PERFORMANCE MODELING OF IN SITU RENDERING

by

MATTHEW CRIVELLI LARSEN

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2016

DISSERTATION APPROVAL PAGE

Student: Matthew Crivelli Larsen

Title: Performance Modeling of In Situ Rendering

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Hank Childs	Chair Advisor
Allen Malony	Core Member
Boyana Norris	Core Member
Eric Corwin	Institutional Representative
Paul Navrátil	Outside Member

and

Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded December 2016

DISSERTATION ABSTRACT

Matthew Crivelli Larsen

Doctor of Philosophy

Department of Computer and Information Science

December 2016

Title: Performance Modeling of In Situ Rendering

With the push to exascale, *in situ* visualization and analysis will play an increasingly important role in high performance computing. Tightly coupling *in situ* visualization with simulations constrains resources for both, and these constraints force a complex balance of trade-offs. A performance model that provides an *a priori* answer for the cost of using an *in situ* approach for a given task would assist in managing the trade-offs between simulation and visualization resources. In this work, we present new statistical performance models, based on algorithmic complexity, that accurately predict the run-time cost of a set of representative rendering algorithms, an essential *in situ* visualization task. To train and validate the models, we create data-parallel rendering algorithms within a light-weight *in situ* infrastructure, and we conduct a performance study of an MPI+X rendering infrastructure used *in situ* with three HPC simulation applications. We then explore feasibility issues using the model for selected *in situ* rendering questions.

This dissertation includes previously published coauthored material.

CURRICULUM VITAE

NAME OF AUTHOR: Matthew Crivelli Larsen

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
California State University at Sacramento, Sacramento, CA

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2016,
University of Oregon
Master of Science, Computer and Information Science, 2015,
University of Oregon
Bachelor of Science, Computer Science, 2012,
California State University at Sacramento

AREAS OF SPECIAL INTEREST:

Scientific Visualization
High Performance Computing, Computer Graphics
Computer Graphics

PROFESSIONAL EXPERIENCE:

Staff Scientist, Lawrence Livermore National Laboratory, 2016 - Present

Gradute Research Assistant, University of Oregon, 2014-2015

Gradute Teaching Assistant, University of Oregon, 2013

GRANTS, AWARDS AND HONORS:

Area Exam, Passed With Distiction, 2016

J. Donald Hubbard Family Scholarship, University of Oregon, 2015

Directed Research Project, Passed With Distiction, 2014

PUBLICATIONS:

M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. S. Meredith, and H. Childs. ”Performance Modeling of In Situ Rendering,” In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 24:1-24:12

M. Larsen, K. Moreland, C. R. Johnson, and H. Childs, ”Optimizing Multi-Image Sort-Last Parallel Rendering,” Proceedings of IEEE Symposium on Large Data Analysis and Visualization, Baltimore, MD, 2016, pp. 37046

K. Moreland, C. Sewell, W. Usher, L. Lo, J. S. Meredith, D. Pugmire, J. Kress, H. Schroots, K. Ma, H. Childs, **M. Larsen**, C. Chen, R. Maynard, and B. Geveci. ”VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures,” IEEE Journal of Computer Graphics and Applications, 2016, pp. Volume 36, Issue 3, pp. 48-58

M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, ”Strawman: A Batch In Situ Visualization and Analysis Infrastructure for Multi-Physics Simulation Codes,” Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, Austin, TX, 2015, pp. 30-35.

K. Moreland, **M. Larsen**, and H. Childs, ”Visualization for Exascale: Portable Performance is Critical,” Journel of Supercomputing frontiers and Innovations, 2015, Volume 2, Issue 3, pp. 53-62.

M. Larsen, S. Labasan, P. Navrátil, J. S. Meredith, and H. Childs, ”Volume Rendering Via Data-Parallel Primitives,” Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV), Cagliari, Italy, 2015, pp. 53-62.

S. Labasan, **M. Larsen**, and H. Childs, ”Exploring Tradeoffs Between Power and Performance for a Scientific Visualization Algorithm”, Proceedings of IEEE Symposium on Large Data Analysis and Visualization, Chicago, IL, 2015, pp. 73-80.

M. Larsen, J. S. Meredith, P. A. Navrtil and H. Childs, ”Ray tracing within a data parallel framework,” 2015 IEEE Pacific Visualization Symposium (PacificVis), Hangzhou, 2015, pp. 279-286.

ACKNOWLEDGEMENTS

I thank Hank Childs for being an exceptional advisor and for his guidance through the Ph.D. processs.

This work was funded in part by US National Science Foundation grants ACI-1339840 and ACI-1339863. 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. Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-CONF-676409 and LLNL-CONF-689101. Some of this material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, including Contract No. DE-AC02-05CH11231, and the U.S. Department of Energy, National Nuclear Security Agency, Advanced Simulation & Computing (ASC) program.

For my parents and my brother, who have supported and encouraged me throughout my life. I would not been able to achieve this without them. To my advisor, Hank Childs, who unwittingly encouraged me to go down this path during a conversation between fellow basketball players, years before he joined the faculty at the University of Oregon. His advising during my time at Oregon was outstanding, and I would not have been able to walk this path without it. I thank my co-authors who all have made valueable contributions to our publications, and those who contibuted to material within this dissertation. They include Eric Brugger, Hank Childs, Jim Elliot, Kevin Griffin, Cyrus Harrison, James Kress, Stephanie Labasan, Jeremy Meredith, Ken Moreland, Paul Navrátil, and Dave Pugmire. I thank all the members of the CDUX research group who have all been supported of one another. Specifically I would like to thank Ryan Bleile, James Kress, Stephanie Labasan, and Shaomeng Li.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Visualization on high performance computers has traditionally been accomplished with *post hoc* processing. With this model, a simulation code writes its state to disk at regular intervals (or irregular intervals), and a visualization program reads that data from disk after the completion of the simulation. In *post hoc* processing, the dominant factor in overall execution time for most common visualization operations is I/O performance (43). Further, while I/O bandwidth is increasing on new supercomputers, the ability to generate data is increasing much more quickly, meaning the percentage of writable FLOPS is decreasing. As a result, *post hoc* processing may soon become impractical on leading-edge supercomputers.

In situ processing is widely viewed as the best suited paradigm for upcoming supercomputers (85; 17; 42). With the in situ paradigm, visualizations are created while the simulation is running, making it unnecessary to store the simulation's state to disk and read it afterwards. The term in situ processing actually represents a family of approaches. One approach, referred to in this dissertation as tightly-coupled, shares compute resources between simulation and visualization. That is, when it is time to perform visualization tasks, the simulation pauses, hands control to a visualization routine which produces images, and then the visualization routine hands control back to the simulation. This approach is used by both ParaView Catalyst (51) and VisIt LibSim (136). Another common approach, referred to as loosely-coupled, uses dedicated sets of resources for simulation and visualization routines, and data is sent between simulation and visualization resources over the network. This approach is also referred to as in transit processing (99) or staging, and is the approach most frequently employed by ADIOS (82).

The requirements for *post hoc* processing and in situ processing are different. For *post hoc* processing, the driving requirement was interactivity for human observers. If the user was satisfied with the delivery rate of results, then the performance requirement was met. With in situ processing, and especially with tightly-coupled in situ processing, the performance requirement is different. Running visualization routines with the tightly-coupled approach means that compute resources are not being devoted to the simulation, and so it slows down the overall simulation time.

1.1 Feasibility Question

Given a set of input parameters, how long does it take to render an image? Rendering is often a relatively quick operation (on the order of a second), and so this feasibility question is not all that interesting if the desired rendering task is to, for example, render one frame every 500 cycles of the simulation. However, recent research (64; 19; 20) has inspired a new approach, where thousands to millions of renderings are extracted during the simulation, in order to construct an image database that can be explored by domain scientists after the simulation has completed. These images are often of the same geometry (such as an isosurface), but with different rendering parameters, including camera angle and shading. When considering this number of renderings, the feasibility question we consider becomes very important.

Compute resources are finite, and simulations operate with a fixed time budget. With tightly-coupled in situ, the simulation cannot advance while images are rendered and spending more time rendering limits the total number of time steps simulations can advance to. With loosely-coupled in situ, simulation time steps are sent over the network to a subset of nodes for visualization, and if rendering cannot keep up with the influx of simulation data, then incoming time steps are lost until memory is freed after rendering is complete. Both scenarios are problematic. On the one hand, long running renderings prevent the simulation from advancing to completion, and on the other hand, when simulations render sparsely in time, they potentially lose valuable information. Simulation scientists must balance rendering time with their own goals, and this trade-off engenders an important feasibility question: is it possible to perform X_1 rendering tasks while devoting no more than X_2 time to these tasks?

Up until now, we have had no answer to this question beyond anecdotal analysis. While the question is easily understood, it is not simple to answer. Emerging exascale architectures, on which in situ analysis will become critical, are becoming increasingly diverse. Current CPU architectures have two very different levels of parallelism, i.e. threads and vector units, while current GPU architectures have tens of thousands of threads. Additionally, there are different types of rendering algorithms each with their own algorithmic complexities (e.g., image order, object order, and acceleration structure traversal). Taking the cross product of architectures and algorithms into account, the feasibility question becomes all the more difficult to answer.

