Eurographics Symposium on Parallel Graphics and Visualization (2015)
C. Dachsbacher, P. Navrátil (Editors)

Volume Rendering Via Data-Parallel Primitives

M. Larsen[†1], S. Labasan[1], P. Navrátil[2], J.S. Meredith[3], and H. Childs[1,4]

[1]University of Oregon, [2]Texas Advanced Computing Center, The University of Texas
[3]Oak Ridge National Laboratory, [4]Lawrence Berkeley National Laboratory

Abstract

Supercomputing designs have recently evolved to include architectures beyond the standard CPU. In response, visualization software must be developed in a manner that obviates the need for porting all visualization algorithms to all architectures. Recent research results indicate that building visualization software on a foundation of data-parallel primitives can meet this goal, providing portability over many architectures, and doing it in a performant way. With this work, we introduce an unstructured data volume rendering algorithm which is composed entirely of data-parallel primitives. We compare the algorithm to community standards, and show that the performance we achieve is similar. That is, although our algorithm is hardware-agnostic, we demonstrate that our performance on GPUs is comparable to code that was written for and optimized for the GPU, and our performance on CPUs is comparable to code written for and optimized for the CPU. The main contribution of this work is in realizing the benefits of data-parallel primitives — portable performance, longevity, and programmability — for volume rendering. A secondary contribution is in providing further evidence of the merits of the data-parallel primitives approach itself.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

1. Introduction

Power constraints are forcing supercomputing architects to shift their focus from FLOPs to FLOPs-per-watt. In response, these architects are choosing nodes consisting of many cores per chip and wide vector units, since massive numbers of cores operating at relatively low clock speeds offer the best combination of performance for price and energy. However, there are many hardware architectures to choose from, both those available right now, and possibilities for the future. The top machines in the world currently are composed of technologies like programmable graphics processors (GPUs, e.g., NVIDIA Tesla), many-core co-processors (e.g., Intel Xeon Phi), and large multi-core CPUs (e.g., IBM Power, Intel Xeon). Further, future supercomputing designs may include low-power architectures (e.g., ARM), hybrid designs (e.g., AMD APU), or experimental designs (e.g., FPGA systems).

This variety in hardware architecture is problematic for software developers, as developers do not want to implement distinct solutions for each architecture. This issue is particularly problematic in the context of visualization software, for two reasons. One, visualization software often requires large code bases, with several community standards containing over a million lines of code. Two, visualization software employs many different algorithms; as a result, optimizing performance for one platform requires optimizing each of its algorithms, and not just one “key loop" as is often the case for simulation codes.

Ideally, software developers could write a single implementation that would simultaneously be insulated from architectural specifics and also obtain excellent performance across all desired architectures. This goal is one of the main drivers behind the recent push for domain-specific languages (DSLs) in high-performance computing. In the case of visualization software, three significant efforts — Dax [MAGM11], EAVL [MAPS12], and PISTON [LSA12] — all realized this goal by building a DSL-like infrastruc-

[†] mlarsen@cs.oregon.edu

DOI: 10.2312/pgv.20151155

ture on top of data-parallel primitives. The three efforts have now merged into a single one, called VTK-m, with a goal of providing the same functionality as VTK [SML96], yet with portable performance across multi-core and many-core architectures.

While data-parallel primitives have shown significant promise to date, the downside of the approach is that our community's existing algorithms cannot be simply "ported" into this new framework. In many cases, the algorithms need to be "re-thought" so that they can be composed entirely of data-parallel operations. While some algorithms map naturally, others are more difficult, since isolating out the interdependence of operations — needed so each core on a many-core node can do its own work without interacting with the other cores — is not always trivial.

With this work, we introduce a volume rendering algorithm that is composed entirely of data-parallel primitives. The algorithm is within the ray-casting family, and operates on unstructured mesh data. Further, although the performance study we show focuses on the shared-memory parallelism available on a single node, the algorithm melds naturally into an existing distributed-memory algorithm as well.

The contributions of this work are:

- Description of a new volume rendering algorithm composed of data-parallel primitives;
- Evaluation of the algorithm on CPU and GPU architectures;
- Exploration of the variation in performance characteristics across architectures, which informs how effective data-parallel primitives are at hiding architectural details; and
- Comparison to community standard volume renderers which do not make use of data-parallel primitives.

The paper is organized as follows: Section 2 surveys related work, Section 3 gives a brief overview of data-parallel primitives, Section 4 describes our algorithm, Section 5 outlines our study, and Section 6 explores the resulting performance.

2. Related Work

2.1. Data-Parallel Primitives

The inspiration for much work on data-parallel primitives comes from Blelloch [Ble90]. In his work, Blelloch considered a model where primitives could carry out operations on vectors of size N in time proportional to $O(log_2(N))$ in the worst case. Libraries such as NVIDIA's Thrust [BH11] follow from this idea, and provide a set of efficient primitives. Further, code written to the Thrust interface can be compiled to work with a variety of architectures, including NVIDIA GPUs and x86 architectures.

In an effort to provide portable performance over varied supercomputing architectures, multiple visualization infrastructures were developed that embraced the data-parallel primitives concept:

- Dax [MAGM11], which focused most heavily on the execution model portion of the problem,
- EAVL [MAPS12], which focused most heavily on the data model portion of the problem, and
- PISTON [LSA12], which focused most heavily on developing algorithms.

