Ray Tracing Within a Data Parallel Framework

Matthew Larsen[1]*, Jeremy S. Meredith[2], Paul A. Navrátil[3], and Hank Childs[1,4]
[1]University of Oregon
[2]Oak Ridge National Laboratory
[3]Texas Advanced Computing Center, The University of Texas at Austin
[4]Lawrence Berkeley National Laboratory

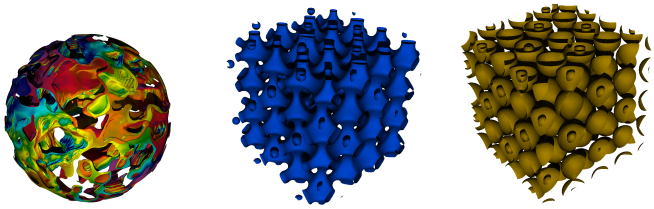

Figure 1: Renderings of three of the scientific data sets used in the performance study. The images were generated in 43 ms on an NVIDIA Titan Black, using a ray tracer consisting entirely of data parallel primitives.

ABSTRACT

Current architectural trends on supercomputers have dramatic increases in the number of cores and available computational power per die, but this power is increasingly difficult for programmers to harness effectively. High-level language constructs can simplify programming many-core devices, but this ease comes with a potential loss of processing power, particularly for cross-platform constructs. Recently, scientific visualization packages have embraced language constructs centering around data parallelism, with familiar operators such as map, reduce, gather, and scatter. Complete adoption of data parallelism will require that central visualization algorithms be revisited, and expressed in this new paradigm while preserving both functionality and performance. This investment has a large potential payoff: portable performance in software bases that can span over the many architectures that scientific visualization applications run on.

With this work, we present a method for ray tracing consisting of entirely of data parallel primitives. Given the extreme computational power on nodes now prevalent on supercomputers, we believe that ray tracing can supplant rasterization as the work-horse graphics solution for scientific visualization. Our ray tracing method is relatively efficient, and we describe its performance with a series of tests, and also compare to leading-edge ray tracers that are optimized for specific platforms. We find that our data parallel approach leads to results that are acceptable for many scientific visualization use cases, with the key benefit of providing a single code base that can run on many architectures.

1 INTRODUCTION

Supercomputing system design emphasis has moved from raw performance to performance-per-watt, and as a result, supercomputing architectures are becoming increasingly varied. While the prevalent architectures all feature wide vector units and many processing cores per chip, the specific architectures are manifold, including: programmable graphics processors (GPUs, e.g., NVIDIA Tesla); many-core co-processors (e.g., Intel Xeon Phi); large multi-core CPUs (e.g., IBM Power, Intel Xeon); low-power architectures (e.g., ARM); hybrid designs (e.g., AMD APU); and experimental designs (e.g., FPGA systems).

This diversity of hardware architectures is increasingly problematic for software developers. Developers are typically not eager to devote time to porting their software to each platform, and then further to optimizing their software's performance on each platform. One solution to this problem is Domain Specific Languages (DSLs). With DSLs, developers write their algorithm one time, and then the burden of supporting many architectures and achieving good performance is shifted to the developers of the DSL. Assuming that many projects make use of a DSL, this shift of burden makes sense, because DSL development team can do the "hard work" once and have wide benefit. Further, DSLs offer developers the possibility of "future-proofing," since the DSL development team will, ideally, port efficiently to new architectures as they arise. Of course, DSLs can only succeed if they pick abstractions that simultaneously create enough flexibility for the software developers who use them and also map to well-performing routines on the underlying architectures.

Guy Blelloch's book, *Vector Models for Data-Parallel Computing* [8], described a computing paradigm where a fixed set of primitives — map, scan, reduce, gather, etc. — operated on vectors of data. The model only included primitives that could be achieved in time proportional to the logarithm of the size of the input vectors. Blelloch's book argued that vector models and data parallelism were at the nexus of programming languages, algorithms, and architecture. Twenty years later, the concept of data-parallel primitives has only grown in popularity. Coming back to DSLs, the NVIDIA Thrust library is a DSL that provides programmers with a set of data-parallel primitives. Thrust natively can generate CUDA or x86 code, and extensions for the Xeon Phi are possible.

With the rise of *in situ* processing, visualization and analysis routines must be run on the same architectures as the simulation code. Further, they must fit within the constraints of the simulation code — execution time, memory usage, network traffic, energy expenditure — and so they must be efficient. To meet this need, a number of new visualization libraries are emerging, including DAX [27], EAVL [26], and PISTON [25]. These libraries are striving to become hubs of software development, with large numbers of visualization algorithms, and, at the same time, offer portable performance over varied architectures. All three do this by exposing data-

*mlarsen@cs.uoregon.edu

parallel primitives to software developers and asking them to rethink algorithms in this new environment. Each of these packages could have been an excellent foundation for a ray tracer; we chose EAVL because we were most familiar with development within that framework.

With this paper, we explore ray tracing within the context of a data-parallel framework. The contributions of this work are:

- A description of a ray tracing algorithm, novel because it is composed entirely of data-parallel primitives in a manner that enables portable performance on many-core nodes;
- Analysis of how this algorithm performs over various platforms;
- A comparison of our data parallel ray tracer with leading-edge platform-specific ray tracers, in an effort to better understand the performance gap that comes from implementing algorithms within a data parallel framework; and
- Additional documentation that ray tracing is viable alternative to rasterization for scientific visualization use cases.