1.2 Performance Modeling

A performance model that provides an *a priori* answer to the cost of a given task would assist in managing the complex trade-offs between visualization and simulation resources, and performance models for rendering algorithms would provide an answer to the feasibility question. With time being a constrained resource at exascale, we could evaluate if a task is even possible. Further, we could immediately identify or rule out alternative rendering algorithms for the same task.

Creating individual performance models for every algorithm and architecture combination would require significant effort. To reduce the effort needed to create a performance model, we need a straight forward methodology that can be applied to any algorithm and architecture. Armed with a methodology, we could apply it to rendering algorithms and answer the feasibility question. Further, a methodology for in situ rendering would open the door to reasonably estimating the cost of other visualization and analysis algorithms.

1.3 Hybrid Parallelism and Portable Performance

Hybrid Parallelism

Traditionally, simulation and visualization codes have used one MPI task per core, but this approach is proving inefficient at large scale (61), since the numbers of cores per node are dramatically increasing on supercomputing architectures. The next Department of Energy (DOE) procurement of supercomputers will push the core count per node even higher than it is today, making the ability to harness node-level parallelism even more important. Additionally, there are two distinct architectures emerging for the coming generation of supercomputers. An in situ visualization infrastructure must be able to utilize the same architecture a simulation runs on, and for tightly-coupled in situ, the infrastructure needs direct access to simulation memory, which exists in different phyisical spaces between GPU and CPU architectures.

We do not currently have such an infrastructure to connect simulations, on arbitrary architectures, to visualization in situ. However, a diverse set of supercomputing architectures, and simulations targeting them, presents a problem for visualization. Visualization has hundreds of algorithms, and with two or more architectures, the community would exhaust all funding implementing each algorithm for each architecture. One solution to this problem is portable performance through data-parallel frameworks.

Portable Performance

The current set of production visualization systems (e.g., VisIt (41) and ParaView (18)) have been retrofitted for in situ, but they still use rendering algorithms developed for post-hoc analysis.

In current architectures, memory is already a constrained resource, and at exascale, memory constraints will increase as architectural trends continue to reduce the memory per core. Additionally, there is a tension between the time and frequency of visualization. Since both memory and time will be constrained resources at exascale, algorithms designed for post-hoc analysis may no longer be appropriate. Consequently, we need to re-examine rendering in terms of exascale in situ; Figure 1 illustrates the unknown subset of current algorithms that will be viable in an exascale environment.

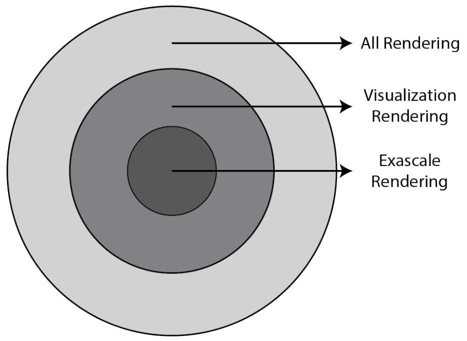

FIGURE 1. The outer circle represents all rendering research, the middle circle represents the portion of rendering used by visualization, and the inner circle represents the rendering techniques that are viable at exascale. Which techniques belong in the inner circle is still an open question.

To effectively answer the feasibility question, we must evaluate and test rendering algorithms that are suitable for in situ. Implementing and testing all algorithms and architectures is time consuming, and not all implementations are publicly available (108). Further, some algorithms have not been adapted for hybrid parallelism and modern architectures.

To address this problem, researchers have created a number of architecturally agnostic programming paradigms for both general purpose computing (25; 50; 59; 127) and visualization specific applications (92; 81; 96; 100). CPU and GPU architectures share thread-level parallelism, and abstracting away the underlying computer architectures, by using data-parallel operations, allows a single implementation to run efficiently on arbitrary architectures. The proposition of portable performance is particularly attractive to visualization and analysis, which has hundreds of algorithms.

Implementing rendering algorithms using data-parallel programming models presents us with three opportunities. First, ray tracing algorithms have been highly optimized for specific architectures (138; 108), and they present an opportunity to validate the promise of portable performance of algorithms implemented using data-parallel frameworks. Second, we can begin to identify those algorithms that are suitable at exascale by evaluating performance on different architectures. Finally, we can validate performance models, based on algorithmic complexity, on multiple architectures using a single implementation.

1.4 Dissertation Outline

The goal of my dissertation is to answer the feasibility question for in situ rendering by constructing a methodology for creating performance models based on algorithmic complexity. The performance models will be applicable to arbitrary architectures and sub-architectures such as CPUs (e.g., Haswell and Sandy Bridge) and GPUs (e.g., NVidia K40 and K20). Additionally, we perform a thorough study to validate the results of the performance models and thus the methodology used to create them. There was a significant amount of work that needed to be performed establish a foundation for the feasibility study. The foundational research is described in Chapters II, III, and IV, while the feasibility research is described in Chapter V. This dissertation is organized as follows:

– Chapters II and III: we created and implemented multiple rendering algorithms that execute on arbitrary architectures and we validated their performance against community standards.

– Chapter IV: we built an in situ infrastructure integrated with multiple simulations, linking the rendering algorithms to data.

– Chapter V: we established a methodology and created performance models based on algorithmic complexity. Finally, we conducted a thorough evaluation and show the performance models utility by answering several in situ viability questions.

– Chapter VI: we discuss the future directions of visualization and performance modeling.

1.5 Co-authored Material

Much of the work in this dissertation is from previously published co-authored material. Below is a listing connecting the chapters with the publications and authors that contributed. That said, for each of these publications, I was not only the first-author of the paper, but also the primary contributor for implementing systems, conducting studies, and writing manuscripts.

– Chapter II: (73) was a collaboration between Jeremy Meredith (ORNL), Paul Navrátil (TACC), Hank Childs (UO and LBL), and myself.

– Chapter III: (76) was a collaboration between Stephanie Labasan (UO), Paul Navrátil (TACC), Jeremy Meredith (ORNL), Hank Childs (UO and LBL), and myself.

– Chapter IV: (74) was a collaboration between Eric Brugger (LLNL), Hank Childs (UO and LBL), Jim Elliot (AWE), Kevin Griffin (LLNL and UC Davis), Cyrus Harrison (LLNL), and myself.

– Chapter V: (75) was a collaboration between Cyrus Harrison (LLNL), James Kress (UO and ORNL), Dave Pugmire (ORNL), Jemery Meredith (ORNL), Hank Childs (UO and LBL), and myself (UO and LLNL).

CHAPTER II

DATA-PARALLEL RAY TRACING

Most of the text in this chapter comes from (73), which was a collaboration between Jeremy Meredith (ORNL), Paul Navrátil (TACC), Hank Childs (UO and LBL), and myself. The introduction was a collaboration between Hank Childs and myself, and the related work was primarily written by Jeremy Meredith and Paul Navrátil, although I provided some edits. In addition, I received some guidance from Hank and Paul about methods and algorithm development. I performed all the software development, designed and executed the performance study, and contributed to the majority of the other sections.

In this chapter, we explore the first of two rendering algorithms within the context of a data-parallel framework. By validating the performance of a single implementation on multiple architectures, we can simply the generation and validation of the performance models in Chapter V.

Ray tracing has both regular and irregular memory access patterns, and there exist highly-optimized architecturally specific implementations for both the CPU and GPU. Due to these properties, ray tracing is an ideal algorithm to validate the data-parallel approach. In this chapter, we describe trends is modern supercomputing architectures that motivate the pursuit of performance portable approaches, describe data-parallelism and fundamental primitives, describe our ray tracing algorithm, and validate the performance through an extensive study,

2.1 Motivation

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* (28), 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 have emerged, including DAX (97), EAVL (90), and PISTON (80). 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-parallel primitives to software developers and asking them to re-think 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 chapter, we explore ray tracing within the context of a data-parallel framework. The contributions of this chapter 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 a viable alternative to rasterization for scientific visualization use cases.

2.2 Related Work

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 (55; 56). For scientific visualization, this approach has been embraced by tools such as VisIt (41) and ParaView (24), and has seen usage ranging from tens of thousands of processing elements for production runs (43) to hundreds of thousands of processing elements for research experiments (62). 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 (105), OpenCL (68), Cilk Plus [1], and Threading Building Blocks [2].

The parallel algorithm primitives described by Blelloch (28) 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 (26). Domain knowledge can extend these techniques; for example, domain specific languages like Liszt (47) for scientific

[1] http://www.cilkplus.org/

[2] http://www.threadingbuildingblocks.org/

solvers and APIs like EAVL (90) for visualization and analysis provide one more level of productivity beyond domain-independent collections of parallel primitives. See Section 2.3 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 (32; 33; 102), 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. (119) 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 (70) 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. (104) 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.

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 (132). This fine-grained parallelism has formed the fundamental building block of many modern high-performance ray tracers (e.g., (21; 27; 109; 115; 132; 133)), including vendor-specific implementations for multi-core and many-core architectures. We discuss implementations from NVIDIA and Intel in greater detail below.

NVIDIA OptiX

On GPUs, NVIDIA's OptiX Prime $^{\text{TM}}$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. (21; 22), 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 (21) 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 (22), but is not beneficial on all architectures.

