderers are especially guilty of this. They offer great computing power, but often require special optimizations for efficiency.

The GPU is a highly-parallel machine and offers enormous performance benefits over a CPU for problems that have a high degree of data-level parallelism. Volume rendering (ray casting specifically) works well with the GPU because it has a high degree of parallelism. However, GPU renderers have a problem with scalability; they tend to be targeted towards either a single-GPU machine, or a single-node multi-GPU machine.

There are, of course, exceptions. ParaView [1] and VisIt [3] are both equipped with capable renderers, provide an extensible API, and are widely distributed within the community. They lack out-of-the-box GPU support though. A library is needed that, like these two packages, provides a scalable infrastructure, but that also provides an inherent ability to utilize GPUs. To make the library even more promising, it is important to use a well known model and an easy-to-use application-programming interface (API). MapReduce [4], modified by Google from the original *map* and *fold* paradigms of functional programming, is exactly this. The one thing that virtually all MapReduce libraries lack though is GPU support.

Volume rendering has many characteristics that would adapt well to a GPU-enabled MapReduce library. Volume rendering requires a large amount of computation, which is amply supplied by GPUs. MapReduce is good at hiding communication requirements behind such computation, something that any good volume renderer must do.

In this paper, we present our work on a highly-specialized, multi-GPU MapReduce library that we use as a substrate for volume rendering. The library has an asynchronous, streaming interface that increases efficiency by allowing network communication, CPU/GPU data transfers, disk access,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MAPREDUCE 2010 Chicago, Illinois USA
Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

and GPU kernel execution to all happen concurrently. The library allows for highly-pluggable code; different volume-resampling and compositing algorithms can easily be swapped in and out. The library provides an easy-to-use API that enables the use of the GPU during both the Map and Reduce phases. As with any other MapReduce library, our library handles all I/O, thus allowing the user to focus on the computation and not the communication. It is important to note that the library handles I/O in this manner without a significant loss of performance over a custom solution. As an aside, we would like to point out that our library does not provide three common aspects found in other MapReduce libraries: fault tolerance (for our experiments and our small cluster, this was not an issue), advanced scheduling (with our renderer this was also not an issue), and a distributed file system.

Our library handles data in a streaming manner; instead of storing intermediate key-value pairs and final reduced values to disk, the library streams these values to the appropriate processes. This overcomes a bottleneck present in many other MapReduce libraries and allows the library to scale well, both with higher data-processing demands and as the number of GPUs increases (especially with more than one GPU per compute node). And even though our library has no explicit disk access, it still allows for out-of-core algorithms (including rendering), something current GPU MapReduce libraries do not allow.

To analyze our library, we tested runtimes for volumes of various sizes under many different configurations. Our results show that we can significantly reduce computation as a bottleneck for volume rendering, even with only a small number of GPUs, and by doing so we can more clearly see where disk accesses, CPU/GPU data transfers, and network I/O become bottlenecks in the multi-GPU volume-rendering process with current system architectures.

The rest of this paper proceeds as follows. Section 2 presents a background on volume rendering and MapReduce. We discuss our implementation of our multi-GPU MapReduce-based volume renderer in section 3. We then present a methodology for evaluating our volume renderer in section 4 and present out results in section 5. Section 6 presents a discussion on our results and their implications, and we draw conclusions about multi-GPU volume rendering in section 7.

2. BACKGROUND

2.1 Volume Rendering

Volume rendering is a technique widely used to visualize volumetric data sets in various fields of study. There has been a large amount of research devoted to the area. Ray casting is the most popular method for volume rendering due to its effectiveness in generating high quality images where internal structures of complicated data can be displayed. The standard ray casting techniques were proposed by Levoy [14] for structured data sets with regular grids. The fundamental idea of ray casting is that, for each screen pixel on the plane, a single ray is traversed from the eye into the volume. During ray casting, a transfer function that maps a scalar value to optical properties, including color and opacity, is applied at each sample point. These colors and opacities are then accumulated along a ray to composite a final color on the screen.