2 Related Work

2.1 Data Parallelism and Visualization

Data parallelism has been advancing at two somewhat distinct levels. Large-scale data parallelism, particularly in the supercomputing arena, commonly involves entire processes executing the same program with decomposed chunks, communicating across a distributed network as necessary through the Message Passing Interface [18, 19]. For scientific visualization, this approach has been embraced by tools such as VisIt [13] and ParaView [5], and has seen usage ranging from tens of thousands of processing elements for production runs [14] to hundreds of thousands of processing elements for research experiments [20]. Fine grain data parallelism has occurred at both coarse grain (threads) and fine grain (vector processors and SIMD instructions), and current graphics processors exhibit both of these techniques with thousands of concurrent threads. Programming languages and APIs which expose these types fine-grain parallelism explicitly include CUDA [30], OpenCL [22], Cilk Plus [15], and Threading Building Blocks [37].

The parallel algorithm primitives described by Blelloch [8] have led to higher-productivity application programming interfaces. Through compile-time expansion of combined data-parallel patterns and user-defined function objects (functors), the compiler can heavily optimize the resulting code, resulting in flexible programmability and high performance across multi-core and many-core devices. One of the most notable libraries to use this technique is Thrust [6]. Domain knowledge can extend these techniques; for example, domain specific languages like Liszt [16] for scientific solvers and APIs like EAVL [26] for visualization and analysis provide one more level of productivity beyond domain-independent collections of parallel primitives. See Section 3 for more information on these primitives and their use.

In terms of re-thinking visualization algorithms in terms of node-level fine-grained data parallel primitives, the work in DAX, EAVL, and PISTON represent the community's latest advancements. Further, interest in ray tracing for large-scale scientific visualization has increased recently [11, 12, 28], and its reputation as an "embarrassingly parallel" algorithm make it a prime candidate to explore the usefulness of data parallel primitives.

In terms of previous work, Schroeder et al. [35] explored the generation of acceleration data structures for ray tracing two decades ago when data parallel primitives were first a hot topic; our own work contrasts with theirs, in that we are considering the ray tracing component (which is complementary to their acceleration data structures), and in that we present extensive performance results on modern architectures. BnsView [23] demonstrated a ray tracing system with portable performance over CPUs via language-based abstractions that hide architecutural details; our work contrasts with theirs in our usage of data-parallel primitives, and our support for additional architectures (specifically GPUs). Finally, Northam et al. [29] demonstrated a MapReduce-based ray tracing system for the cloud. While both systems share operators such as "map" and "reduce," their usage is as a mechanism for programming to a distributed-memory (cloud) machine, while ours are aimed at achieving portable performance on vector machines.

2.2 SIMD Ray Tracing

During a ray tracer's execution, sibling rays often start at similar points and travel in similar directions (e.g., camera rays, shadow rays). This spatial coherence can be exploited through the use of vectorized SIMD instructions to reduce the total number of instructions required for both ray traversal and ray intersection [39]. This fine-grained parallelism has formed the fundamental building block of many modern high-performance ray tracers (e.g., [3, 7, 31, 34, 39, 40]), including vendor-specific implementations for multi-core and many-core architectures. We discuss implementations from NVIDIA and Intel in greater detail below.

2.2.1 NVIDIA OptiX

On GPUs, NVIDIA's OptiX Prime™ leads the way with a set of high and low level APIs that supports a rich feature set, including scene graph management and a variety of acceleration structures to meet the needs of static and dynamic scenes. The newly introduced OptiX Prime in Optix V3.5 replaces the previous rtuTraversal API that has been available in the API since version 2.1. The OptiX Prime kernels are based on the hand-tuned kernels by Aila et al. [3, 4], and they leverage features of the recent Kepler GPU architecture for improved per-ray performance.

In order to maximize SIMD efficiency on the GPU, intra-warp thread polling is used to coordinate the behavior of threads, which either all traverse inner nodes of the BVH or intersect with the triangles contained within the leaves. Using this type of coordinated action, SIMD efficiency is increased by reducing the number of predicated instructions, thus improving ray throughput. The OptiX code was also organized to reduce the number of instructions generated by the compiler, and, in some cases, in-line assembly was used. Since the OptiX kernels are not public, it is unknown if this is included in the release, but seems likely. Another source of inefficiency on the GPU is dead rays within a warp. Since a warp terminates when all of the threads within it have completed, utilization decreases as more rays terminate. Alia and Laine [3] proposed the use of dynamically fetching work. Dynamic fetching of rays increases the SIMD efficiency on the Kepler architecture by replacing dead rays if the number of dead rays reaches some specified value [4], but is not beneficial on all architectures.

2.2.2 Intel Embree

On the CPU, Embree is a low-level API developed by Intel Labs that contains a number of public ray tracing kernels optimized for CPUs and the Intel Xeon Phi co-processor. These kernels leverage CPU support for the vector instruction sets SSE and AVX, and are "hand optimized" to further improve performance [40]. Using vector instruction allows for the efficient traversal of a BVH with branching factors that match the width of the SIMD lanes [38]. The axis-aligned bounding boxes (AABBs) of the child nodes are fetched and tested within the CPU vector units, and, using the same technique, a group of triangles can be tested against a single ray at once. Finally, the code is highly optimized, including usage of intrinsics, compiler hints, and goto statements.

3 Overview of Data Parallelism

The data parallel primitives used in the ray tracer consist of map, gather, scatter, reduce, and scan. In combination with user defined

functors, these primitives are combined to produce algorithms that execute on arbitrary architectures without the knowledge of their underlying details. Though some understanding of the architectures — such as the memory hierarchy and available resources — could be beneficial to performance, it is not included in the data parallel model. Memory transfers, in the context of a GPU, are also abstracted away, freeing the user of the responsibility for copying arrays from the host to device and vice-versa.