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 (133). Using vector instruction allows for the efficient traversal of a BVH with branching factors that match the width of the SIMD lanes (131). 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.

2.3 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.

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.

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.

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 (78).

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 (28), 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 (54; 109; 133). Furthermore, scan can be used to build histograms, conduct binary searches, construct k-D trees, and many more (28).

2.4 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. (30) 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. (112).

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 (102)).

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 2.6, 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.

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 (21), 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.

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.

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., (30)).

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.

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 (29) and adds the resulting color contribution to the appropriate pixel in the framebuffer.

Pseudocode

2.5 Study Overview

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)

```
rays←rayGenerationMap(eye) //initial rays
rayDepth←0
while $rayDepth < maxDepth$ do
    hits:=map¡intersect¿(rays)
    (reflectedRays,intersects) ← map<reflect>(hits)
    rays:=reflectedRays
    if(compactOn) compactArrays()
    //ambient occlusion
    occRays←scatter<occRayGen>(intersects,normals,numSamples)
    hits←map<intersect>(occRays,maxDistance)
    occlusion←gather(hits,numSamples)
    shadowHits←=map<intersect>(intersects,lights)
    //generate and accumulate rgb values
    rgb←map<shader>(intersects, occlusion,shadowHits)
    colorBuffer←map<accum>(rgb,colorBuffer)
    rayDepth++
end
screenBuffer←gather<antiAlias>(colorBuffer)
Function compactArrays
    thresh←map<thresholdDeadRays>(rays)
    scanResult←exclusiveScan(thresh)
    newIndexes←reverseIndex(scanResult)
    arrays←gather(newIndexes,arrays)
```

Algorithm 1: Pseudocode for our ray-tracing algorithm made up of data-parallel primitives. Parallel primitives are shown in the form : primitive<functor>(args)

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 $12 \times 7 \times 3 \times 3 = 756$).

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 2 shows renderings of the Richtmyer-Meshkov data sets.

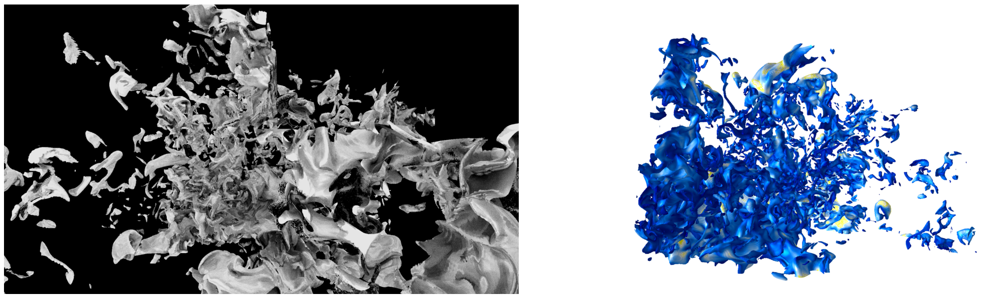

FIGURE 2. Ray tracings of the Richtmyer-Meshkov isosurfaces used in this study, using the RM_3.2M version of the data. The left image represents the results of basic intersection tests (i.e., WORKLOAD1 from Section) and the right image contains the pictures produced from the shaded pictures (i.e., WORKLOAD2).

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:

- **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.

Ray tracing software

We considered three different implementations of ray tracing software:

- An EAVL-based implementation of the ray tracer described in Section 2.4.
- NVIDIA's OptiX Prime (see Section 2.2).
- Intel's Embree (see Section 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 (126), adapted from Alia and Laine's publicly available implementation (21). 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 re-used 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 TRBVH (67).

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.

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.

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

2.6 Results

The first subsection describes the performance we observed for scientific visualization workloads using our data parallel ray tracer, and the second subsection compares performance with leading edge ray tracers.

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 2.5) and the columns correspond to data sets.

	GPU1	GPU2	GPU3	GPU4	CPU1	CPU2
RM_3.2M	59.1	38.9	21.8	3.7	2.0	5.5
RM_1.7M	62.3	40.9	22.8	3.9	2.1	5.9
RM_970K	68.0	44.0	24.0	4.3	2.2	6.2
RM_650K	73.5	47.9	25.6	4.7	2.5	6.7
RM_350K	79.5	52.2	27.1	5.3	2.4	6.6
LT_350K	66.5	43.4	21.6	4.3	2.1	5.8
LT_372K	57.8	38.1	21.9	3.8	2.1	5.7
Seismic	51.3	34.4	20.3	3.3	2.0	5.7
Dragon	86.1	55.3	28.1	5.2	2.4	6.8
Conference	76.9	48.6	25.7	5.3	2.0	5.7
Sponza	69.7	46.2	24.7	24.9	1.9	5.4
Buddha	77.1	50.1	26.9	5.0	2.5	6.8

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)

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.

	CPU2 (Intel Xeon)	GPU1 (Titan Black)
RM_3.2M	1.7	10.3
RM_1.7M	1.8	9.7
RM_970K	1.9	10.6
RM_650K	2.3	14.6
RM_350K	2.3	15.6
LT_350K	1.7	9.0
LT_372K	1.7	8.5
Seismic	1.6	9.6
Dragon	2.3	16.1
Conference	2.0	12.4
Sponza	1.8	9.6
Buddha	2.3	16.3

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.

Comparisons with Leading Ray Tracers

We compare separately with OptiX Prime and Embree using WORKLOAD1.

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 [3]. 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 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.

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,

[3]NVIDIA OptiX ray tracing engine: https://developer.nvidia.com/optix

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 the previous subsection, 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.

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 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 (113), 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.

2.7 Summary

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. 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.

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 and the columns correspond to hardware configurations. This only measures intersection time (a common ray tracing benchmark) and there was no shading for WORKLOAD1.

	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 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 and the columns correspond to hardware configurations (see Section 2.5). This only measures intersection time (a common ray tracing benchmark) and there was no shading for these workloads.

	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 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.

	OpenMP	OpenMP/ISPC
RM_3.2M	5.77	30.5
RM_1.7M	6.2	35.5
RM_970K	6.4	38.6
RM_650K	7.3	38.3
RM_350K	6.8	43.7
LT_350K	6.3	36.2
LT_372K	6.0	34.4
Seismic	6.2	25.4
Dragon	7.5	56.3
Conference	6.3	43.7
Sponza	5.6	50.8
Buddha	7.6	41.0

CHAPTER III

DATA-PARALLEL VOLUME RENDERING

Most of the text in this chapter comes from (77), which was a collaboration between Stephanie Labasan (UO), Paul Navrátil (TACC), Jeremy Meredith (ORNL), Hank Childs (UO and LBL), and myself. The introduction was a collaboration between Hank Childs and myself, and the related work was primarily written by Jeremy Meredith and Paul Navrátil, although I provided some edits. The algorithm was developed by myself with ideas contributed by Hank Childs and Jeremy Meredith. The figures were created by Stephanie Labasan. I performed all the software development, designed and executed the performance study, and contributed to the majority of the other sections.

In the previous chapter, we explored ray tracing, a surface rendering algorithm, within a data-parallel framework. In this chapter, we explore volume rendering, another important rendering algorithm, by implementing an novel algorithm using a data-parallel framework. This chapter further validates the data-parallel approach and its results will be useful for developing performance models on heterogeneous architectures.

We start by motivating the work described in this chapter, then we cover volume rendering and data-parallelism related work. Next, we describe our algorithm and perform a study to validate the performance against community standards. Finally, we discuss and summarize the results.

3.1 Motivation

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).

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 (97), EAVL (91), and PISTON (80) — all realized this goal by building a DSL-like infrastructure 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 (120), 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 chapter is organized as follows: Section 3.2 surveys related work, Section 3.3 describes our algorithm, Section 3.4 outlines our study, and Section 3.5 explores the resulting performance.

3.2 Related Work

For an overview of data-parallel primitives, we refer the reader to Section 2.2.

Volume Rendering

Volume rendering (79; 49) 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.

Unstructured Volume Rendering

Unstructured grid volume rendering algorithms were surveyed excellently by Silva et al (124).

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 (123). This method was extended to GPUs in 2002 (139). An important advancement to the algorithm came with HAVS (Hardware Assisted Visibility Sorting) (36), 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. (35) developed a ray-caster for unstructured data. Their algorithm was implemented in VTK and is still commonly used today.
- Z-Sweep (52) is an algorithm that advances a plane through the volume in depth. Childs et al. (40) 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 (39).

While our closest relative with respect to volume rendering is likely the distributed-memory algorithm by Childs (40), 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 (124). On the CPU side, there are fewer shared-memory parallel works. Notable examples include the hybrid-parallel work by Howison et al. (60; 62) and the CPU and Xeon Phi work by Knoll et al.(69). Expanding the scope beyond volume rendering, Larsen et al. (73) 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 (128). 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.

3.3 Algorithm

For an overview of the data-parallel primitives discussed in this chapter, please refer to Section 2.3.

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.

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.

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*.

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.

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.

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.

Data-Parallel Primitives Pseudocode

The pseudocode is specified in Algorithm 2.

3.4 Study Overview

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) (113) and GPU (creating coalesced memory accesses)(38).