As mentioned in the introduction, these efforts are now merging into a single product (VTK-m). But the combined product is still short on algorithms — while the foundations have been explored, many specific algorithms are still missing. Our study helps with this problem by contributing an important visualization algorithm.

2.2. Volume Rendering

Volume rendering [Lev88, DCH88] uses a combination of color and transparency to allow users to see the entirety of a three-dimensional volume. The technique starts with a "transfer function," which specifies a mapping of opacity and color for each value in a scalar field. This transfer function is then applied to the entire volume. The resulting images show the color/opacity information, integrated in depth.

2.2.1. Unstructured Volume Rendering

Unstructured grid volume rendering algorithms were surveyed excellently by Silva [SCCB05].

Here, we focus on three arcs of research that were used for comparators in our study:

- One of the first algorithms for volume rendering unstructured meshes, was the "projected tetrahedra" method [ST90]. This method was extended to GPUs in 2002 [WMFC02]. An important advancement to the algorithm came with HAVS (Hardware Assisted Visibility Sorting) [CICS05], which is used as a comparator in this study. HAVS improved the visibility ordering of projected tetrahedra by using the k-buffer, which allowed for compositing of out-of-order pixel fragments.
- Bunyk et al. [BKS97] developed a ray-caster for unstructured data. Their algorithm was implemented in VTK and is still commonly used today.
- Z-Sweep [FMS00] is an algorithm that advances a plane through the volume in depth. Childs et al. [CDM06] developed a parallel algorithm which can be thought of as a descendent of Z-Sweep. The algorithm is based on sampling, and requires a large buffer to hold all the samples. This buffer was divided over many compute nodes, making computation and memory usage tractable. The implementation of Childs' algorithm is available in VisIt [CBW*12].

While our closest relative with respect to volume rendering is likely the distributed-memory algorithm by Childs [CDM06], the closest relative with respect to our research is that which explores volume rendering and shared-memory parallelism. Of these works, the focus is typically on a specific architecture, which contrasts with our hardware-agnostic approach. For example, on the GPU side, there have been many GPU-specific unstructured grid volume rendering algorithms [SCCB05]. On the CPU side, there are fewer shared-memory parallel works. Notable examples include the hybrid-parallel work by Howison et al. [HBC10, HBC12] and the CPU and Xeon Phi work by Knoll et al. [KWN*14]. Expanding the scope beyond volume rendering, Larsen et al. [LMNC15] also considered data-parallel primitives and portable performance, but did it with ray-tracing. While the findings of that paper had similarities in theme, it was looking at a fundamentally different algorithm.

Finally, we note that MapReduce (with Map and Reduce being two of the data-parallel primitives) has been investigated in conjunction with volume rendering [SCMO10]. Our focus is on using these primitives to get excellent single node performance, as opposed to cloud-based usage. Further, we considered a wider range of primitives.

3. Data-Parallel Primitives

Within the data-parallel paradigm, primitives such as map, reduce, gather, and scatter form the basis from which algorithms can be constructed. Dax, EAVL and PISTON implemented a programming model based on a functor-plus-operator approach. A functor is a user-provided struct which defines a function and local data which the data-parallel primitive operator applies in some fashion to the input arrays. Using this approach, libraries abstract away the details such as memory allocation and thread management, leaving the user the task of re-imagining an algorithm using the data-parallel primitives.

Map is the computational work-horse of the data-parallel primitives. Conceptually, Map is the body of a *for* loop, where each iteration of the loop can be executed independently. Without any dependencies on previous executions, each iteration is arbitrarily executed in parallel. A Map operates on any number of input arrays and output arrays, but all must be of the same size.

Gather and **Scatter** copy items in parallel from the input arrays to the output arrays, where the input arrays are of length n and the output arrays are of length m. In Gather, the user specifies a set of indices x of length m, where $out[i] = input[x_i]$ for each i in m, and in Scatter, the user specifies a set of indices x of length n, where $out[x_i] = input[i]$ for each i in n.

Gather is generally preferred when $n > m$ (i.e., the output arrays are shorter than the input arrays), since it is more efficient and cannot result in race conditions. One common use for Gather is downselecting a set of input data to operate on a subset of the full data set. A specialized Scatter operation called **ReverseIndex** can generate the set of indices to be used in subsequent Gather operations when passed a Boolean array of flags specifying whether each input element is included in the set.

Reduce operations combine the input values of an array to a single output value, such as summing all the values in an array or finding the maximum value in an array. In serial code, this is often implemented as a loop with sequential dependencies. However, tree-style reductions can enable efficient parallelism.

Scan, like Reduce, has loop-carried dependencies, but in Scan, the output is an array instead of a single value, and the result at each point in the output array is the partial reduction up to that point. For example, in the “prefix sum" variant of Scan, the output at position i is the sum of all values in the input array up to position i.

In the following section, we describe how we build our volume renderer using these data-parallel primitive operations combined with custom functors.

4. Algorithm

4.1. Algorithm Description

At a high-level, the algorithm is sampling-based, meaning the goal is to populate a buffer of $W \times H \times S$ samples, assuming W and H are the width and height of the image and S is the number of samples in depth. Of course, this buffer size could be very large: for a 1024×1024 image with 1000 samples in depth, the buffer would be approximately 4GB.