Ray-casting volume rendering using CPUs is computationally expensive since it requires the interpolation and shading calculations for every sample point along the ray in the data. Interactive volume ray casting was previously restricted to high-end workstations until GPUs became fast enough to perform such computations. GPU implementations of ray-casting rendering approaches have received much attention since they enable interactive visualization of volumetric data. Krüger and Westermann [12] proposed hardware-accelerated ray-casting rendering techniques for structured volumetric data. By utilizing programmable graphics hardware including vertex and fragment shaders, and storing volume data in 3D textures, these renderers achieve an appropriate rendering quality at interactive rates. We followed ideas similar to those presented in these papers to implement our GPU-based ray-casting renderer. Weiler et at. [18] introduced a GPU-enhanced ray-casting renderer for unstructured tetrahedral meshes. Based on Garrity's algorithm, they used 2D textures to encode mesh vertices, corresponding scalar values, and connectivity information, to be able to march through the tetrahedral mesh during rendering. Kahler et al. [11], Vollrath et al. [27], and Gosink et al. [7] all demonstrated different techniques for rendering adaptive mesh refinement datasets.

CPU-cluster based volume-rendering methods provide feasible solutions to large data visualization by distributing both data and rendering calculations to multiple compute nodes. In this way, high-quality interactive rendering of large scale data can be achieved. Ma et al. [17] presented a parallel adaptive rendering algorithm for visualizing massive data. Their goal was to come up with a scalable, high-fidelity visualization solution that allowed scientists to explore many domains of their data. Wang, Gao, and their colleagues [6, 28] proposed a parallel multiresolution volume rendering framework for large-scale data. Yu et al. [29] proposed a parallel visualization pipeline for studying terascale datasets. Their solution is based on a parallel adaptive-rendering algorithm coupled with a new parallel-I/O strategy which effectively reduces interframe delay by dedicating some processors to I/O and preprocessing tasks. Hsieh et al. [10] developed a set of interactive 3D-visualization and exploration algorithms based on distributed computing, level-of-detail mesh construction, and view-dependent refinement. Their work significantly improved rendering capacity and facilitated the exploration large-scale data analysis. Strengert et al. [26] presented parallel ray-casting volume-rendering methods on a cluster of commodity, GPU-equipped machines. Their work combined hardware-accelerated ray casting with traditional parallel volume-rendering techniques and I/O strategies.

2.2 MapReduce

MapReduce is a programming model that began as two separate higher-order functions in functional-programming languages, *map* and *reduce* (also known as *fold*). Google extended the model for large-scale data processing and Google popularized MapReduce with industry and academia. Since that time, developers and researchers have created many MapReduce packages. Two popular packages are Phoenix [25], a C++ implementation from Stanford, and Hadoop MapReduce [13], a Java implementation from the Apache foundation.

i-MapReduce [5] (previously referred to as CGL MapRe-

duce) is another MapReduce package, and potentially the first to use data streams instead of hard disk access. Instead of sending intermediate data values and reduction results to disk, they are streamed directly to new mapper and reducer nodes for further processing. This allows for an efficient, iterative MapReduce algorithm with many consecutive MapReduce processes.

Recent efforts have gone towards porting MapReduce to commodity parallel- processing resources such as GPUs and IBM's Cell. Catanzaro et al. created a MapReduce library for GPUs [2], though the primary focus of his work was performing many small-scale MapReduce tasks. The main contribution of their work was finding an efficient way to sort small mapping outputs on the GPU. Mars [9] was the first large-scale, GPU- based MapReduce system. It works with a single GPU on a single node, but only on in-core datasets. CellMR [24] is a single-node implementation of MapReduce on the Cell Engine that alleviates the in-core dilemma of Mars. CellMR accomplishes this by streaming map data on to the compute devices in small pieces and performing partial reductions on resident data. The problem with CellMR is that it relies on these partial reductions to eliminate as much I/O as possible. Given the formulation of our volume renderer, we would rarely see any benefits.

3. IMPLEMENTATION

Parallel volume rendering, particularly ray casting against bricked input with partial-ray compositing, has unique and well-defined computation and memory-, disk-, and network-access characteristics. We argue that these characteristics make volume rendering well suited for parallel rendering on a cluster of GPUs.

The two primary steps in parallel volume rendering are partial ray casting against bricks of the volume, and compositing a sets of previously unsorted ray fragments into final pixels. Both of these tasks are embarrassingly parallel; however, communication is required to progress from one to the other.