3.1 Map

In its simplest form, the map operation performs a single function on every element of an array and outputs an equally sized array containing the transformed elements. In more complex forms, map can have multiple input and output arrays, provided all arrays have the same size.

Usage Within Ray-Tracing Algorithm: The map operation is used in many locations in the ray tracer. Primary ray generation is a map operation, with the input array being the index of each ray, and the functor initialized with the screen dimensions and camera vectors. The functor then computes the ray direction and stores it in an output array containing the X, Y, and Z components for each ray. Other usages of the map operation include ray intersection and color accumulation functions.

3.2 Gather and Scatter

Gather and scatter copy a set of items from the input arrays to the output arrays, potentially performing some operation on the copied value. Here, the inputs and outputs may differ in length, as the mapping is determined by an array of indices passed by the caller. In gather, the index array is the same length as the output arrays, and the index specified for each output value specifies which input element to read from. In scatter, the index array is the same length as the input arrays, and the index specified for each input value specifies the location into which to write it. Since it is possible for scatter to write to the same location from multiple threads, the operation can result in race conditions, so use of scatter generally requires more care than gather. Gather can also be faster, particularly when going from a longer array to a shorter one.

Usage Within Ray-Tracing Algorithm: Gather is used to collect color values from all rays contributing to the same pixel to perform anti-aliasing, and it is used to remove dead rays and compact remaining rays into a smaller memory footprint. Scatter is used within the ray tracer to perform ambient occlusion. Specifically, we transform ray intersection points into directionally random rays about the hemisphere defined by the point normal, and then scatter them into an array n times larger than the input array (where n is the number of samples per intersect point). The ray tracer also uses a scatter operation to expand previously compacted arrays back to their original size. For example, the ray tracer scatters color values out to the frame buffer, so that a map operation can accumulate their contribution to the final image.

3.3 Reduce

A reduce operation combines all input values in some way to generate a single output value. Although user-defined functors are permitted (as long as they obey the associative property), reduction is most commonly performed using pre-defined functors, such as addition (to sum the values in the array) or minimum and maximum (to find extrema within an array).

Usage Within Ray-Tracing Algorithm: The reduce operation is used within the stream compaction step to count the number of output elements, so new arrays may be allocated. The operation can also be used to quickly calculate the AABBs of large groups of geometric primitives [24].

3.4 Scan

The scan operator is similar to reduce. It make use of a binary operator, but instead of producing in a single value like reduce, scan produces an output array of the same length as the input array, where each element is the result of the reduction *up until that location in the array*. For example, the result of scan using the addition operator is called a prefix sum [8], where the element at position i is the result of the partial sum of elements up to position i. (The *inclusive* variant includes element i in this partial sum; *exclusive* does not.) A variant of scan called *segmented scan* performs the scan within only partitioned sections of the array, and is useful to implement steps of complex algorithms like parallel quicksort.

Usage Within Ray-Tracing Algorithm: In our ray tracing algorithm, scan is only used to assist with stream compaction. However, scan is very useful, and may be used more in the future. Methods for sorting rays into coherent frustrums for better SIMD efficiency have already been implemented using the scan and other data parallel primitives [17, 31, 40]. Furthermore, scan can be used to build histograms, conduct binary searches, construct k-D trees, and many more [8].

4 Algorithm Description

In this section, we describe our ray tracing algorithm based on the data parallel primitives described in the previous section. We implement a modified breadth-first ray tracer that processes rays by type within each ray generation, similar in spirit to Boulos et al. [10] though without ray sorting. Rays are processed via a pipeline model that maps well onto the data parallel primitive substrate. We present these pipeline stages below. Note that we add image contributions to the framebuffer as they are generated at each stage using the additive formulation of the rendering equation described by Pharr et al. [32].

Finally, note that this ray-tracing algorithm is for a single node, and it assumes access to all mesh data. In a parallel setting, we would assume that data is decomposed over nodes, and that the ray-tracing algorithm would be run on each node. We envision two mechanisms for this parallelism: (i) OpenGL/rasterization-style rendering where there are no secondary rays, and so the produced images can be used as input to a distributed-memory compositer or (ii) extensions to this algorithm where secondary rays are passed between nodes (i.e., as in [28]).

4.1 Initial Ray Generation

Camera rays are generated in one pass according to a given camera and lens model and stored in a contiguous array for processing. For our experiments described in Section 6, we use a pinhole camera with rays ordered by a Morton-curve traversal of the framebuffer. Our algorithm supports both single-ray pixel sampling and four-ray pixel super-sampling.

4.2 Traversal and Intersection

Within each generation of rays, the rays are separated by type (shadow, reflection, ambient occlusion, etc), and each type is processed in a separate pipeline invocation. For each invocation, the specific distribution of work across the hardware is determined by the data parallel primitive engine. The traversal and intersection of each ray follows the `if-if` algorithm described by Aila and Laine [3], with minor modifications to operate within the data parallel primitive framework. For each ray that intersects some geometry, the geometry index and intersection point are stored for use in later pipeline stages. Rays that fail to intersect any geometry are flagged to prevent further processing.

4.3 Specular Reflections

If the material of the intersected geometry contains a specular component, our algorithm generates a reflection ray. All generated reflection rays are stored in a contiguous array and processed together, and the current ray generation information is pushed on a stack for use by later pipeline stages. Any resulting intersections are processed recursively until no more specular rays are generated, either

because there are no intersections or because the remaining rays have reached the bounce limit.