To reduce memory requirements, the algorithm can break up the sampling work into passes, evaluating sections of the buffer each pass. When it does this, the algorithm defines the portion of the buffer to operate on at the beginning of the pass. In our implementation, we break up the buffer by depth: when doing multiple passes, we have the first pass evaluate the front portion of the buffer, the second pass evaluate the portion behind it, and so on. This particular strategy enables early ray termination, since we can evaluate which pixels have become opaque at the end of each pass.

Each pass consists of four phases: Pass Selection, Screen Space Transformation, Sampling, and Compositing. When running with two passes, the four phases are executed in sequence (focusing on extracting the close half of the samples) and then executed a second time, again in sequence (focusing on the far half of the samples).

The algorithm also depends on an initialization step. The initialization step and the four phases are described in the following subsections.

4.1.1. Initialization Step

Unlike the four phases, this initialization is executed just a single time, before the passes begin. The goal of this step is to calculate the minimum and maximum depth of each tetrahedron. It does this by applying the camera transform to each tetrahedron, and storing the smallest and largest depths in respective arrays. This is accomplished with a Map primitive.

4.1.2. Pass Selection

The job of this phase is to identify which cells can possibly contribute a sample. If there are N cells, then the first step of this phase is to construct a Boolean array of size N. An element of the array should contain “true" if the corresponding cell can possibly contribute, and “false" otherwise. This is determined by consulting the minimum and maximum depth arrays calculated in the initialization step. This step can again be accomplished with a Map operation.

The second step of this phase is to create an array of just the tetrahedrons that can contribute samples during this pass. This is accomplished by successive use of four data-parallel primitives. First, a Reduce primitive counts the number of active tetrahedrons, i.e., tetrahedrons that have “true" in the Boolean array. Second, an Exclusive Scan primitive calculates the index that each of the active tetrahedrons will be copied into. Third, a Reverse Index primitive uses the result of the Exclusive Scan in order to do the final primitive, a Gather. This Gather collects the indices of the active tetrahedrons into the output array from this phase. If m is the number of active tetrahedrons for a given pass, then the result of this second step is an array of size m.

4.1.3. Screen Space Transformation

This phase uses a Map primitive to transform the active tetrahedrons into screen space using the camera transform. The result of this phase is an array of m tetrahedrons.

4.1.4. Sampling

This phase again uses a Map primitive. The functor for this primitive does the sampling, and outputs the sample values into the buffer. The sampling operation uses the screen-space vertices of each cell to calculate the axis-aligned bounding box (AABB). Using the AABB, the sampler considers every possible pixel and depth that the cell could contribute to, and extracts barycentric coordinates by solving a system of parametric equations defined by the vertices of the tetrahedron.

Finally, a pointer to an array containing the opacity for each pixel is also an input to this phase. With this information, the sampling functor can decide to abort sampling, in a vein similar to early ray termination.

4.1.5. Compositing

This final phase again uses a Map primitive, iterating over groups of samples. This phase again has access to the pixel information. The functor uses the samples and pixel information to composite the color for that pixel, at least with the samples seen so far.

4.2. Data-Parallel Primitives Pseudocode

The pseudocode is specified in Algorithm 1.

```
/*Input*/
array: float tetCoords[$N$*12]
/*Output*/
array: byte pixels[$w * h$]
/*Local Arrays*/
array: byte passRanges[$N$*2] //min pass, max pass
array: bool passFlags[$N$]
array: int currentTets[$M$]
array: int indxScan[$M$]
array: int gatherIndxs[$M$]
array: float screenSpaceTets[$M$*12]
array: float samples[($w * h$) / $numPasses$]
//Initialization
passRanges← map<FindPasses>(tetCoords)
for $pass = 0 < numPasses$ do
    //Pass Selection
    flags← map<Thresh>(passRanges, $pass$)
    $m$ ← reduce<Add>(flags)
    indxScan←scan<Exclusive>(flags)
    gatherIndxs←reverseIndex<>(indxScan,flags)
    currentTets←gather<>(tetIndxs,gatherIndxs)
    //Screen Space Transformation
    screenSpaceTets←map<ScreenSpace>(currentTets,tets)
    //Sampling
    samples←map<Sampler>(screenSpaceTets,pixels)
    //Compositing
    pixels←map<Composite>(samples,pixels)
end
```

Algorithm 1: Pseudocode for our data-parallel primitives-based algorithm. Data-parallel primitives are shown in the form: *primitive<functor>(args)*. N is the total number of tetrahedrons. M is the maximum number of tetrahedrons in a single pass, and m is the actual number of tetrahedrons in the current pass. w and h are the width and height of the image, respectively.

5. Study Overview

5.1. Software Implementation

We implemented our algorithm in EAVL. In this environment, algorithm developers compose a series of data-parallel primitives, augmenting each primitive with functors that perform specialized operations. During compilation, EAVL applies an optimized implementation of each primitive in OpenMP or CUDA as appropriate. In terms of memory layout, we organized our data structures into structs-of-arrays, following acknowledged best practices for both CPU (enabling vectorization) [PM12] and GPU (creating coalesced memory accesses) [CSS11].

EAVL's usage of the memory hierarchy varies by platform. On the CPU side, its usage is conventional: registers, cache, and memory. On the GPU side, however, the memory usage varies based on context. Specifically, while EAVL's built-in operations, such as scan and reduce, make use of the GPU's shared memory, this memory is not exposed to algorithm developers. As a result, algorithms frequently use global memory. This was the case for almost all of our algorithms, with the one exception being our color look-ups, which used texture memory.