Using a cluster of GPUs to execute each of these steps results in a performance increase, even though the cost of communication increases slightly. Loading even a small brick from disk can take a substantial amount of time; loading a 64^3 block from disk takes approximately 20 ms on our cluster. Transfering that brick to the GPU takes less than 0.2 ms (less than 1% overhead) and we achieve a very significant decrease in ray-casting time. This is because the VRAM of modern GPUs is more than 10X faster than that of modern CPU DRAM, and the GPU has eight texture-fetch units and eight texture-filtering units. Transmitting final ray fragments from the GPU to the CPU also requires very little time (empirically found to be less than 2 ms).

For a given image of X pixels with Y nodes and B bricks, the lower bound on generated ray fragments is $O(X)$ and a loose upper bound is $O(BX)$. Of course, the actual number is affected by the volume itself (ray fragments with no contributions are discarded) and the view (the more view-dependent overlap between bricks, the more potential for a higher number of ray fragments). Given the worst-case scenario, each of the Y nodes must make $Y-1$ communication requests of size $sizeof(Ray\ Fragment) \times O(\frac{BX}{Y})$. A GPU-to-CPU transfer of the finished ray fragments must first complete before network I/O can commence. Just as was the case with loading the volume data, the network transmission time is several orders of magnitude higher than the GPU-to-CPU transfer time of those ray fragments. And of course, just as a CPU-based volume renderer can overlap sending of ray fragments with computing more ray fragments, so can a GPU-based renderer.

Thus, with a marginal increase or even a possible decrease of disk, network, and memory access through all stages of the volume-rendering pipeline, we believe that a distributed multi-GPU scheme is well-suited to the parallel volume- rendering workflow. We proceed to describe our implementations of a parallel volume renderer and the multi-GPU MapReduce library that serves as its substrate.

3.1 MapReduce Implementation

Our implementation of MapReduce was written in C++ and CUDA [22, 21, 15] to take advantage of NVIDIA GPUs. The implementation was written to be easy-to-use and extensible; all user-required tasks are represented via objects with virtual functions used as callbacks.

There are four main stages to the MapReduce workflow: Map, Partition, Sort, and Reduce (we specifically omitted partial reduce/combine because it didn't increase performance for our volume renderer). Each of these stages are available for user customization by inheriting and extending from virtual classes. Figure 1 shows a visual diagram of our system.

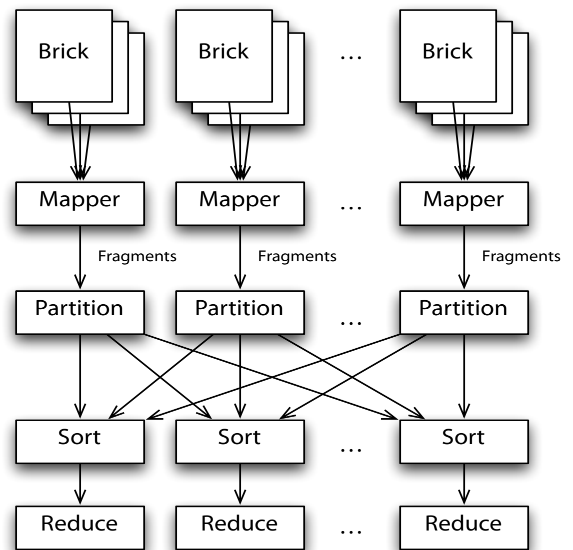

Figure 1: The MapReduce workflow: Bricks are individually streamed to the GPU *Mapper*. The output from each kernel execution is a set of ray fragments that are sent to the *Partition* phase. As rays are partitioned, they are sent to the appropriate process for the *Sort* phase, and then finally are sent to a *Reducer*.

3.1.1 Restrictions

In an effort to make the library as efficient as possible, we made a number of optimizations and design decisions that are specific to volume rendering, but would not necessarily work well for every MapReduce task.