4.4 Stream Compaction

After the specular reflection check above, our algorithm can perform an optional stream compaction step to coalesce the active rays together in memory. This compaction can improve the amount of useful work performed when there are a large number of inactive rays that would otherwise be processed but have their results masked away. Compaction is useful both for scenes where a large number of primary rays fail to intersect with geometry and for long-lived, divergent secondary rays, like those used in Monte-Carlo global illumination estimation, where the time spent compacting the arrays is amortized over the computation savings from culling a large number of dead rays (see, e.g., [10]).

4.5 Ambient Occlusion

For each successful intersection, our algorithm performs an ambient occlusion pass by casting a user-defined number of random hemispheric sample rays around each intersection point (our experiments use a default of four samples). All sample rays within a generation are scattered to a single contiguous array and are processed together. While the random sample rays typically diverge from their origin, we keep the sampling distance sufficiently short so that the limited distance that these rays travel within the scene keep data accesses coherent around each intersection point.

4.6 Shadows

For each successful non-shadow intersection, our algorithm also tests visibility between each intersection point and the point light sources defined in the scene. For each visibility query that succeeds (i.e., the ray misses all potentially intervening geometry), the algorithm performs Blinn-Phong shading [9] and adds the resulting color contribution to the appropriate pixel in the framebuffer.

4.7 Pseudocode

```
01 rays:=rayGenerationMap(eye) //initial rays
02 rayDepth=0
03 while rayDepth<maxDepth
04   hits:=map<intersect>(rays)
05   (reflectedRays,intersects):=
                map<reflect>(hits)
06   rays:=reflectedRays
08   if(compactOn) compactArrays()
09
10   //ambient occlusion
11   occRays:=scatter<occRayGen>
              (intersects,normals,numSamples)
12   hits:=map<intersect>(occRays,maxDistance)
13   occlusion:=gather(hits,numSamples)
14   shadowHits:=map<intersect>(intersects,lights)
15   //generate and accumulate rgb values
16   rgb:=map<shader>
              (intersects, occlusion,shadowHits)
17   colorBuffer:=map<accum>(rgb,colorBuffer)
18   rayDepth++
19  //end while
20 screenBuffer:=gather<antiAlias>(colorBuffer)
21
22 compactArrays()
23   thresh:=map<thresholdDeadRays>(rays)
24   scanResult:=exclusiveScan(thresh)
25   newIndexes:=reverseIndex(scanResult)
26   arrays:=gather(newIndexes,arrays)
```

Figure 2: Pseudocode for our ray-tracing algorithm made up of data-parallel primitives. Parallel primitives are shown in the form : primitive<functor>(args)

The functions referenced in this pseudocode are described in more detail online [2].

5 Study Overview

5.1 Test configurations

Our study was designed to test the viability of a data-parallel approach for ray tracing, and also to better understand the gaps incumbent to using this approach compared with architecture-specific ray tracers. Our tests varied four factors:

- Data set (12 options)
- Hardware architecture (7 options)
- Ray tracing software (3 options)
- Workload (3 options)

Not all combinations of options made sense, since not all ray tracing software ran on all architectures, and since not all workloads were needed for our evaluations. As a result, we tested 246 configurations (i.e., not $12 \times 7 \times 3 \times 3 = 756$).

5.1.1 Data set

Our pool of data sets consisted of scientific visualization data and standard ray tracing benchmarks:

- **Richtmyer-Meshkov (RM):** an isosurface of density on a time-slice of a RM-instability simulation from Lawrence Livermore National Laboratory. We considered five different sizes:
  - **RM_3.2M**: 3.2M triangles from a 41M cell regular grid (400x400x256).
  - **RM_1.7M**: 1.7M triangles from a 17M cell regular grid (256x256x256).
  - **RM_970K**: 970K triangles from a 8M cell regular grid (200x200x200).
  - **RM_650K**: 650K triangles from a 3M cell regular grid (192x144x144).
  - **RM_350K**: 350K triangles from a 2M cell regular grid (128x128x128).
- **Lead Telluride** : an isosurface from the charge density of a Lead Telluride (PbTe) crystal lattice provided by Oak Ridge National Laboratory.
  - **LT_350K**: 351K triangles from a 1.4M cell regular grid (113x113x133)
  - **LT_372K**: 372K triangles from a 1.4M cell regular grid (113x113x133)
- **Seismic**: 6.2M triangles generated from SPECFEM3D representing wave speed perturbations measured by seismograms provided by Oak Ridge National Laboratory.
- **Stanford Dragon**: a 100K triangle model based on the dragon from the Stanford Computer Graphics Laboratory.
- **Conference Room**: a 331K triangle model of a conference room at Lawrence Berkeley National Laboratory created by Anat Grynberg and Greg Ward.
- **Dabrovic Sponza**: a 66K triangle model of the interior of a building, created by Marko Dabrovic.
- **Happy Buddha**: a 1.2M triangle model from the Stanford Computer Graphics Laboratory.

Figure 3 shows renderings of the Richtmyer-Meshkov data sets.

5.1.2 Hardware architecture

We ran on seven hardware architectures: four GPU, two CPU, and a coprocessor. Of the GPUs, two were from NVIDIA's Kepler line, one from the Fermi line, and one from the Maxwell line. The hardware architectures were:


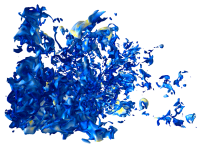

Figure 3: Ray tracings of the Richtmyer-Meshkov isosurfaces used in this study, using the RM_3.2M version of the data. The left image represents the results of basic intersection tests (i.e., WORKLOAD1 from Section 5.1.4) and the right image contains the pictures produced from the shaded pictures (i.e., WORKLOAD2).