5.2. Configurations

Our study consisted of two rounds and 56 total tests. Each test was of a volume rendering that created a 1024×1024 image. Sampling-based volume rendering algorithms used 1000 samples in depth, and the near and far clipping planes were made as close as possible without clipping away data.

5.2.1. Round 1: Evaluation of Data-Parallel Primitives

This round was designed to better understand the basic performance of our volume renderer. It varied three factors:

- Hardware architecture (CPU and GPU): 2 options
- Data set: 4 options
- Camera position (zoomed in and zoomed out): 2 options

We tested on the cross product of these options: $2 \times 4 \times 2 = 16$.

Finally, we also tested multiple transfer functions, but their variation did not significantly impact results; we present results from just a single transfer function from our pool.

5.2.2. Round 2: Comparison With Community Software

This round compared our algorithm to existing standards for unstructured data, specifically the HAVS volume renderer [CICS05] on the GPU, the integration-based ray-caster derived from Bunyk et al. [BKS97] on the CPU, and the sampling-based ray-caster in VisIt [CDM06] on the CPU.

In this round, we ran 24 tests, coming from the cross product of three community standards, four data sets, and two camera positions. Further, to adapt to limitations in the community standards, we changed the CPU platforms we used. As a result, we ran additional CPU tests with our algorithm, for a total of 16 additional tests. (The 8 GPU tests from Round 1 could be re-used in this Round.)

5.3. Testing Options

5.3.1. Hardware Architectures

We used the following architectures:

- **CPU1**: NERSC's Edison machine, where each node contains two Intel "Ivy Bridge" processors, and each processor contains 12 cores, running at 2.4 GHz. Each node contains 64 GB of memory.
- **CPU2**: the same configuration as CPU1, but using only one of the 24 cores.
- **CPU3**: An Intel i7 4770K with 4 hyper-threaded cores (8 virtual cores total) running at 3.5GHz, and with 32 GB of memory.
- **GPU1**: An NVIDIA GTX Titan Black (Kepler architecture) with 2,880 CUDA cores running at 889 MHz, and with 6 GB of memory.

5.3.2. Data Sets and Camera Positions

Our study examined the following four data sets:

- **Enzo-1M**: a cosmology data set from the Enzo [OBB*04] simulation code. This data set was natively on a rectilinear grid, which we then decomposed into tetrahedrons. The total number of tetrahedrons was 1.31 million.
- **Enzo-10M**: a 10.5 million tetrahedron version of **Enzo-1M**.
- **Nek5000**: a 50 million tetrahedron unstructured mesh from a Nek5000 [FLK08] thermal hydraulics simulation. Nek5000's native mesh is unstructured, but composed of hexahedrons; we divided these hexahedrons into tetrahedrons for our study.
- **Enzo-80M**: an 83.9 million tetrahedron version of **Enzo-1M**.

Figure 1 shows volume renderings for these data sets, including the zoomed in and close up camera positions.

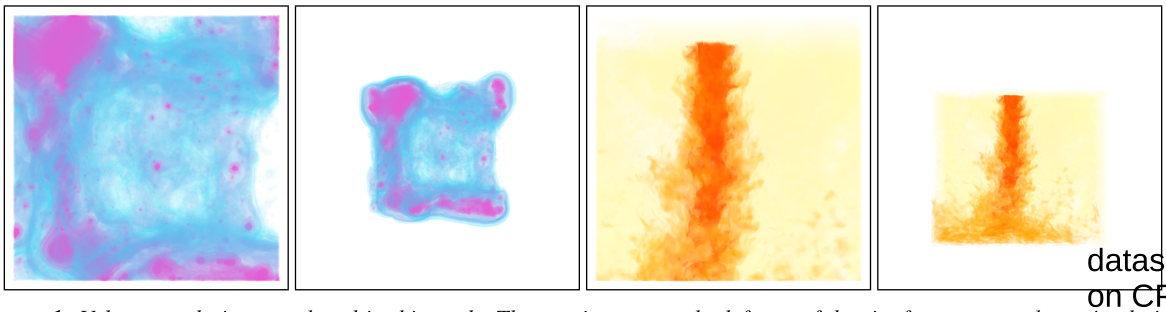
5.4. Performance Measurements

We used the following techniques for performance measurement:

- With our algorithm on **CPU1**, we enabled PAPI [pap12] performance counters to measure the total instructions executed and total cycles during each phase of the algorithm. Specifically, we capture `PAPI_TOT_INS` and `PAPI_TOT_CYC` and use these results to derive instructions executed per cycle.
- With our algorithm on **GPU1**, the nvprof profiler [NVI15] was used to measure total instructions issued, instructions executed, total cycles, registers per thread, and achieved occupancy.
- For each of the community standards, we used that software's built-in timing infrastructure.