- Any single map (ray casting) task must be able to fit in the main memory of the GPU. Large volumes must be bricked and mapped in stages.
- Keys are always four-byte integers. If a key X exists, then all keys $0 \leq X$ have a high probability of existing. This allows for easy partitioning and efficient binning.
- Emitted values are homogeneous in size and computed in GPU local memory. This allows for optimizations when copying key-value pairs to global GPU memory (to then be copied to the CPU).
- Every GPU thread must emit a key-value pair. If the thread computes a useless key-value pair, the kernel emits a later-discared place holder.
- Partitioning is done in a per-pixel round-robin fashion. This is, empirically, the highest-performing method. A modulo is sufficient to determine the reducer to which a key-value pair must be sent.
- Any single reduce (compositing of a single pixel) task must have the ability to completely reside in the main memory of the GPU. Many reductions can be scheduled per kernel to achieve high throughput.

3.1.2 Stages and Objects of our MapReduce Work Flow

The volume data is bricked into small pieces, with each piece represented as a *Chunk*, and distributed to a *Mapper* that executes a ray-casting kernel for each *Chunk*. A *Partition* is performed on all key-value pairs to find their respective *Reducer* process. Once enough pairs have been generated by a Mapper, they are sent asynchronously to the *Reducer*. Once all *Mappers* have finished and all data has been routed to the proper *Reducer*, a *Sort* is performed to compact all like keys and arrange their respective values together. Finally, each *Reducer* iterates through the data.

Chunk.

A *Chunk* represents a collection of work to be mapped, in our case, it is a brick of a volume. As GPUs are highly data-parallel machines, they are more performant when large amounts of data need to be processed. Each *Chunk* requests a certain amount of GPU memory to hold its volume data. For efficiency reasons, the library allocates this memory. The volume data is then copied to the GPU immediately before kernel execution. Our wish was to do this asynchronously, but in order to use a CUDA 3-D texture, we were forced to use synchronous memory copies. We chose not to put any GPU-memory [de]allocation functions in the staging functions because these functions are implicitly synchronous and have a high overhead.

Mapper.

The *Mapper* is the first worker unit of our MapReduce library. *Mappers* execute a ray-casting kernel on each *Chunk*. Each *Mapper* has an initialization function that allocates static data on the GPU (e.g. view matrix). This function can allocate GPU memory and need not worry about asynchronous activity since it is called at the very beginning of the MapReduce process. Once a chunk is ready, the library calls the execution function of the Mapper, this triggers the ray-casting kernel.

Partition.

The *Partition* task is implicit in the library and leverages some pre-existing knowledge about the problem domain. We use the pixel index of the ray ($y(pixel) * width(finalImage) + x(pixel)$) as the key, and distribute the keys in a per-pixel round-robin fashion because it is empirically the most performant method of distribution

Sort.

Sort also is implicit in the library. We use a specialized counting sort on the CPU or GPU (depending on the amount of data) that runs in $\theta(n)$ since the library knows the minimum and maximum keys for each node, as well as the maximum number of keys for each node.

Reducer.

The *Reducer* tasks can execute on either the CPU or the GPU. We found empirically that while the GPU would be very good at compositing (compositing is a very data-parallel task), it is actually quicker to do the compositing on the CPU. This is because of the required sort of the ray fragments of each pixel. And while the image stitching is not included in our results or timings (it is a separate phase from *Map*, *Sort*, *Partition*, and *Reduce*), it must happen in parallel, and thus would require more data be copied up from the GPU. We believe that with sufficiently many ray fragments (from hundreds or thousands of GPUs), GPU compositing may be more efficient.

3.2 Volume Renderer Implementation

We implemented our ray caster in CUDA. The volume data is stored in a 3D texture with floating-point samples to enable the hardware texture caches and filtering units, thus leading to a substantial increase in volume-sampling throughput. We execute our kernel with a 2D grid of 2D blocks. Each block is 16×16, and the grid is made to match the size of the sub-image (with a potentially small amount of padding) onto which the current chunk projects. In the kernel, we use a similar implementation to that proposed by Hadwiger et al. [8]. All rays are intersected against a bounding box and any non-intersecting rays are immediately discarded. We use non-adaptive trilinear sampling; rays are advanced through the chunk at fixed increments until they exit. We use early ray termination, and we use front-to-back compositing with a texture-based 1D transfer function to obtain the final color and opacity of each ray fragment. We use the same type of front-to-back compositing during the reduce phase. All ray fragments for a given pixel are ascending-depth sorted, composited, and blended against the background color.