- **GPU1:** A desktop computer with a GeForce GTX Titan Black, running 2,688 CUDA cores at 837MHz, 6GB of video memory, and a memory bandwidth of 288.4 GB/s.
- **GPU2:** The Texas Advanced Computing Center's (TACC) Maverick machine using its Tesla K40M, running 2,880 CUDA cores at 745 MHz, 12GB of video memory, and a memory bandwidth of 288.4 GB/s.
- **GPU3:** A desktop computer with a GeForce GTX 750Ti, running 640 CUDA cores at 1020 MHz, 2 GB of video memory, and a memory bandwidth of 86.4 GB/s.
- **GPU4:** A laptop computer with a GeForce GT 620M, running 96 CUDA cores at 625 MHz, 1 GB of video memory, and a memory bandwidth of 28.8 GB/s.
- **CPU1:** A desktop computer with Intel's i7 quad-core architecture (model 4770K), running at 3.5 GHz and with 32GB of memory.
- **CPU2:** The TACC Maverick machine with a Intel Xeon E5 CPU (model E5-2680 V2), containing ten cores running at 2.7GHz and with a total of 256GB of memory.
- **MIC:** An Intel Xeon Phi Coprocessor 3120 containing 57 cores running at 1.1GHz with 6GB of memory.

5.1.3 Ray tracing software

We considered three different implementations of ray tracing software:

- An EAVL-based implementation of the ray tracer described in Section 4.
- NVIDIA's OptiX Prime (see Section 2.2.1)
- Intel's Embree (see Section 2.2.2)

We now describe, for each package, the options for kernel and Bounding Volume Hierarchy (BVH), the quality of which greatly affects the number of triangle intersection tests.

EAVL: The EAVL-based ray tracer uses a split BVH [36], adapted from Alia and Laine's publicly available implementation [3]. BVH construction uses a split alpha of $1e^{-6}$ and a maximum leaf size of eight triangles. BVH construction is performed in serial on the CPU and construction time is not measured. Also, padding was not used in the final flat array representation of the BVH and is a possible area for future improvement.

Embree: Embree offers a wide variety of kernels for intersection. We tested three different Embree configurations and found that, on the CPU1 architecture, the most performant choice was a branching factor of four, performing four triangle intersections at once, and using Embree's high-quality BVH (i.e., "SBVH"). While Embree provides kernels for both single ray and packet-based traversal, single ray traversal was used to match the algorithm employed by OptiX Prime and the EAVL based ray tracer. We reused this Embree configuration for the CPU2 architecture for consistency, although we note that a different configuration could have led to increased performance. Finally, we used the gcc compiler, not Intel's ICC compiler, which advertises 10% to 20% improvements in performance. (The EAVL code also was run with gcc and also could have benefited.)

OptiX Prime: We used the default OptiX Prime configuration. (OptiX Prime does not have many configuration options, compared to Embree.) OptiX Prime's acceleration structure is the TR-BVH [21].

5.1.4 Workload

We considered three workloads:

- **WORKLOAD1:** Tracing rays with no shading.
- **WORKLOAD2:** Tracing rays with shading.
- **WORKLOAD3:** Tracing rays with all features enabled.

WORKLOAD1 corresponds to conventional ray tracing performance studies, where evaluation is based on how many millions of rays can be traced in one second. For this workload, the work for each ray is simply to calculate the index of the nearest intersected triangle and distance to the intersection point.

WORKLOAD2 corresponds to a scientific visualization use case, which is focused on meeting interactivity goals — a fixed frame rate with basic shading — for end users. For WORKLOAD2, the work for each ray is to calculate the index of the nearest intersected triangle, to calculate the exact intersection point and its barycentric coordinates, to interpolate the normal at the intersection using the barycentric coordinates, and then to calculate the shading (ambient, diffusion, specular) using the normal and the triangle's material properties, which are fetched from an array. Shading includes light attenuation factors and additional color using interpolated scalars that are indexed into a color map.

WORKLOAD3 also corresponds to a scientific visualization use case, but includes all features of the EAVL based implementation. The workload adds ambient occlusion with four samples per pixel, shadows, anti-aliasing, and stream compaction on top of the second workload.

5.2 Testing Procedure

When comparing ray tracing software, we used a shared software infrastructure for all tests, regardless of ray the tracing software package being tested. This software infrastructure was responsible for generating rays and interpreting results. The individual ray tracing software packages were only responsible for calculating intersections. We felt it was appropriate to use uniform infrastructure to surround the intersection calculation, since intersection calculation is understood to be the dominant term in ray tracing.

For each scene, we chose three or four camera positions, picking positions from the front of a data set, from the back, and zoomed in. For each of these camera positions, we performed one-hundred fifty renderings (all producing identical pictures). The first fifty rounds were warm-ups, and the latter one hundred rounds were used for measurement.

All tests were conducted at 1080P resolution (1920x1080). For GPU tests, data was transferred to the device once and output was retrieved only after all rounds were completed. Rays were sorted into Morton order to increase SIMD efficiency. On the CPU, transfer was a non-issue and the rays were left unsorted.