6. Results

The results are organized following the two rounds of our study: Section 6.1 details the performance of our algorithm over multiple architectures, and Section 6.2 compares our performance to community standards.



dataset executed on CPU

Figure 1: *Volume renderings produced in this study. The two images on the left are of density from a cosmology simulation. The two images on the right are of temperature from a thermal hydraulics simulation. For each pair of images, the larger one is zoomed in (meaning the data set fills the screen) and the smaller one is zoomed out (meaning the data set is surrounded by white space, which is the default view for many visualization tools).*

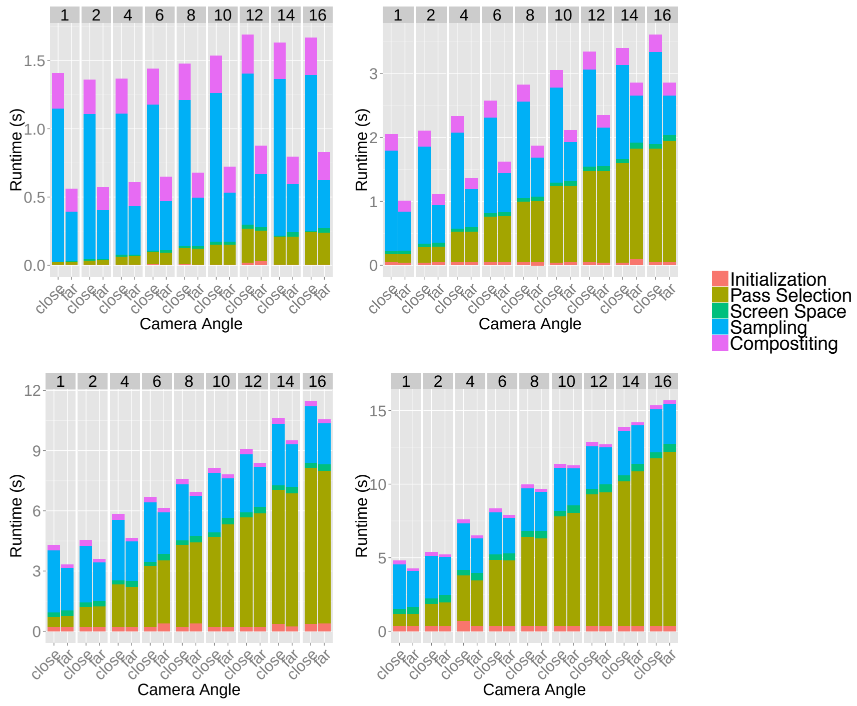


Figure 2: *Running times for our algorithm.* ***Enzo-1M*** *is top left,* ***Enzo-10M*** *is top right,* ***Nek5000*** *is bottom left, and* ***Enzo-80M*** *is bottom right. These tests were run on* ***CPU1*** *and renderings from both camera angles were made. Within a figure, the number of passes increases from left to right.*

6.1. Performance Analysis of Algorithm

6.1.1. CPU Performance

Figure 2 shows the runtime per test by phase. Although our approach requires the evaluation of up to one billion samples (over one million pixels with one thousand samples for each pixel), the algorithm can render an image in approximately one second for small data. As the data size grows, the overall time also goes up, but not in proportion to the data size. This is because the amount of work is proportional to the number of samples, as well as the number of cells. For small data sets, i.e., **Enzo-1M**, the extraction of the samples dominates the overhead for handling each cell. As the data gets larger, though, the handling for each cell dominates. For **Enzo-80M**, the sampling time is nearly the same for both camera positions. This is because the number of samples extracted has nearly doubled, so the majority of the time is

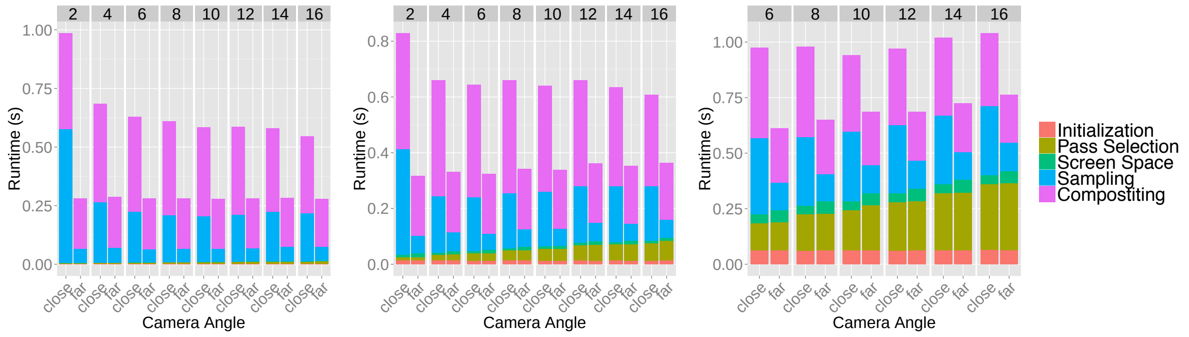


Figure 3: *Running times for our algorithm. From left to right, **Enzo-1M**, **Enzo-10M**, and **Nek5000**. These tests were run on **GPU1** and renderings from both camera angles were made. Within a figure, the number of passes increases from left to right. The **Enzo-80M** run failed, since it was too large for the GPU's 6 GB memory. Further, the **Nek5000** test only has results for 6 passes and above; fewer numbers of passes again ran into the GPU's memory limit.*

being spent iterating over tetrahedrons, as opposed to identifying values at sample locations.

6.1.2. GPU Performance

We present the data in two ways. Figure 3 shows the runtime per test by phase, and Table 1 summarizes kernel register usage and achieved occupancy.

While the dominant factor for CPU performance is sampling time, the dominant factor for GPU performance is compositing time. Even though the compositing kernel uses a lower number of registers per thread and has a higher achieved occupancy, data access patterns and the small number of operations needed to perform compositing make this step a bottleneck.