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

```
/*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 2: 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.

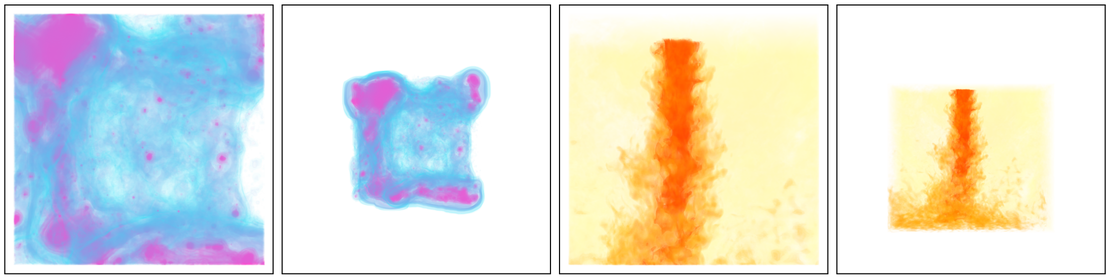
FIGURE 3. Volume renderings produced in this study. The two images on the left are of density from a cosmology simulation. The two images on the right are of temperature from a thermal hydraulics simulation. For each pair of images, the larger one is zoomed in (meaning the data set fills the screen) 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).

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.

Configurations

Our study consisted of two rounds and 56 total tests. Each test was of a volume rendering that created a 1024 $\times$ 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.

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.

Round 2: Comparison With Community Software

This round compared our algorithm to existing standards for unstructured data, specifically the HAVS volume renderer (36) on the GPU, the integration-based ray-caster derived from Bunyk et al. (35) on the CPU, and the sampling-based ray-caster in VisIt (40) 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.)

Testing Options

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.

Data Sets and Camera Positions

Our study examined the following four data sets:

– **Enzo-1M**: a cosmology data set from the Enzo (107) 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 (53) 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 3 shows volume renderings for these data sets, including the zoomed in and close up camera positions.

Performance Measurements

We used the following techniques for performance measurement:

– With our algorithm on **CPU1**, we enabled PAPI (16) 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 (106) 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.

3.5 Results

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

Performance Analysis of Algorithm

CPU Performance

Figure 4 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 being spent iterating over tetrahedrons, as opposed to identifying values at sample locations.

GPU Performance

We present the data in two ways. Figure 5 shows the runtime per test by phase, and Table 6 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.

TABLE 6. 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.

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

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 7 shows measurements on the CPU and GPU for the **Enzo-10M** data set with a close up view, using four passes.

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

TABLE 7. 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.

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

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.

Scalability

Table 8 shows the scalability of the algorithm on **CPU1**. While total time (i.e., the number of threads multiplied by the wall clock time) increases by 50% from 1 to 24 threads (the number of CPU cores on the node), the algorithm appears to scale generally well overall.

TABLE 8. 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.

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

Comparisons With Community Software

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 HAVS. 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 6 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.

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. (35). 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 7 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.

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 (40). 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 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 9.

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.

TABLE 9. 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.

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

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).

3.6 Summary

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.

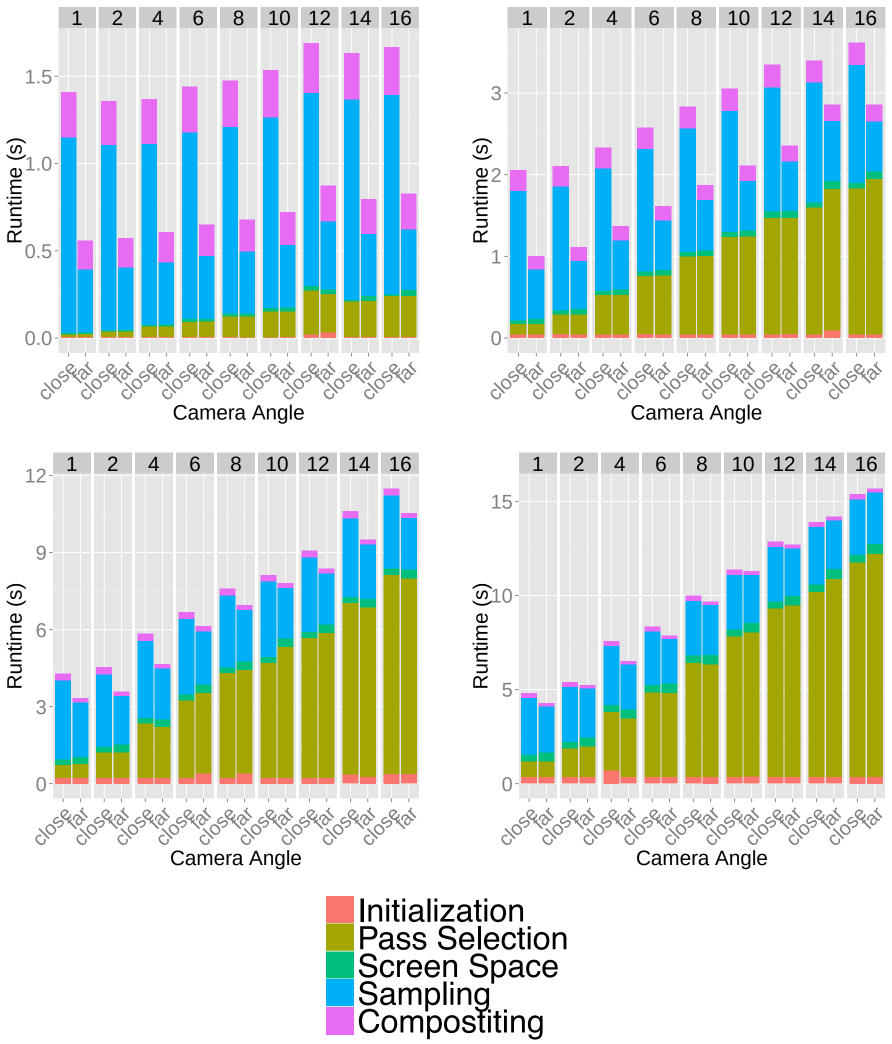

FIGURE 4. Running times for our algorithm. **Enzo-1M** is top left, **Enzo-10M** is top right, **Nek5000** is bottom left, and **Enzo-80M** is bottom right. These tests were run on **CPU1** and renderings from both camera angles were made. Within a figure, the number of passes increases from left to right.

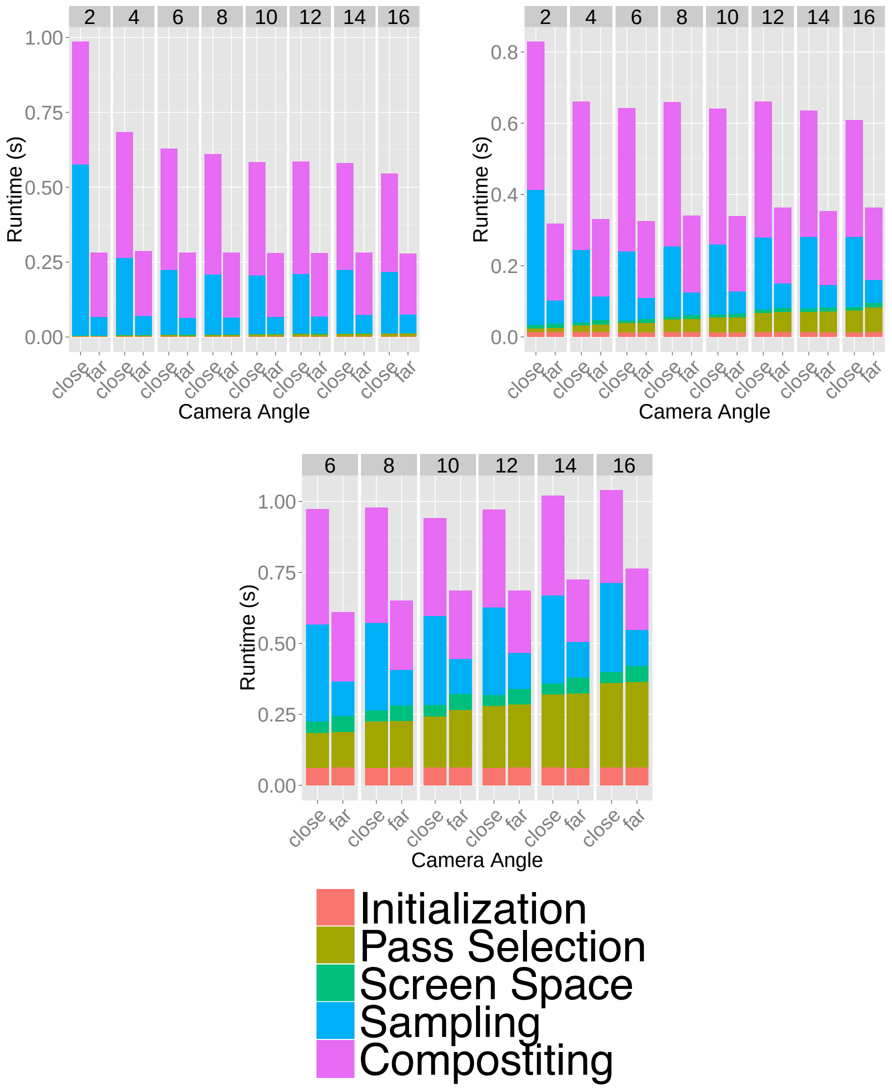

FIGURE 5. Running times for our algorithm. From left to right and top to bottom, **Enzo-1M**, **Enzo-10M**, and **Nek5000**. These tests were run on **GPU1** and renderings from both camera angles were made. Within a figure, the number of passes increases from left to right. The **Enzo-80M** run failed, 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.

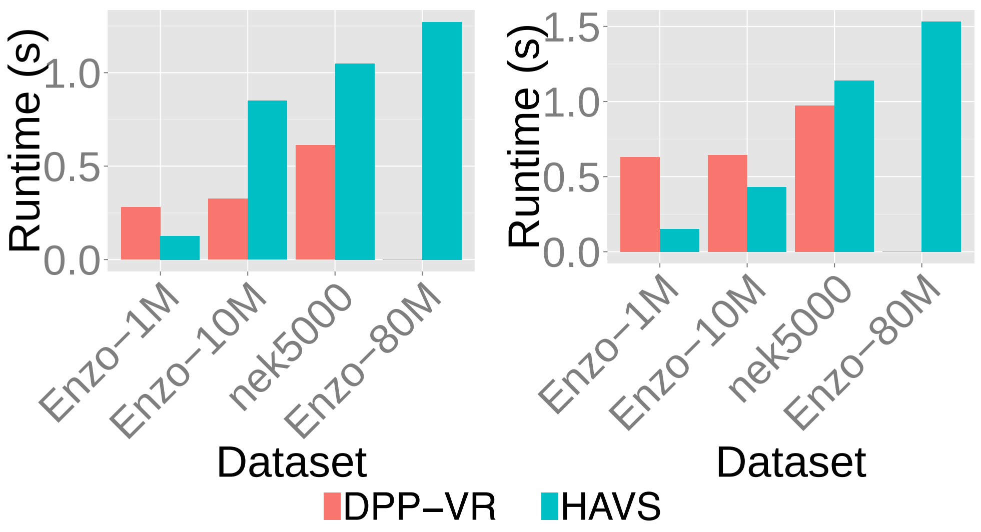

FIGURE 6. Comparing the running times for our algorithm and **HAVS** on **GPU1** for multiple data sets. The left figure is for a zoomed out view, and the right figure is for a close up view.

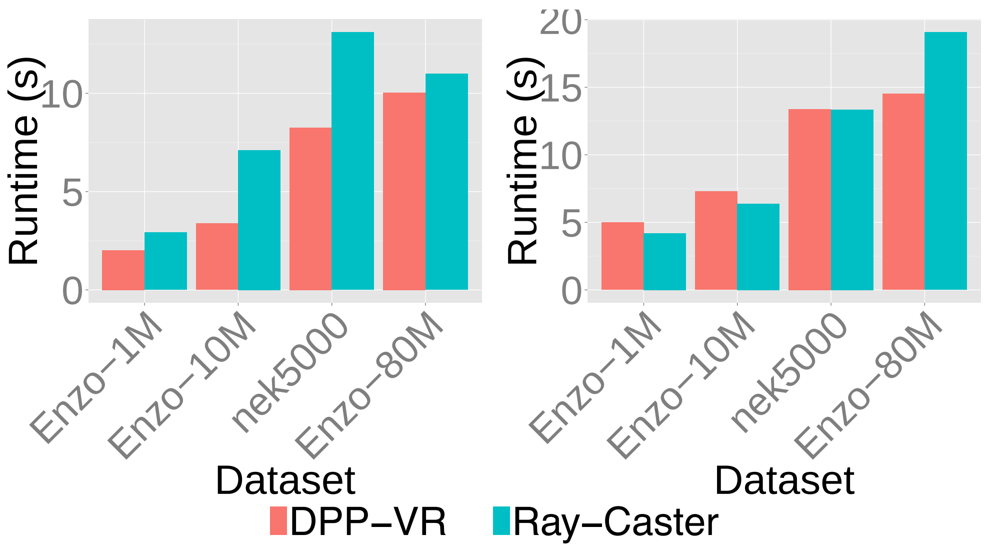

FIGURE 7. Comparing the running times for our algorithm to a **sample-based** on **CPU3** for multiple data sets. The left figure is for a zoomed out view, and the right figure is for a close up view.

CHAPTER IV

IN SITU VISUALIZATION AND ANALYSIS INFRASTRUCTURE

Most of the text in this chapter comes from (74), which was a collaboration between Eric Brugger (LLNL), Hank Childs (UO and LBL), Jim Elliot (AWE), Kevin Griffin (LLNL and UCD), Cyrus Harrison (LLNL), and myself. The requirements section was written by Cyrus Harrison and Eric Brugger. Two of the three integrations were coded by Eric Brugger and Cyrus Harrison and one by myself. The introduction was a collaboration between Hank Childs and myself. Kevin Griffin assisted with data processing scripts. The majority of the infrastructure, including algorithms were implemented by me, and I was the primary contributor to the writing of the paper overall.

In the previous two chapters, we validated the performance of rendering algorithms within a data-parallel framework, and by doing so, we can generate a single performance model for each algorithm that can be applied to multiple architectures. In this chapter, we describe a in situ visualization framework that connects mesh data from several physics simulations to our rendering algorithms. We begin by introducing the problems and background that motivate the creation of a light-weight in situ visualization frame work. Next we describe the system components and requirements, and we discuss our integration experiences.

4.1 Motivation