5.3 Measurements

For WORKLOAD1, we measured the number of primary ray intersections that were performed in one second. This number ranged from the low millions on CPUs to hundreds of millions on GPUs. For WORKLOAD2 and WORKLOAD3, we measured the time to render an image, although this time does not include readback from the device or display to a monitor. For all workloads, the numbers reported are the average over each camera position and over the

	RM_ 3.2M	RM_ 1.7M	RM_ 970K	RM_ 650K	RM_ 350K	LT_ 350K	LT_ 372K	Seismic	Stanford Dragon	Conf- erence	Dabrovic Sponza	Happy Buddha
GPU1 (Titan Black)	59.1	62.3	68.0	73.5	79.5	66.5	57.8	51.3	86.1	76.9	69.7	77.1
GPU2 (Tesla K40)	38.9	40.9	44.0	47.9	52.2	43.4	38.1	34.4	55.3	48.5	46.2	50.1
GPU3 (GTX 750Ti)	21.8	22.8	24.0	25.6	27.1	21.6	21.9	20.3	28.1	25.7	24.7	26.9
GPU4 (GT 620M)	3.7	3.9	4.3	4.7	5.3	4.3	3.8	3.3	5.2	5.3	4.9	5.0
CPU1 (Intel i7)	2.0	2.1	2.2	2.5	2.4	2.1	2.1	2.0	2.4	2.0	1.9	2.5
CPU2 (Intel Xeon)	5.5	5.9	6.2	6.7	6.6	5.8	5.7	5.7	6.8	5.7	5.4	6.8

Table 1: This table shows the frames per second for the EAVL-based ray tracer to do a rendering workload with shading, i.e., the performance expected when approximating traditional rasterization. The rows correspond to hardware configurations (see Section 5.1.2) and the columns correspond to data sets (see Section 5.1.1).

	RM_ 3.2M	RM_ 1.7M	RM_ 970K	RM_ 650K	RM_ 350K	LT_ 350K	LT_ 372K	Seismic	Stanford Dragon	Conf- erence	Dabrovic Sponza	Happy Buddha
CPU2 (Intel Xeon)	1.7	1.8	1.9	2.3	2.3	1.7	1.7	1.6	2.3	2.0	1.8	2.3
GPU1 (Titan Black)	10.3	9.7	10.6	14.6	15.6	9.0	8.5	9.6	16.1	12.4	9.6	16.3

Table 2: This table shows the frames per second for the EAVL-based ray tracer to do WORKLOAD3, i.e., all features of the EAVL-based ray tracer.

hundred rounds for each of these positions. No initialization time, such as BVH construction, is included in the averages.

6 Results

Section 6.1 describes the performance we observed for scientific visualization workloads using our data parallel ray tracer. Section 6.2 compares with leading edge ray tracers.

6.1 Performance of the Data-Parallel Approach

Table 1 shows the frames per second achieved with the EAVL-based ray tracer while rendering with shading (i.e., WORKLOAD2), while Tables 2 shows the rates achieved with full lighting effects (i.e., WORKLOAD3). WORKLOAD2 provides images that are substantially similar to the OpenGL/rasterization-based images produced by GPUs for most scientific visualization use cases, while WORKLOAD3 reflects rendering effects that benefit high-quality visualizations.

The Intel i7 architecture (CPU1) had a poor frame rate, ranging from 1.9 to 2.5 frames per second (FPS), as did the GT 620M (GPU4), which ranged from 3.3 to 5.3 FPS. However, these architecture are not representative of those found on modern supercomputers, because they lack sufficient computational power. The remaining architectures (GPU1, GPU2, GPU3, and CPU2) fared quite well, with the Xeon (CPU2) having a minimum of 5.4 FPS, and the GPUs all exceeding 20 FPS. Further, the Xeon rendered above 5 FPS, but would be over 10 FPS at 1024x1024.

In all, we conclude that ray tracing is a viable alternative to OpenGL/rasterization in a high-performance computing context, since compositing often limits parallel rendering's performance to similar overall frame rates. Further, the data parallel approach itself is shown to be viable for this task, which is important because scientific visualization programs need to run on many architectures, and our data parallel approach can provide portable performance.

6.2 Comparisons with Leading Ray-Tracers

We compare separately with OptiX Prime and Embree using WORKLOAD1.

6.2.1 OptiX Prime

Table 3 compares the performance between our data parallel ray tracer and OptiX Prime. OptiX Prime is clearly superior on the Kepler architectures (GPU1 and GPU2), and calculates between two and four times as many rays per second as our data parallel ray tracer. Optimization specifically for GPU1 and GPU2 provide an additional 25% performance increase over other GPUs [1]. On the Fermi and Maxwell architectures (GPU3 and GPU4), our data parallel ray tracer fares better, and outperforms OptiX Prime occasionally. While OptiX Prime's performance on the Keplers is extremely impressive, we note that the frames per second achieved on these data sets by our ray tracer (see Section 6.1) exceeded 20 FPS.

Saying it another way, Optix Prime's large number of triangle intersection tests can either enable direct lighting (i.e., the basic OpenGL lighting model) at frame rates well within the constraints traditionally imposed by distributed-memory compositing, or, alternatively, they can lead to significant additional lighting effects (e.g., ambient occlusion, area light sources for soft shadows, etc.) or different rendering paradigms (e.g., Monte Carlo techniques). While direct lighting is typically the goal for scientific visualization use cases, the latter approaches can both lead to insight and improve aesthetics. Finally, we believe that the performance gap can be attributed to the use of the GPU texture units in addition to methods used to maximize SIMD efficiency with GPUs.

6.2.2 Embree

Table 4 compares Embree and our data parallel ray tracer, and shows that Embree was approximately twice as fast in all configurations. We attribute the performance gap, in part, to Embree's use of architecture specific SIMD vector instructions which the EAVL OpenMP back-end is unable to capitalize on. In a similar vein to the observations made in Section 6.2.1, the frame rates we observed on CPU2 were at the interactivity threshold for scientific visualization use cases. That said, these results are closer to the threshold, and additional performance could be useful. Fortunately, architectural trends are pushing towards more and more compute power per node, meaning that advances in hardware should naturally push the frame rates even higher.