4. METHODOLOGY

4.1 Cluster Configuration

To test our library and volume renderer, we used the Accelerator Cluster (AC) at the NCSA, 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.

4.2 Benchmarks

When trying to analyze the performance of our library and renderer, we consider three figures of merit: voxels per second (VPS), runtime, and parallel efficiency.

Voxels per second is an important figure to consider because volumes grow larger cubicly while the renderings grow quadratically. Showing the capability of a renderer in terms of VPS illustrates the performance of the software, and also the PCI-e bus and memory system of the GPU.

Runtime is just as important as VPS. Scientists care about the frame rate of their visualization. This has an inverse correlation to the runtime. The slower the runtime, the more GPUs required to an interactive-rate visualization.

Finally, parallel efficiency is important because it shows the true scalability of the system, both in terms of data size and in terms of number of GPUs. Being able to double the number of GPUs and achieve almost twice the performance is a desirable goal. Our main goal is runtimes of interactive rates and a system that scales well to many GPUs as the size of the data grows.

5. RESULTS

We tested three datasets, *Skull*, *Supernova*, and *Plume* (Figure 2). We have the first two datasets in resolutions of 128^3, 256^3, 512^3, and 1024^3. The *Plume* dataset is stored at a resolution of $512 \times 512 \times 2048$. All volumes use four-byte floating- point samples and for the purposes of timing, were rendered with an image size of 512^2 (trends in speedup and runtime were similar with different image sizes, this was an arbitrarily chosen representative). Our timings do not include the time taken to brick the volumes, nor the time taken to stitch the composited pixels. Neither of these tasks use our library, and both can be implemented in many different ways. Instead, our results focus on our library.

Figure 3 shows the breakdown of runtimes by *Map*, *Partition*, *Sort*, and *Reduce*, for a 128^3, 256^3, 512^3, and 1024^3 volume. Figure 4 shows the FPS and VPS rates of our renderer. [1]

6. DISCUSSION

All of the design decisions that went into the library and the renderer were made with two things in mind: performance and efficiency. We strived to use existing practices from the research community. At times though, there was no clear solution provided in the literature. In this case we used empirical evidence to guide our decisions.

For volume sampling, we had three viable choices: ray casting, splatting, and slicing. We chose ray casting over splatting and slicing 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, allowing the output values to be written to GPU main memory in an efficient manner.

The choice for compositing was also not initially obvious. We had two options, either direct-send compositing [20] with a checkerboard, tiled, or striped distribution, or swap compositing [16, 30]. We chose direct-send compositing because it allows an overlap of communication and computation, and also because it fits within the MapReduce model. With our renderer, it is quite common to have each GPU render several chunks.

By building our renderer and MapReduce library in this way, we created something that works well for configurations where the number of bricks is close (roughly within a factor of four) to the number of GPUs. The renderer is capable of efficiently handling arbitrarily-sized input. We can run the renderer in either an in-core or out-of-core manner and reduce bottlenecks as much as possible in both cases. If enough GPUs are available to fit the bricked volume entirely in core, the speed benefits are obvious. But if not, the speed of the rendering is still quite good, and better than out-of-core CPU renderers.

6.1 Advantages

We would again like to mention an important benefit of using MapReduce: it allows for a highly modular design. It is straightforward to change either the volume-sampling technique or the compositing technique, without changing both. For example, swap compositing can be implemented by changing the partitioning on each node. Every node would consume all generated ray fragments to create its partial image. The reduction phase would then be changed to perform swap compositing. If the user wished to use splatting or slicing instead of ray casting, the map phase is all that would need to be changed.

One notion to be taken from the results is that our ray casting method is perhaps *too* easy for the GPU. The transfer time of a brick to the GPU and the generated ray fragments from the GPU take almost as much time as ray casting against the brick. And in fact, as the number of GPUs grows large, the communication time for fragments is the dominant part of the algorithm. For a small volume with only a few bricks, the time for these three combined can be far less than the time required for direct-send compositing. Thus, small inputs do not scale very well in terms of the number of nodes. Of course this is not actually a problem; why would one wish to use more resources than necessary to complete a task? Our system reacts extremely well when the rendering is performed with just enough (or less than enough) GPUs to hold the volume.