Like many other institutions, Lawrence Livermore National Laboratory (LLNL) and the Atomic Weapons Establishment (AWE) are moving towards in situ processing to handle the growing divide between computational power and I/O capabilities on modern supercomputers. At these laboratories, the simulation codes are often very complex — meaning multi-physics codes incorporating many libraries — and diverse — e.g., multiple programming languages, data models, etc. In this chapter, we describe our experiences designing, implementing, and evaluating a system to respond to these simulation code teams' requirements. We call this system Strawman. We consider Strawman to be a mini-app, i.e., it is designed to be lightweight and allow us to perform research.

LLNL and AWE are home to more than fifty different simulation code groups. While our requirements are not distilled from conversations with all of these groups, they do represent commonalities from conversations across many of them. Further, in this paper, we present results from integrating Strawman with three prominent proxy simulations: LLNL's Lulesh (7; 65) and Kripke (71; 10) and AWE's CloverLeaf3D (88; 4). Our experiences have led us to believe the system has promising elements for meeting their needs.

The remainder of the chapter describes Strawman and our experiences with it. Section 4.2 describes technologies that we used within Strawman, and Section 4.3 describes the requirements and design of Strawman. Then, in Section 4.4, we describe the results of integrating Strawman into the three proxy simulations.

4.2 Background

Both Paraview Catalyst (51; 84) and VisIt's LibSim (136) provide full-featured tightly-coupled in situ visualization and analysis solutions for simulations running on current HPC platforms. As they are both fully featured and have complex designs, they are not well suited for quick evaluation of new technologies. With Strawman, we have developed a small system that enabled us to explore solutions that may help us evolve the state of the art in tightly-coupled in situ processing for batch use cases on modern and future HPC architectures. We hope to test if solutions are viable enough to be considered for use in larger production systems. To build Strawman, we heavily leveraged three libraries: Conduit, EAVL, and IceT. This section provides background information on these libraries and why they were selected.

Conduit

Conduit (5) is an open source development effort from LLNL that provides an interface for in-core description of scientific data that can be used across C++, C, Python, and Fortran. It is used for data coupling between packages in-core, serialization, and I/O tasks.

Conduit provides a hierarchical object model similar to JSON (9), but differs from JSON and other message exchange formats such as BSON (2), Protocol Buffers (13), and Apache Thrift (1) in a few fundamental ways:

– **Bit-width Style Leaf Types**: To describe scientific data in-core, you must capture the specific details about your data in memory, such as the bits used for numeric primitives, offsets, endianness, etc. To address this Conduit provides bit-width specified scalar and array leaf types. Arrays can be contiguous or strided. The bit-width descriptions follow the style of NumPy (134). Leaf types are always concretely specified in Conduit, however the API also allows you to use native types. For example in C++, when using native *double* data, Conduit will map this to the concrete bit-width type for the current platform, usually Conduit's *float64* type.

– **Separation of Description from Data**: To describe existing in-core data, you cannot use a solution that embeds a description mechanism into the data. Because of this, Conduit separates description metadata from data and does not impose a packed data representation. This design supports zero-copy and allows a Conduit to bring context to chunks of data allocated in different regions of memory.

- **Runtime Focus**: Many existing message exchange formats utilize code generation to implement an object model. Conduit provides a runtime API for describing and accessing data and introspection features that allow you to query and discover the structure of described data. Conduit does not use compile time code generation. This avoids complex build system requirements and makes it easer for components across the simulation ecosystem to share data. This design does have runtime costs and shifts some errors to runtime. These tradeoffs are similar to properties of weakly typed languages like Python. The runtime costs are acceptable for simulation applications where descriptions of the data will be small compared to the large arrays that hold the simulation state.

These features make Conduit a suitable multi-language solution for passing mesh data and commands from a simulation to a visualization pipeline.

Node Level Parallelism

For Strawman, we chose to use EAVL for node level parallelism. For building an in situ visualization application, EAVL is an ideal building block for hybrid parallelism, since it decouples architecturally specific node level parallelism and distributed-memory parallelism. EAVL's data model, as described in (93), is expected to consume 70 to 75 percent less memory because of its use of an underlying array class that can do in-place memory writes. The flexibility of the data model also opens up more memory layouts for zero-copy. Further, EAVL has a rendering infrastructure that includes OpenGL, ray tracing, and volume rendering options (77). These features make EAVL a natural choice for rendering the subset of a distributed mesh on each MPI task in a distributed-memory simulation.

IceT

Producing a coherent image from a distributed render requires parallel compositing. IceT (95; 98) provides a scalable solution for compositing using MPI on distributed-memory parallel systems. It implements a sort-last compositing with a suite of efficient parallel partitioning and reduction strategies (94). We selected IceT as our compositing solution for this effort since it has been used at large scales for years in production visualization tools including Paraview, VisIt, and their in situ offerings.

4.3 System Overview

This section provides details on the requirements and design of Strawman.

System Requirements

To guide the development of our mini-app, we selected a set of important batch in situ visualization and analysis requirements extracted from our interactions and experiences with simulation code teams. Here are our 11 requirements broken out into three broader categories:

- **Support a diverse set of simulations on many-core architectures.**
  - **R1**: Support execution on many-core architectures.
  - **R2**: Support usage within a batch environment (i.e., no simulation user involvement once the simulation has begun running).