6.3 Emerging Architecture: MIC

Finally, we measured performance on the Intel Xeon Phi. Unlike the rest of our tests, these experiments used the Intel compiler, a requirement for MIC usage. Our usage was as a native application running directly on the Xeon Phi (i.e., not from a host running the main program offloading compute-intensive routines to the Phi).

Table 5 shows the performance for WORKLOAD1. Our first set of runs used EAVL's OpenMP back-end, which is primarily intended for multi-core CPUs. This led to disappointing results, with

	GPU1 (Titan Black)		GPU2 (Tesla K40)	
	EAVL	OptiX Prime	EAVL	OptiX Prime
RM_3.2M	189.1	333.1	124.8	264.5
RM_1.7M	203.2	319.9	136.6	266.8
RM_970K	228.5	437.1	152.8	347.1
RM_650K	262.9	538.4	172.9	420.4
RM_350K	300.7	564.4	197.5	436.5
LT_350K	230.1	431.4	150.8	357.6
LT_372K	187.3	394.9	124.7	322.4
Seismic	160.3	340.1	106.3	267.8
Dragon	344.9	667.5	224.1	533.9
Conference	299.4	673.8	197.1	524.8
Sponza	272.0	499.4	180.0	394.4
Buddha	282.0	608.7	185.5	477.3

	GPU3 (GTX 750Ti)		GPU4 (GT 620M)	
	EAVL	OptiX Prime	EAVL	OptiX Prime
RM_3.2M	86.4	96.8	11.6	10.0
RM_1.7M	94.4	110.9	12.7	11.1
RM_970K	103.9	123.5	14.4	15.2
RM_650K	119.5	151.9	16.7	17.9
RM_350K	134.5	155.0	20.4	20.8
LT_350K	103.3	119.3	15.0	16.4
LT_372K	88.5	111.3	12.2	14.0
Seismic	77.9	111.6	9.7	n/a
Dragon	144.7	197.7	20.2	27.3
Conference	127.4	180.4	21.9	29.2
Sponza	116.9	134.2	19.3	21.3
Buddha	130.9	176.0	18.1	22.8

Table 3: These tables show the number of rays per second (in millions) for the EAVL-based ray tracer and NVIDIA's OptiX Prime. The top and bottom tables correspond to Kepler and Fermi GPUs, respectively. The rows correspond to data sets (see Section 5.1.1) and the columns correspond to hardware configurations (see Section 5.1.2). This only measures intersection time (a common ray tracing benchmark) and there was no shading for WORKLOAD1.

rates that were not only far below CPU2, but also were below the rates of CPU1. However, this was likely because the Phi's vector unit was not being utilized. We then ran a second set of experiments with a prototype version of EAVL using a back-end based on Intel's ISPC [33], an open-source compiler that is capable of utilizing the Phi's vector unit. The resulting runs were dramatically better, with speedups ranging from 5X to 9X over OpenMP, and outperforming CPU2 in all cases. This experience supported one of the premises of this paper: by focusing on developing a data-parallel algorithm, we were able to achieve encouraging performance numbers via an improved back-end provided by the domain-specific language, rather than providing separate implementations of the algorithm.

We note that Embree can run on the MIC. We will compare Embree and our EAVL-based ray tracer in future work. (Note to reviewers: we could not get Embree ported to the Phi and are engaging with the Intel team.)

7 Conclusion and Future Work

We described an algorithm for ray tracing built on data parallel primitives and performed a series of experiments that demonstrated that this ray tracer can be used in the place of OpenGL/rasterization for scientific visualization use cases. We believe this algorithm to be novel, as it is the first algorithm constructed entirely from data parallel primitives, although we recognize that the spirit of the algorithm is similar to preceding, non-hardware-agnostic approaches.

	CPU1 (Intel i7)		CPU2 (Intel Xeon)	
	EAVL	Embree	EAVL	Embree
RM_3.2M	9.1	21.4	28.3	48.4
RM_1.7M	9.5	21.8	27.0	52.4
RM_970K	10.3	24.4	29.3	59.1
RM_650K	13.0	28.8	35.6	65.9
RM_350K	11.9	27.7	33.3	64.8
LT_350K	9.7	20.1	27.7	51.9
LT_372K	9.4	21.2	26.1	56.5
Seismic	9.4	18.1	25.2	43.2
Dragon	13.3	30.9	37.2	67.8
Conference	9.7	33.9	28.3	70.4
Sponza	8.0	17.2	24.0	50.5
Buddha	13.7	37.7	37.3	73.9

Table 4: This table shows the number of rays per second (in millions) for the EAVL-based ray tracer and Intel's Embree product. The rows correspond to data sets (see Section 5.1.1) and the columns correspond to hardware configurations (see Section 5.1.2). This only measures intersection time (a common ray tracing benchmark) and there was no shading for these workloads.

Further, we believe this result is impactful, because scientific visualization software developers are striving to build large, community projects that run on many, many architectures. New, portably performant algorithms such as our own help bolster this effort.

We also compared our ray tracer to leading edge ray tracers, to better understand the performance gap incumbent to a hardware-agnostic approach. Although the performance gap was significant (ranging from 1.6X to 2.6X for likely HPC architectures on scientific data sets), it was small enough to provide encouragement that the data parallel approach — and its benefits in portable performance, longevity, and programmability — are a good direction.