6.2 Limitations

Another problem encountered is thrashing. Sometimes a volume is so big that it simply cannot fit in system memory, let alone GPU VRAM. When this is the case, the data must reside in virtual memory or it must be streamed in from disk or from a network resource. For non-in-situ visualization, the data most likely will be stored on disk (either in virtual memory or in the file system). Relying on virtual memory *severely* impacts the performance of any application, and our renderer is no exception. However, we feel that this is not a shortcoming of the library and is more a problem that all applications that rely on hard disks must overcome (our library is hard-disk agnostic).

6.3 Analysis of Bottlenecks

We argue that based on the bottlenecks presented in 6.2, the combination of our library and renderer are as efficient as possible; meaning that the computation from ray casting

[1] Comparing different volume renderers is difficult because of differences in hardware (e.g. CPUs, GPUs, interconnect, etc.) and the rendering algorithm. However, as a point of reference, Moreland et al. [19] show that Paraview can render 346M VPS using 512 processes on 256 nodes. Using 16 GPUs on 4 nodes, we achieve more than double this rate.

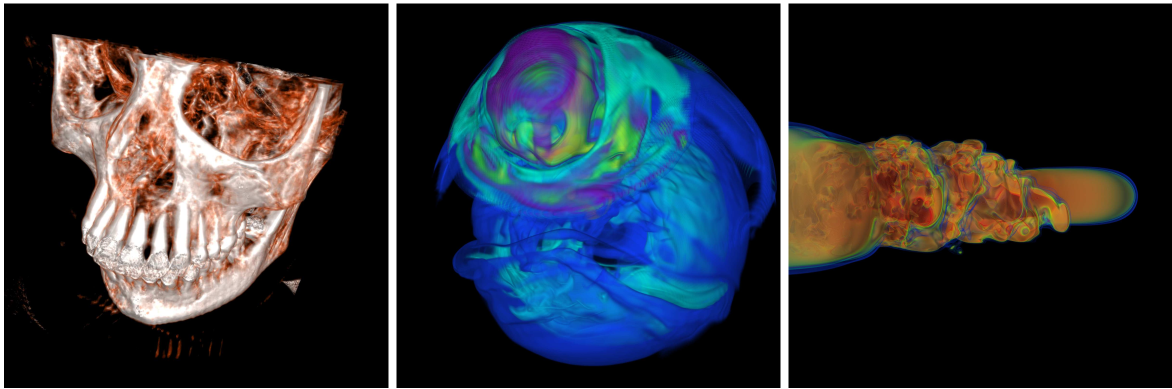

Figure 2: The *Skull*, *Supernova*, and *Plume* datasets.