    * **R3**: Support the four most common languages used by simulation code teams: C, C++, Python, and Fortran.
    * **R4**: Support for multiple data models, including uniform, rectilinear, and unstructured grids.
  - **Provide a streamlined interface to improve usability.**
    * **R5**: Provide straight forward data ownership semantics between simulation routines and visualization and analysis routines
    * **R6**: Provide a low-barrier to entry with respect to developer time for integration.
    * **R7**: Ease-of-use in terms of directing visualization and analysis actions to occur during runtime.
    * **R8**: Ease-of-use in terms of consuming visualization results, including delivery mechanisms both for images on a file system and for streaming to a web browser.
  - **Minimize the resource impacts on host simulations.**
    * **R9**: Synchronous in situ processing, meaning that visualization and analysis routines can directly access the memory of a simulation code.
    * **R10**: Efficient execution times that do not significantly slow down overall simulation time.
    * **R11**: Minimal memory usage, including zero-copy usage when bringing data from simulation routines to visualization and analysis routines.

In the following sections, we describe how Strawman's design and our integration experiences with three proxy simulations demonstrate how Strawman addresses these requirements.

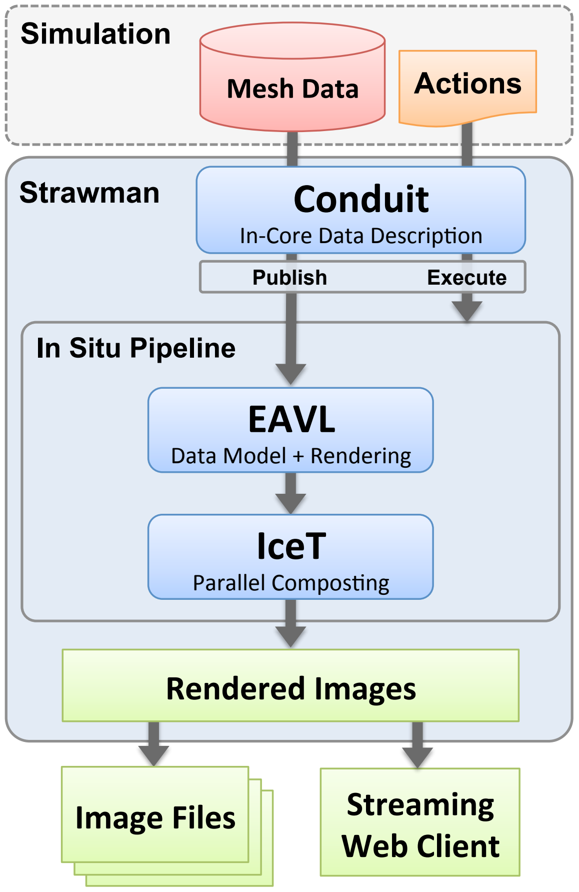

FIGURE 8. A diagram of the Strawman system architecture.

System Design

Figure 8 shows how Strawman interfaces with a simulation and uses Conduit, EAVL, and IceT to produce in situ visualizations. Starting from the top, simulation mesh data and the desired in situ actions are described using Conduit and passed into the Strawman API. (Details about the API and how Conduit is used are described in the Strawman Interface subsection.) The visualization actions are then realized in a simple in situ pipeline that leverages EAVL for rendering and IceT for compositing. This process is described in the Strawman Pipeline subsection. Finally, the results are saved to image files on disk or streamed to a web browser via a WebSocket as detailed in the last subsection.

Strawman Interface

The Strawman API consists of only a few function calls. The complexity of describing options, mesh data, and actions is handled by dynamic and hierarchal features of Conduit's Node class. Conduit supports all of the languages in **R3**, so the function calls in Strawman can easily be exposed in these languages.

Strawman is first initialized with a small set of parameters: a MPI communicator for the parallel case and optional settings for WebSocket streaming.

Second, mesh data structures are described using Conduit Node instances following a set of mesh description conventions. For Strawman, we are not creating a new mesh data model. Instead we provide a set of conventions to describe mesh data using Conduit that can be easily used with a wide set of existing concrete data models. The conventions we developed to describe meshes for our interface were informed by a survey of several mesh-related APIs, including: ADIOS (83), Damaris (48), EAVL, MFEM (12), Silo (14), VTK(121), and Xdmf(15) to support **R4**. The simulation owns the Conduit Nodes describing mesh data and the Nodes hold information on if the data is externally owned (zero-copied) or allocated by Conduit. These ownership semantics support **R5**. After description, the mesh data is published to Strawman via *Publish*. In cases where simulation mesh data structures are not reallocated and match EAVL's data model, the mesh data only needs to be published once during a simulation run, supporting **R11**.

Next, the set of desired in situ actions are specified using Conduit Node instances and passed to Strawman via *Execute*. The simplicity of the actions interface supports **R7**.

These actions are then translated into a simple pipeline and executed as described in the next subsection.

In Situ Pipeline

The in situ pipeline converts the mesh data described using Conduit and passed via *Publish* into concrete EAVL data sets and then uses the action descriptions passed via *Execute* to construct a simple rendering pipeline. The pipeline uses EAVL's filters and rendering infrastructure, including surface, volume, and ray-casting rendering techniques, to render the subsets of the data owned by each MPI task. As discussed in Section 4.2, algorithms in EAVL are designed to run on many-core architectures, supporting **R1** and EAVL's data model has zero-copy features supporting **R9** and **R11**. We made enhancements to EAVL to support rendering in a distributed-memory parallel setting. For example, we added visibility ordering calculations to support volume rendering image compositing and data extent reductions in order to apply consistent color tables across MPI tasks. IceT is then used to composite these renders using MPI to create a final image.

Presentation of Results

To satisfy **R8**, Strawman saves rendered images to PNG files, and includes a lightweight embedded web server (3) that can be used to optionally stream results over a WebSocket. To allow a simulation user to view the rendered images via a web browser, it also provides a WebSocket client that displays images as they are streamed from the WebSocket connection.

4.4 Results

To test in situ integration with a range of simulations and mesh types, we selected three proxies for larger production multi-physics simulations: LULESH, Kripke, and CloverLeaf3D. LULESH is written in C++ and uses a Lagrangian approach on a 3D unstructured hex mesh to simulate shock hydrodynamics. Kripke is written in C++ and uses a deterministic discrete ordinates solver on a 3D uniform mesh to simulate particle transport. CloverLeaf3D is written in Fortran and solves the compressible Euler equations on a 3D rectilinear mesh to simulate hydrodynamics. Figure 9 shows images produced from the integrations with each of the simulations.

Integration Experiences

To evaluate the barrier of entry **R6** and ease-of-use **R7** requirements, we present the lines of code required to integrate Strawman into each of these proxy simulations to produce the simplest and most common visualization result, a Pseudocolor rendering.

As outlined in Section 4.3, integrating Strawman into a simulation can be broken down into three steps: describing simulation mesh data, describing the set of in situ actions for Strawman to perform, and calling the Strawman API to *Publish* the mesh data and *Execute* the specified actions. To demonstrate these three steps, we show code fragments for each step from our LULESH Strawman integration.

Listing 4.1 Describing the simulation data.