In terms of future work, we would like to better understand the causes for the performance gap. Is it because the leading edge ray tracers have made such a significant investment? (And thus a data parallel approach might close the gap.) Or is it because the leading edge ray tracers make use of architectural-specific features that are outside of the hardware-agnostic philosophy of data parallelism? (And thus a data parallel approach can never catch up.) Likely, the answer is a combination of the two. We have performed some preliminary work, and found that adding features such as texture support does make a significant difference. In another direction for future work, we would like to explore additional architectures, and further understand whether portable performance is maintained.

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. As such, the U.S. Government retains a nonexclusive, royalty-free right to publish or reproduce this article.

Finally, we thank the anonymous reviewers for their helpful comments.

References

[1] Nvidia optix ray tracing engine, 2014. https://developer.nvidia.com/optix.

[2] Supplemental description of pseudocode for this algorithm, 2014. http://ix.cs.uoregon.edu/ mlarsen/pacvis14.pdf.

	RM_ 3.2M	RM_ 1.7M	RM_ 970K	RM_ 650K	RM_ 350K	LT_ 350K	LT_ 372K	Seismic	Stanford Dragon	Conf- erence	Dabrovic Sponza	Happy Buddha
OpenMP	5.77	6.2	6.4	7.3	6.8	6.3	6.0	6.2	7.5	6.3	5.6	7.6
OpenMP/ISPC	30.5	35.5	38.6	38.3	43.7	36.2	34.4	25.4	56.3	43.7	50.8	41.0

Table 5: Millions of ray intersections per second for WORKLOAD1 on the Intel Xeon Phi. While initial results using the OpenMP back-end were disappointing, incorporating an ISPC-based back-end into EAVL significantly improved performance and eliminated the need to modify our data-parallel primitives-based ray intersection algorithm.

[3] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149. ACM, 2009.

[4] T. Aila, S. Laine, and T. Karras. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *Proceedings of ACM High Performance Graphics 2012, Posters*, pages 9–16, 2012.

[5] U. Ayachit, B. Geveci, K. Moreland, J. Patchett, and J. Ahrens. The ParaView Visualization Application. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 383–400. Oct. 2012.

[6] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W.-M. Hwu, editor, *GPU Computing Gems*, pages 359–371. Elsevier/Morgan Kaufmann, 2011.

[7] J. Bigler, A. Stephens, and S. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of Interactive Ray Tracing*, pages 187–196, 2006.

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

[9] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH Computer Graphics*, volume 11, pages 192–198. ACM, 1977.

[10] S. Boulos, I. Wald, and C. Benthin. Adaptive Ray Packet Reordering. In *Proceedings of Interactive Ray Tracing*, pages 131–138, 2008.

[11] C. Brownlee, T. Fogal, and C. Hansen. Gluray: Enhanced ray tracing in existing scientific visualization applications using opengl interception. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50. The Eurographics Association, 2012.

[12] C. Brownlee, J. Patchett, L. Lo, D. DeMarle, C. Mitchell, J. Ahrens, and C. Hansen. A study of ray tracing large-scale scientific data in two widely used parallel visualization applications. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 51–60. The Eurographics Association, 2012.

[13] H. Childs et al. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *Proceedings of SciDAC 2011*, Denver, CO, July 2011.

[14] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. Weber, and E. W. Bethel. Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 30(3):22–31, May/June 2010.

[15] Cilk plus. `http://www.cilkplus.org/`, 2014.

[16] Z. DeVito et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[17] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *EUROGRAPHICS*, 2010.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Scientific and engineering computation. MIT Press, Cambridge, MA, 2nd edition, 1999. 99016613 William Gropp, Ewing Lusk, Anthony Skjellum. Includes bibliographical references and index.

[19] W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

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

[21] T. Karras and T. Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99. ACM, 2013.

[22] Khronos OpenCL Working Group. *The OpenCL specification, version 1.1*. Khronos, 2010.

[23] A. Knoll, I. Wald, P. A. Navrátil, M. E. Papka, and K. P. Gaither. Ray tracing and volume rendering large molecular data on multi-core and many-core architectures. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, page 5. ACM, 2013.

[24] C. Lauterbach1 et al. Fast bvh construction on gpus. *EUROGRAPHICS*, March 2009.

[25] L.-t. Lo, C. Sewell, and J. Ahrens. PISTON: A portable cross-platform framework for data-parallel visualization operators. pages 11–20. Eurographics Symposium on Parallel Graphics and Visualization, 2012.

[26] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30. The Eurographics Association, 2012.

[27] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. 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*, pages 97–104, October 2011.

[28] P. A. Navrátil, H. Childs, D. S. Fussell, and C. Lin. Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 20(6), 2014.

[29] L. Northam and R. Smits. Hort: Hadoop online ray tracing with mapreduce. In *ACM SIGGRAPH 2011 Posters*, page 22. ACM, 2011.

[30] NVIDIA. *NVIDIA CUDA: Reference manual*, February 2014.

[31] S. G. Parker et al. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (proceedings of SIGGRAPH)*, 29(4):66, August 2010.

[32] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics (Proceedings of SIGGRAPH)*, 31(Annual Conference Series):101–108, August 1997.

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

[34] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):1176–1185, 2005.

[35] P. Schroder and S. Drucker. A data parallel algorithm for raytracing of heterogeneous databases. *Proceedings of Computer Graphics Interface*, pages 167–175, 1992.

[36] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13. ACM, 2009.

[37] Intel theading building blocks (TBB). `http://www.threadingbuildingblocks.org/`, 2014.

[38] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 49–57. IEEE, 2008.

[39] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):153–164, 2001.

[40] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (proceedings of SIGGRAPH, to appear)*, 2014.