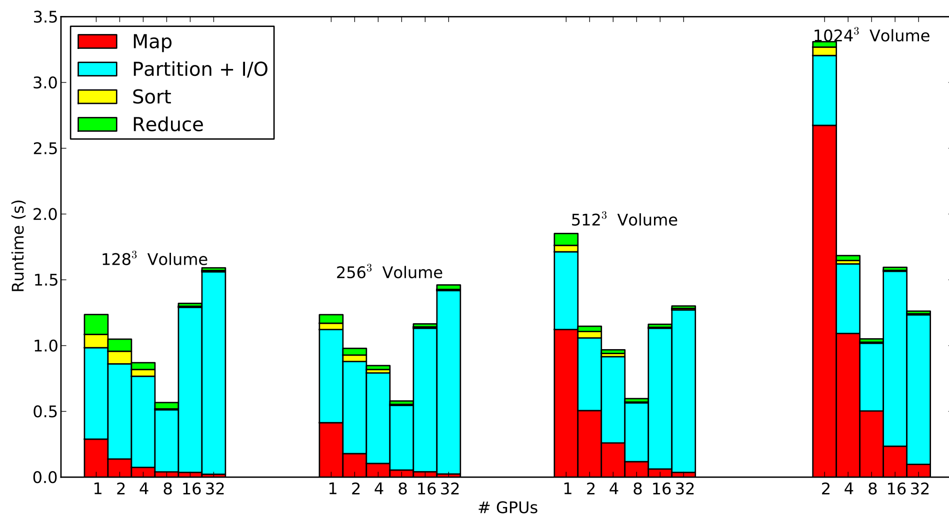

Figure 3: The relative cost of compute-to-communication can be seen from these graphs. The total time taken to ray cast (a portion of the Map phases) scales linearly with the number of GPUs. The not-quite linear decrease, or in some cases increase, as the number of GPUs increase is due to the extra communication required. As more GPUs are added, more ray fragments generated, and thus more time is required for reduce and more time is required for communication. With volumes of this size, the best runtime configuration is 8 GPUs, primarily because it strikes a good balance between splitting work and minimizing communication. With less than 8 GPUs, there is too much work, with more than 8 GPUs, there is too much communication. The 1024^3 volume shows a certain trend: the additional communication with 32 GPUs over 16 GPUs is outweighed by the saving in compute time. With sufficiently large volumes, we believe that performance increases should be seen beyond eight GPUs.

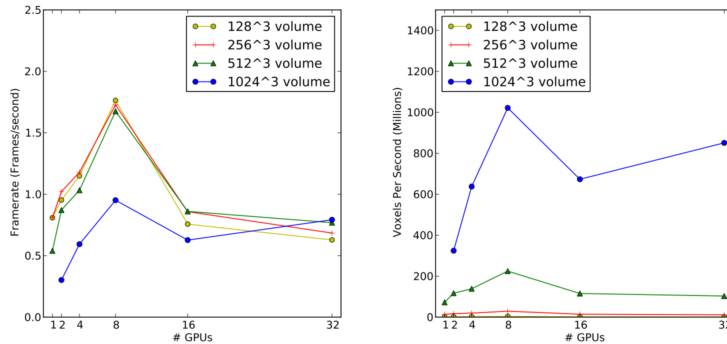

Figure 4: FPS and VPS for various volume sizes.

is no longer a limiting factor in rendering [23]. We do this by showing the "speed-of-light" and the realistic peak speeds for the tasks in the renderer, then showing that we come very close to achieving those.

Reading bricks from disk can take several orders of magnitude more time than the entire MapReduce process. Thus we exclude disk access time from our "speed-of- light" calculations and instead assume that all data is initially resident within CPU system memory. A runtime analysis of the map phase shows that a 1024^3 volume split into two bricks across 8 GPUs requires 515 ms of communication and 503 ms of computation. If we increase this to 16 GPUs, the communication time raises to over 1 second and the computation decreases to 97 ms. We believe this shows that fitting parallel volume rendering into a multi-GPU MapReduce model severely reduces computation as a bottleneck in volume rendering. More GPUs equates to more rendering power, but to fully utilize the power of additional GPUs, it is necessary to have a volume large enough to keep each GPU busy.

7. CONCLUSION

The multi-GPU MapReduce programming model is a viable and high-performance option for volume rendering because it is simple to use, highly pluggable, and allows modular development of parallel volume renderers. Our specific implementation of multi-GPU MapReduce offers scalable performance. A single GPU efficiently renders small volumes in core, and only a minimal number of GPUs is required to efficiently render a volume out of core.

Because we can fit parallel volume rendering into the MapReduce model, we are able to optimize required communication. This allows our library to scale well with data-set size. It also allows more nodes to be used for larger data sets without sacrificing efficiency.

We propose a number of possibilities to achieve better scaling. Some possibilities are research challenges that must be overcome, and some are engineering efforts. To overcome the bottleneck of disk access when only rendering one frame per volume, we believe in-situ visualization is the best choice. This allows the simulation nodes to efficiently split the volume and transfer it over a high-speed interconnect.

With these in mind, we propose several avenues for future work. The first and most obvious is extending our MapReduce implementation by making it more robust, more pluggable, and thus more efficient for any kind of MapReduce task. We feel this would be the largest possible contribution to the community as it would allow commodity GPUs to be added cheaply to large clusters, and hopefully yield a substantial performance gain for many tasks.

We want to explore a few possbilities in volume rendering. One in particular is 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. We also wish to see the results from partitioning the work with respect to the volume instead of the image. This would allow us to load pieces of the volume into the shared memory of each GPU multiprocessor.