```
conduit::Node data;
data["state/time"].set_external(&time);
data["state/cycle"].set_external(&cycle);
data["state/domain"] = my_mpi_rank;
data["coords/type"] = "explicit";
data["coords/x"].set_external(x);
data["coords/y"].set_external(y);
data["coords/z"].set_external(z);
data["topology/type"] = "unstructured";
data["topology/coordset"] = "coords";
data["topology/elements/shape"] = "hexs";
data["topology/elements/connectivity"].set_external(nodelist);
data["fields/e/association"] = "element";
data["fields/e/type"] = "scalar";
data["fields/e/values"].set_external(e);
```

Listing 4.2 Describing the actions to perform.

```
conduit::Node actions;
conduit::Node &add = actions.append();
add["action"] = "AddPlot";
add["var"]    = "p";
conduit::Node &draw = actions.append();
draw["action"] = "DrawPlots";
conduit::Node &save = actions.append();
```

```
char output_filename[30];
sprintf(output_filename,"image%04d",cycle);
save["action"] = "SaveImage";
save["fileName"] = output_filename;
save["format"] = "png";
save["width"] = 1024;
save["height"] = 1024;
```

Listing 4.3 Calling the Strawman API to publish data and execute the specified actions.

```
Strawman strawman;
Node options;
options["mpi_comm"] = mpi_comm_handle(MPI_COMM_WORLD);
strawman.Open(options);
strawman.Publish(data);
strawman.Execute(actions);
strawman.Close();
```

In the case of LULESH, the simulation data model exactly matches EAVL's data model and all the data can be passed zero-copy to Strawman. Because of this, LULESH requires the least amount of Strawman integration code. In the case of Kripke, it was necessary to copy the field data since the array ordering did not match EAVL's data model. In the case of CloverLeaf3D, it was necessary to copy the coordinate and field data to remove the embedded ghost zones, which Strawman currently does not support. The difference in lines of code required between the proxy simulations is due entirely to code related to modifying the simulation data to match EAVL's data model.

TABLE 10. Lines of code needed to instrument the three selected proxy apps to produce a PNG file with a Pseudocolor rendering of a scalar variable.

	LULESH	Kripke	CloverLeaf3D
Data Description	15 loc	21 loc	39 loc
Action Descriptions	14 loc	14 loc	14 loc
Strawman API Calls	7 loc	7 loc	9 loc

TABLE 11. Simulation burden from visualization for large runs using 4096 cores on 256 nodes of LLNL's Cab machine using various renderers. Times are listed in average seconds per cycle for both the visualization and simulation. All images were generated at a resolution of 1024^2.

	Vis	Sim
CloverLeaf3D 4B Cells (Ray Tracing)	1.68s	5.77s
Kripke 4B Cells (OSMesa)	3.80s	16.55s
LULESH 8B Cells (Vol. Ren.)	30.85s	12.62s

Looking at the complexity of listings 4.2 and 4.3 we assert it is straight forward to instruct Strawman on the actions to perform. Listing 4.1 demonstrates the most complex step, describing simulation mesh data. This resembles conventional transformations simulation developers are already familiar with from supporting post processed visualization and analysis tools.

Based on the lines of code to instrument the three simulation codes and the complexity of the code fragments we believe we have satisfied **R6** and **R7**.

Performance

To evaluate the performance of our infrastructure, we ran Strawman with batch jobs using all three proxy simulations on 4096 cores with mesh sizes ranging between 4 billion and 8 billion cells, satisfying **R2**. Table 11 shows the average time per cycle of both visualization and simulation. We will explore the performance characteristics in more depth in future work to understand how Strawman addresses **R10**.

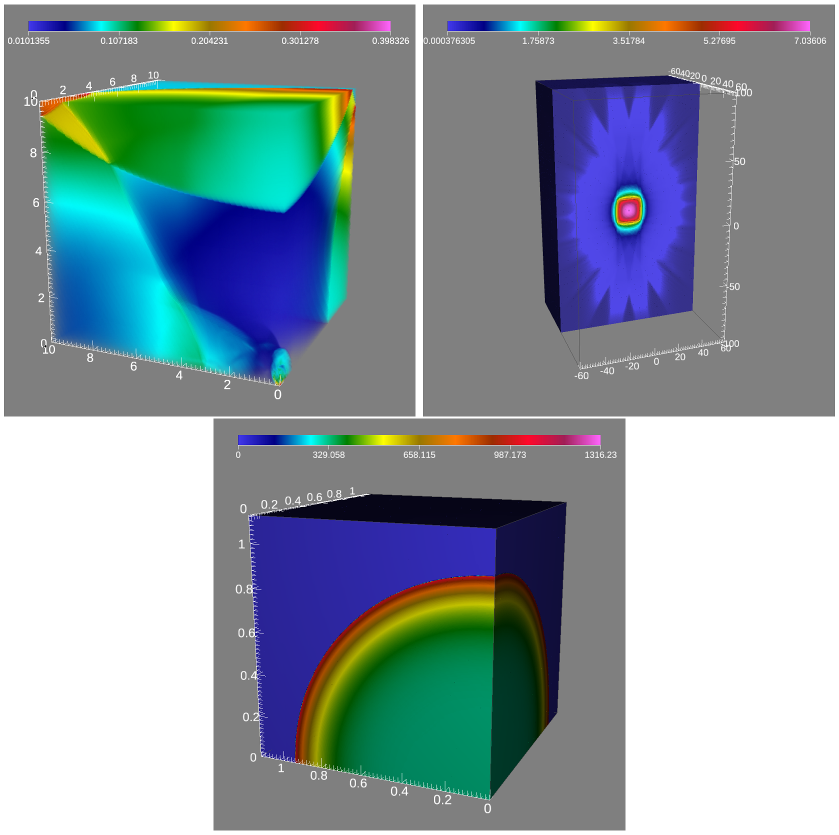

FIGURE 9. Images produced using Strawman from the three integrations. From left to right and from top to bottom, a volume rendered image from CloverLeaf3D, a ray traced image from Kripke, and a rasterized image (OSMesa) from LULESH.

4.5 Summary

We described our mini-app for in situ, Strawman, which enabled us to explore a system meeting our stakeholders requirements: modern architectures, streamlined user interface, and minimal burden on the simulation. The results were promising, especially in terms of streamlined user interface, although we believe more effort is needed to reduce the time to carry out visualization operations.

CHAPTER V

IN SITU PERFORMANCE MODELING

Most of the work in this chapter comes from (75), which was a collaboration between Cyrus Harrison (LLNL), James Kress (UO and ORNL), Dave Pugmire (ORNL), Jemery Meredith (ORNL), Hank Childs (UO and LBL), and myself (UO and LLNL). The introduction was a collaboration between Hank Childs and myself. The related work section was written by Jeremy Meredith and myself. The methodology, performance models, study overview and model evaluation sections were written by me, with edits from the other authors. The majority of the software infrastructure and algorithms were written by me, with build system and porting assisted by James Kress, Dave Pugmire and Cyrus Harrison. The final two sections were a collaboration between myself and the other authors.

The previous chapters introduced the problem and built the foundation for the work in this chapter. Chapters II and Chapter III showed that we are able to construct rendering algorithms using a data-parallel framework that are competitive with community standards, validating the data-parallel approach. With a single implementation, we can build a performance model that applies to both the CPU and GPU architectures. In Chapter IV, we constructed an in situ infrastructure that connects the data from three simulations to the data-parallel framework. In this chapter, we describe our performance model creation methodology and performance models for ray tracing, rasterization, and volume rendering. Finally, we conduct an extensive performance study, create a mapping between simulation parameters and model inputs, and answer several important in situ viability questions.

5.1 Introduction

In Chapter I, we introduced the motivation and importance of answering the in situ rendering feasibility question: is it possible to perform X_1 visualization tasks while devoting no more than X_2 time to these tasks? We approached the issue of rendering feasibility through performance modeling. We started by implementing three rendering algorithms: rasterization, ray-tracing, and volume rendering. We implemented them in the VTK-m framework (101), which allows a single implementation to run efficiently over multiple architectures. Further, a data-parallel implementation allows us to study the same performance model across architectures. We then ran 1,350 experiments to collect performance data, varying over rendering algorithm, architecture (CPU and GPU), concurrency, simulation code, and data size. Next we established performance models for each rendering technique using linear regression and evaluated these models using standard regression metrics and k-fold cross validation. We expanded our study to a leading-edge supercomputer (ORNL's Titan) and predicted performance at large scale. Once the performance models were validated, we used them to answer rendering feasibility questions. We believe the result is a novel study, both in our application of performance modeling to visualization and in the results we obtain informing important *in situ* feasibility questions.

This chapter is organized as follows:

- Section 5.2 surveys related work.
- Section 5.3 describes our methodology for establishing performance models.
- Section 5.4 gives an overview of our study: rendering configurations tested, software used, and measurements taken.
- Section 5.5 defines a performance model for single node rendering.

– Section 5.6 extends the single node model to cover parallel rendering.

– Section 5.7 describes our evaluation using a leading-edge supercomputer.

– Section 5.8 describes how to map a given rendering configuration to our performance model's input variables.

– Section 5.9 explores rendering feasibility questions using our performance model.

5.2 Related Work

Rendering

There are three major types of parallel rendering (45): sort-first, sort-middle, and sort-last. Sort-first rendering parallelizes over pixels within an image, and distributes geometry so that every processor has the data necessary to generate its portion of the image. Sort-middle rendering follows a similar approach, but transforms the geometry into screen space before distributing. Sort-last rendering parallelizes over data, having each processor create a sub-image that contains a rendering of its own data, and then compositing the sub-images together, i.e., using depth information and parallel communication to create the final image.

Parallel rendering on supercomputers is typically done with the sort-last approach (57). With this approach, there are two distinct phases: local rendering and compositing. Local rendering requires no parallel coordination, while compositing typically does not begin until all MPI tasks have finished their local rendering.

Each of our rendering algorithms were implemented in VTK-m (101). VTK-m makes use of data parallel primitives, an abstraction that enables algorithms to be written one time, but used on multiple architectures (CPU and GPU).

We derived our ray-tracer from the implementation in EAVL (91). An evaluation of the EAVL ray-tracer (73) found that it generally performed within a factor of two of Intel's Embree (138) and NVIDIA's OptiX (108).

Our volume renderer was developed for this study and was designed to work with structured grids. Previous studies of data parallel primitives-based volume renderers (77; 122) have found that the data parallel primitive approach is competitive with hardware-specific versions.

Our rasterizer was also developed for this project using VTK-m. Based on recent successes with rendering and portable performance, we felt that having a single implementation over multiple architectures was better than using architecture-specific rasterizers, such as implementations of the OpenGL(8) standard. Examples include CPU implementations (e.g., Mesa (11) and OpenSWR (6)) and GPU implementations from hardware vendors. We note that our rasterizer is not the first CUDA-based software rasterizer on a GPU (72) (with CUDA being a back-end option for our data parallel primitives).

Although we implemented our own renderers, our study incorporated an existing parallel image compositor, IceT (98). IceT has been shown to be scalable up to tens of thousands of MPI tasks, and contains implementations of many compositing algorithms, including direct send (103), binary swap (86), and Radix-k (111; 110), and we chose to use Radix-k.