Kernel	Time	Registers	Occupancy
Screen Space	0.008s	70	38%
Sampling	0.202s	57	47%
Compositing	0.416s	37	68%

Table 1: *Elapsed time, registers per thread, and achieved occupancy for a close up view of the **Enzo-10M** data set with four passes on **GPU1**. The statistics for pass selection were omitted since they were difficult to extract; this phase makes use of multiple data-parallel primitives, which in turn each use multiple CUDA kernels.*

6.1.3. Assessing Performance Portability

The main draw of the data-parallel primitive approach is portable performance. Since we have analyzed the performance on both a CPU and a GPU, we can investigate whether we are achieving this goal. Table 2 shows measurements on the CPU and GPU for the **Enzo-10M** data set with a close up view, using four passes.

Phases	GPU		CPU	
	Time	IPC	Time	IPC
Pass Selection	0.018	1.628	0.514	0.268
Screen Space	0.008	1.704	0.047	0.682
Sampling	0.202	2.477	1.495	1.125
Compositing	0.416	0.131	0.249	1.071

Table 2: *Measurements of CPU and GPU performance, by phase for a close up view of the **Enzo-10M** data set with four passes. The measurements are of time (in seconds) and of instructions executed per cycle (denoted IPC) per core.*

The achieved performance on the architectures at different phases has some interesting results. The instructions per cycle (IPC) indicates how data-intensive the computation is. If a core cannot issue any instructions because it is waiting on data to return from cache or memory, then the IPC will drop. Alternatively, if there are significant computations per load, then the IPC will be high, since the data loads are amortized.

Intuitively, pass selection should have a low IPC value, since it involves iterating through an array of data and performing few computations on them, and the CPU indeed has a low IPC value for this phase. But the GPU has a high IPC value. This is because the data-parallel operations map to built-in constructs on the GPU that are specifically optimized to perform this job, i.e., coalescing memory accesses of large arrays quickly.

The screen space and sampling phases both have high IPC values. This matches our intuition that these phases are compute-bound, and that data movement is not the determining factor in performance. Further the elapsed time is consistent: the GPU has significantly more FLOPs, and so the GPU is much faster.

The compositing phase is the most noteworthy. Where the

GPU benefited from built-in support for coalesced memory accesses in the pass selection phase, it is not receiving that benefit here. Our implementation organizes the data so that the samples for a given ray are spread out over memory. We suspect this choice of data organization is leading to poor performance, since each thread within a warp on the GPU competes with the others to get the data they need. As a result, the IPC on the GPU is very low, and the phase is 50% slower on the GPU than it is on the CPU.

Summarizing, our tests show the CPU behaving as expected, and the GPU doing well on compute-based activities. However, the data-intensive activities can be well accelerated (pass selection) or not (compositing) based on the specifics of the usage. So we see the evidence that portable performance from data-parallel primitives is mostly effective, but there are still pitfalls.

6.1.4. Scalability

Table 3 shows the scalability of the algorithm on **CPU1**. While adding threads does lead to a dropoff of 50% up to 24 threads (the number of CPU cores on the node), the algorithm appears to scale generally well overall.

Threads	1	2	4	8	16	24
Raw Time	43.9	24.2	12.9	7.1	3.8	2.5
Total Time	43.9	48.4	51.7	57.0	60.2	60.7

Table 3: *Times, in seconds, of a strong scaling study. The experiments were run on **CPU1**, using the **Enzo-10M** data set with the close up view and one pass. The times are reported as "total time", meaning the raw time to render the image multiplied by the number of threads. With this measurement, perfect scaling gives a fixed total time over all threads, while poor scaling leads to increases.*

6.2. Comparisons With Community Software

6.2.1. HAVS

First, we reiterate that HAVS is a projected tetrahedron algorithm and ours is a ray-casting algorithm. As a result, the pictures produced will be different (but similar), and the fundamental performance bottlenecks will be different.

HAVS first sorts geometry, and then rasterizes that geometry to the screen. The VTK implementation of HAVS does the sorting step on the CPU, and then transfers the sorted geometry to the GPU for rendering. We felt comparing against this implementation would be improperly handicapping the projected tetrahedron implementation. Instead, we evaluated a parallel radix sort on **GPU1**, measuring the sorting time for different data sizes. The data presented here for HAVS is then the time for our radix sort and the rendering time in HAVS. Explicitly, the CPU sorting time in HAVS has been replaced in these results by our GPU radix sort times.

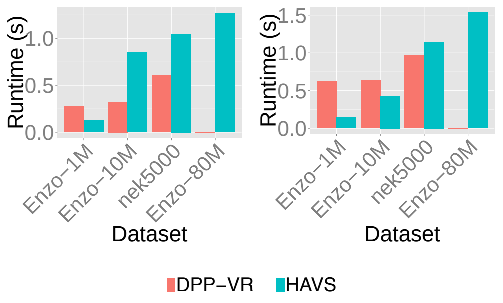


Figure 4: *Comparing the running times for our algorithm and **HAVS** on **GPU1** for multiple data sets. The left figure is for a zoomed out view, and the right figure is for a close up view.*

Figure 4 shows the results from our study. Our algorithm is mostly slower than HAVS when zoomed in (because so many samples need to be evaluated), but faster than HAVS when zoomed out (because there are fewer samples evaluated). We note that the HAVS running times were highly correlated to data size, and that our algorithm did not slow down as quickly as HAVS when data size increased. That said, HAVS was able to complete on all data sets, because it did not need additional memory to store samples.