We conclude with two more rendering extensions we feel would be worthwhile; a ray-fragment sort on the GPU, and exploring the benefits of direct access for the GPU to system memory (0-copy memory). If the GPUs used for MapReduce are also tied to a display, it may be more efficient to have the final pixels on the GPU as to allow the final pixels to be rendered immediately after compositing is finished. As for 0-copy memory, there is potential for significant overlap of communication with computation because ray fragments could be consumed immediately upon production. There would be no need for transfers between the CPU and GPU. This remains a research topic though because 0-copy memory is orders of magnitude slower than GPU VRAM.

8. ACKNOWLEDGEMENTS

Thanks to our funding agencies, the SciDAC Institute for Ultrascale Visualization and the National Science Foundation (Award 0541448), and to NVIDIA for equipment donations. Also, thanks to Wen-Mei Hu and the NCSA for allowing us use of the Accelerator Cluster.

9. REFERENCES

[1] J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*. Academic Press, Dec. 2004.

[2] B. Catanzaro, N. Sundaram, and K. Keutzer. A Map Reduce framework for programming graphics processors. In *Third Workshop on Software Tools for MultiCore Systems*, Apr. 2008.

[3] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A contract based system for large data visualization. In *VIS '05: Proceedings of the 16th IEEE Visualization Conference*, pages 190–198, Oct. 2005.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, pages 137–150, Dec. 2004.

[5] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. In *Proceedings of the First International Conference on Cloud Computing*, Oct. 2009.

[6] J. Gao, C. Wang, L. Li, and H.-W. Shen. A parallel multiresolution volume rendering algorithm for large data visualization. *Parallel Computing*, 31(2):185–204, Feb. 2005.

[7] L. J. Gosink, J. C. Anderson, E. W. Bethel, and K. I. Joy. Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics*, 14:1715–1722, Nov. 2008.

[8] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. H. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(3):303–312, Sept. 2005.

[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, Oct. 2008.

[10] T.-J. Hsieh, F. Kuester, and T. C. Hutchison. Visualization of large-scale seismic data records. In *Proceedings of the 4th International Conference on Earthquake Geotechnical Engineering (4ICEGE 2007)*, pages 1–12, June 2007.

[11] R. Kaehler, M. Simon, and H.-C. Hege. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):341–351, July– Sept. 2003.

[12] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization Conference*, pages 287–292, Washington, DC, USA, Oct. 2003. IEEE Computer Society.

[13] C. Lam. *Hadoop in Action (Manning Early Access Program)*. Manning Publications Co., 2010.

[14] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[15] D. Luebke. High performance computing with CUDA. *IEEE/ACM Supercomputing* 2009 *Course Notes*, Nov. 2009. `http://www.gpgpu.org/sc2009/`.

[16] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.

[17] K.-L. Ma, A. Stompel, J. Bielak, O. Ghattas, and E. J. Kim. Visualizing very large-scale earthquake simulations. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 48, Nov. 2003.

[18] M. W. Martin, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE Visualization Conference*, pages 333–340, Oct. 2003.

[19] K. Moreland, D. Rogers, J. Greenfield, B. Geveci, P. Marion, A. Neundorf, and K. Eschenberg. Large scale visualization on the Cray XT3 Using ParaView. In *Cray User Group 2008*, May 2008.

[20] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics & Applications*, 14(4):49–58, July/Aug. 1994.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, Mar./Apr. 2008.

[22] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. `http://developer.nvidia.com/cuda`, Jan. 2007.

[23] T. Peterka, H. Yu, R. Ross, and K.-L. Ma. Parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of the Eurographics Parallel Graphics and Visualization Symposium (EGPGV '08)*, pages 73–80, Apr. 2008.

[24] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. CellMR: A framework for supporting MapReduce on asymmetric Cell-based clusters. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Feb. 2007.

[26] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical visualization and compression of large volume datasets using GPU clusters. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '04)*, pages 41–48, June 2004.

[27] J. E. Vollrath, T. Schafhitzel, and T. Ertl. Employing complex GPU data structures for the interactive visualization of adaptive mesh refinement data. In *Proceedings of the International Workshop on Volume Graphics '06*, pages 55–58, July 2006.

[28] C. Wang, J. Gao, L. Li, and H.-W. Shen. A multiresolution volume rendering framework for large-scale time-varying data visualization. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 11–19, June 2005.

[29] H. Yu, K.-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 49, Nov. 2004.

[30] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, page 48, Nov. 2008.