6.2.2. Unstructured Ray-Caster

Our next comparison was on the CPU to the unstructured ray-caster implemented in VTK in the style outlined by Bunyk et al. [BKS97]. We intended to run this study on Edison, but their implementation exhibited poor scaling properties, and we felt the comparison was unfair. So we switched to the **CPU3** architecture, where their implementation performed better.

This algorithm has a pre-processing step to trace face connectivity. This step is implemented in serial and took over 50 minutes in the case of the **Enzo-80M** data set. The timings for this pre-processing step are omitted in our results.

Figure 5 shows the results from our study. As a trend, we were faster on larger data sets, and results were mixed on smaller data sets. Overall, we concluded that our algorithm performs comparably to the integration-based ray-caster.

6.2.3. VisIt

VisIt's volume rendering algorithm is also sampling-based, although it extracts samples by "rasterizing" geometry, i.e., by transforming cells into image space, then slicing them by planes that are aligned with columns of pixels, and then extracting lines along those planes in depth [CDM06]. The VisIt volume renderer is designed for distributed-memory parallelism; after sampling, it redistributes the samples to do compositing. That said, it is not threaded or ported to

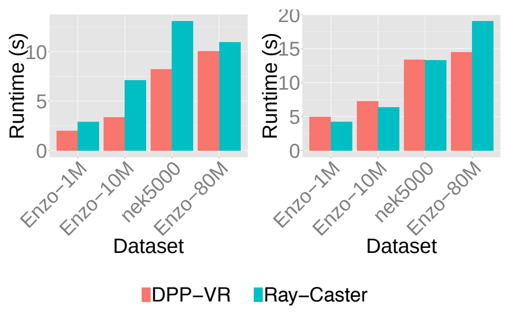


Figure 5: *Comparing the running times for our algorithm to a* ***sample-based*** *on* ***CPU3*** *for multiple data sets. The left figure is for a zoomed out view, and the right figure is for a close up view.*

the GPU. As a result, we performed this comparison using the **CPU2** hardware configuration, namely NERSC's Edison machine, but limited to one core. Explicitly, we ran VisIt in serial, and we ran our algorithm using only one core, meaning both implementations had access to the same hardware. The results of these runs are in Table 4.

Data & View	SW	SS	S	C	TOT
E-1M/Far	VisIt	0.47	12.9	2.34	15.7
E-1M/Far	DPP-VR	0.17	8.4	2.60	11.5
E-1M/Close	VisIt	0.47	23.5	5.35	29.4
E-1M/Close	DPP-VR	0.13	24.9	5.88	31.1
E-10M/Far	VisIt	3.94	48.3	0.81	53.0
E-10M/Far	DPP-VR	1.41	13.4	2.60	19.2
E-10M/Close	VisIt	4.03	51.8	1.74	57.5
E-10M/Close	DPP-VR	1.06	35.3	5.88	43.9
N-50M/Far	VisIt	24.7	355.5	0.58	391
N-50M/Far	DPP-VR	6.93	24.3	2.92	42.8
N-50M/Close	VisIt	24.8	395	1.02	421
N-50M/Close	DPP-VR	4.93	49.7	5.88	68.7
E-80M/Far	VisIt	33.6	351	0.31	385
E-80M/Far	DPP-VR	11.4	27.6	2.60	55.7
E-80M/Close	VisIt	33.6	361	0.62	396
E-80M/Close	DPP-VR	8.62	55.9	5.88	84.1

Table 4: *Time to volume render a single frame, in seconds.* ***SW*** *indicates the software used, either VisIt, or our data-parallel primitives algorithm (DPP-VR).* ***SS*** *denotes the screen space transformation time (expensive since it is done repeatedly in a multi-pass setting),* ***S*** *denotes the sampling time,* ***C*** *denotes the compositing time, and* ***TOT*** *denotes the total time to make the image.*

The VisIt algorithm and our algorithm are the most closely related of the three we studied, and performance between the two is similar. One difference is that VisIt uses a three-dimensional rasterization algorithm and our algorithm considers the samples within the tetrahedron's bounding box and does an inside-outside test. VisIt's approach is beneficial with large cells (i.e., **Enzo-1M**), since it is amortizing its calculations. But our approach is faster with small cells (i.e., **Enzo-80M**), since the overhead VisIt pays per cell is no longer amortized away. Another difference is that our algorithm ran with only a single pass, meaning that no early ray termination was used. VisIt did use an early ray termination criteria, leading to lower compositing times.

On the whole, however, this comparison shows that the data-parallel approach leads to better performance to an algorithm that was optimized for one platform (in this case, the CPU).

7. Conclusion

We presented a new algorithm for unstructured volume rendering that was composed entirely of data-parallel primitives. The algorithm performed comparably to community standards on the GPU and CPU. Moreover, because the algorithm used data-parallel primitives, the real advantages are benefits in portable performance, longevity, and programmability. Each new algorithm re-thought in terms of data-parallel primitives, including this one, enables the advancement of data-parallel primitive-based infrastructures that can run on multiple architectures. Finally, we investigated whether the promise of portable performance is being achieved, and found that it mostly was, although some data-intensive patterns can lead the GPU to perform poorly relative to its potential.

In terms of future work, we plan to extend the algorithm for distributed-memory parallelism. Our work melds well with the distributed-memory algorithm by Childs et al. [CDM06]; the natural extension is to replace its sampling and compositing phases with our data-parallel primitive approach. We believe this will lead to excellent performance, including frame rates significantly below those observed in this work, since the samples will be divided over the nodes, and thus the computational workload per node will be reduced.

Acknowledgments

This work was funded in part by US National Science Foundation grants ACI-1339840 and ACI-1339863. Hank Childs is grateful for support from the DOE Early Career Award, Contract No. DE-FG02-13ER26150, Program Manager Lucy Nowell. Some work was performed by Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. DOE Department of Energy under Contract No. DE-AC05-00OR22725 and funded in part by the Advanced Scientific Computing Research (ASCR) program. This study used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[BH11] Bell N., Hoberock J.: Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems*, Hwu W.-M., (Ed.). Elsevier/Morgan Kaufmann, 2011, pp. 359–371. 2

[BKS97] Bunyk P., Kaufman A., Silva C. T.: Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization Conference, 1997* (1997), IEEE, pp. 30–36. 2, 5, 8

[Ble90] Blelloch G. E.: *Vector Models for Data-Parallel Computing*. MIT Press, 1990. 2

[CBW*12] Childs H., Brugger E., Whitlock B., Meredith J., Ahern S., Pugmire D., Biagas K., Miller M., Harrison C., Weber G. H., Krishnan H., Fogal T., Sanderson A., Garth C., Bethel E. W., Camp D., Rübel O., Durant M., Favre J. M., Navrátil P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Oct. 2012, pp. 357–372. 2

[CDM06] Childs H., Duchaineau M., Ma K.-L.: A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (Braga, Portugal, May 2006), pp. 153–162. 2, 3, 5, 8, 9

[CICS05] Callahan S. P., Ikits M., Comba J. L. D., Silva C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *Visualization and Computer Graphics, IEEE Transactions on 11*, 3 (2005), 285–295. 2, 5

[CSS11] Che S., Sheaffer J. W., Skadron K.: Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 13. 5

[DCH88] Drebin R. A., Carpenter L., Hanrahan P.: Volume rendering. *SIGGRAPH Computer Graphics 22*, 4 (1988), 65–74. doi:http://doi.acm.org/10.1145/378456.378484. 2

[FLK08] Fischer P. F., Lottes J. W., Kerkemeier S. G.: nek5000 Web page, 2008. http://nek5000.mcs.anl.gov. 5

[FMS00] Farias R., Mitchell J. S., Silva C. T.: Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of the 2000 IEEE symposium on Volume visualization* (2000), ACM, pp. 91–99. 2

[HBC10] Howison M., Bethel E. W., Childs H.: MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (Norrköping, Sweden, Apr. 2010), pp. 1–10. 3

[HBC12] Howison M., Bethel E. W., Childs H.: Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 18*, 1 (Jan. 2012), 17–29. doi:http://doi.ieeecomputersociety.org/10.1109/TVCG.2011.24. 3

[KWN*14] Knoll A., Wald I., Navratil P., Bowen A., Reda K., Papka M. E., Gaither K.: Rbf volume ray casting on multicore and manycore cpus. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 71–80. 3

[Lev88] Levoy M.: Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications 8*, 3 (May 1988), 29–37. 2

[LMNC15] Larsen M., Meredith J., Navrátil P., Childs H.: Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium* (Hangzhou, China, Apr. 2015). To appear. 3

[LSA12] Lo L.-T., Sewell C., Ahrens J.: PISTON: A portable cross-platform framework for data-parallel visualization operators. Eurographics Symposium on Parallel Graphics and Visualization, pp. 11–20. 1, 2

[MAGM11] Moreland K., Ayachit U., Geveci B., Ma K.-L.: Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization* (October 2011), pp. 97–104. 1, 2

[MAPS12] Meredith J. S., Ahern S., Pugmire D., Sisneros R.: Eavl: the extreme-scale analysis and visualization library. 1, 2

[NVI15] NVIDIA: CUDA Profiler Web page, 2015. http://docs.nvidia.com/cuda/profiler-users-guide. 5

[OBB*04] O'Shea B. W., Bryan G., Bordner J., Norman M. L., Abel T., Harkness R., Kritsuk A.: Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints* (Mar. 2004). arXiv:astro-ph/0403044. 5

[pap12] Performance application programming interface, 2012. URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. 5

[PM12] Pharr M., Mark W. R.: ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012* (2012), IEEE, pp. 1–13. 5

[SCCB05] Silva C. T., Comba J. L. D., Callahan S. P., Bernardon F. F.: A survey of gpu-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA) 12*, 2 (2005), 9–29. 2, 3

[SCMO10] Stuart J. A., Chen C.-K., Ma K.-L., Owens J. D.: Multi-gpu volume rendering using mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), ACM, pp. 841–848. 3

[SML96] Schroeder W. J., Martin K. M., Lorensen W. E.: The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), IEEE Computer Society Press, pp. 93–ff. 2

[ST90] Shirley P., Tuchman A.: *A polygonal approximation to direct scalar volume rendering*, vol. 24. ACM, 1990. 2

[WMFC02] Wylie B., Moreland K., Fisk L. A., Crossno P.: Tetrahedral projection using vertex shaders. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics* (2002), IEEE Press, pp. 7–12. 2