Several previous studies have looked at rendering performance at massive scale (98; 86; 111; 61). Our study differs from these previous works in that we are considering multiple rendering methods and that we have developed a performance model that allows us to answer feasibility questions.

Performance Modeling

Simulators take a fine-grain approach to performance modeling, by attempting to model the behavior of underlying features of the target system, from the microprocessor instruction level to the network messaging level. Examples include SST, ROSS, and GEMS (117; 63; 37; 89). At the other end of the spectrum, analytical approaches are methods for generating symbolic equations describing an algorithm or program which can quickly generate performance predictions for known input parameters. Examples of analytical models and tools include BSP, LogP, and Aspen (130; 46; 125).

Performance models can use data from either known hardware parameters (e.g., peak floating point or messaging rates) or from empirical measurements such as performance counters and execution time. When the latter are combined with analytical performance models, the derived models are termed "semi-empirical" (58). Our approach falls into this category, generating an *a priori* analytical performance model based on known attributes of a specific algorithm and fitting with observed data.

Performance Modeling and Visualization

There are relatively few rendering works that incorporate performance models. Cohen et al. (44) used an architecture-specific constant to evaluate a cost model, maximizing performance by switching between polygon and point rendering based on an evaluation. Similarly, Tack et al. (129) used a fine-grain performance model to estimates run-time cost per triangle, based on the steps within a software-based rasterization pipeline, on resource constrained mobile devices. Their system ran a set of tests to estimate constants in their model, then adjusted rendering for the given device. While these studies share the elements of performance modeling and rendering, their goals and rendering tasks differ from our own.

Bowman et al. (31), described a performance model for a visualization pipeline, and included rendering in their model. They modeled rendering by multiplying the number of triangles to render by a constant, which was determined from the average time to render different amounts of triangles using a GPU rasterization pipeline. Finally, Rizzi et al. (116) described a distributed GPU ray casting performance model for volume rendering by enumerating each component, including network bandwidth and GPU transfer times, and their goal was to help to provide guidance in the design of future machines. While both were successfully able to model their use cases, our study is different in that we are focused on in situ rendering, which allows us to go beyond their studies in significant ways: by considering multiple rendering techniques, by evaluating varied rendering configurations, by establishing statistical bases for our models, by considering multiple parallel architectures, and by exploring the feasibility regions implied by our models.

5.3 Methodology

In this section, we introduce the methodology we used to develop the performance models for the three rendering techniques (ray-tracing, rasterization, and volume rendering).

Our goal is to create linear models that relate key variables about data and rendering configurations to the run-time performance of each renderer. We created general linear models based on the nature of the algorithms used in each renderer. To allow our general linear models to be used for estimates on specific hardware architectures, we ran a study using a range of representative configurations and fit model coefficients to this data using multiple linear regression. We then used several methods to evaluate our models.

Model Input Variables

The first modeling step was to select the key input variables of the rendering process. Several factors influence the run-time of rendering, but at a high level, the two essential variables are the number of objects (e.g., cells or triangles) and the number of pixels rendered. For each of the renderers we studied, we started with initial linear models that relate these two variables to the algorithmic complexity of their rendering pipelines.

These models captured the run-times of the different pipeline stages for each type of renderer. While rendering algorithms are generally characterized by the iteration target of their outer loop — object-order algorithms (e.g., rasterization) loop over objects, and image-order algorithms (e.g., ray casting) loop over pixels — this basic characterization overlooks the fact that rendering implementations may actually use pipelines with multiple steps that mix object-order and image-order algorithms. For example, ray-tracing is an image-order algorithm that uses an acceleration structure constructed with an object-order algorithm.

Directly using all of the individual pipeline steps as components in our models could create too many degrees of freedom. To avoid this, we grouped related pipeline steps into higher-level stages and related these stages to the input variables to create the components of our models. For example, our ray-tracing performance models considers the acceleration structure build stage as a single model component, although it actually consists of five smaller steps.

To capture architecture-specific details, we added experimentally-obtained coefficients for each of our linear model components.

We also selected variables that are quantitative proxies to important user settings (e.g. data set size, output image sizes, etc.), which influence rendering performance.

These variables include the number of objects in view of the camera, the actual amount of pixels that need to be rendered for the view, and view specific counters for rasterization and volume rendering. The values for these proxies are not known exactly *a priori*, but can be estimated from a user's specific rendering settings. When we fit coefficients to capture architecture-specific details, we used measured data points for these variables to create more accurate models.

In summary, the input variables we used for our models are:

- General Input Variables
  - Objects (O): the number of cells or triangles to render
  - Active Pixels (AP): the number of pixels that are updated as a result of rendering
  - c_i: empirical constants to capture architecture-specific details
- View-Specific Variables for Rasterization
  - Visible Objects (VO): the number of cells or triangles to render visible to the camera
  - Pixels Per Triangle (PPT): the average number of pixels considered per triangle
- View-Specific Variables for Volume Rendering
  - Samples Per Ray (SPR): the average number of samples along a ray that are inside the data set
  - Cells Spanned (CS): the maximum number of cells that a ray can span

Study Parameters

To obtain data to fit model coefficients and evaluate our models, we conducted a study to gather run-time performance data for each renderer. We tested our models using a range of configurations that represent *in situ* use cases.

We selected configurations exploring parameters that, from a user's perspective, influence the number of objects and pixels relevant to rendering. These parameters include the type of simulation data, simulation data set size, desired image resolution, and the total number of MPI tasks.

We used multiple physics simulation applications to explore performance with structured and unstructured meshes. We also adjusted the size of the simulations (i.e. the total number of cells in the discretization) to vary the number of objects rendered. We varied the the total number of MPI tasks to change the number of active pixels on each task and to test distributed-memory image compositing run-times.

Model Fitting and Evaluation

We used multiple linear regression to fit coefficients that allow us to use our models to estimate rendering run-times for a specific architecture. We used the R programming environment (114) and many of its packages (137; 23; 135; 87; 118) to compute and evaluate the regressions and to create plots.

To evaluate our models, we examined the metrics of multiple R-squared, residual standard deviation, the values of our model coefficients, and the average relative error. Multiple R-squared effectively reports percentage of the variance of the run-time that is captured by a model. The residual standard deviation is an indicator of how well a model fits the data, where a value of zero means the model perfectly fits the data. For rendering algorithms, no input variables should have a negative linear relationship to run-time, and so the presence of regression coefficients less than zero typically indicates that a model is not valid or some feature of the hardware architecture is compensating.

To further evaluate our models, we used correlation analysis and k-fold cross validation. Correlation analysis gave us a basic view of how the input variables are related to the measured run-times. This helped us check our linear modeling assumptions and quickly screen for potential implementation issues. K-fold cross validation systematically fits model coefficients to subsets of the available data and tests how well these fitted models represent the remaining data. We used k-fold cross validation to check that we were not overfitting our models to data and as another measure of the variance of our models.

5.4 Study Overview

Software Implementation

We implemented our rendering algorithms using VTK-m (101), which allows a single implementation to execute on multiple architectures. Currently, VTK-m supports two back-ends: TBB and CUDA. VTK-m algorithms are composed of a series of data-parallel primitives such as map, scan (also known as prefix sum), and reduce. Each back-end implements supported data-parallel primitives that are optimized for the architecture, which enables portable performance of algorithms with different back-ends. The majority of the rendering algorithm implementations use a map operation to loop over pixels (e.g., volume rendering and ray tracing) or objects (e.g., rasterization).

For the *in situ* infrastructure, we used Strawman (74), a lightweight *in situ* framework that includes integrations with three physics simulation codes. Strawman uses a modular infrastructure that allowed us to implement a visualization pipeline using the VTK-m rendering infrastructure. For the sort-last rendering, IceT (98) was used to composite the images generated by each task.

Study Options

Our study was designed to test and validate the performance models under a wide variety of rendering conditions. In order to examine a representative space of rendering configurations, we varied the following factors:

- Architecture (2 options)
- Rendering Algorithm (3 options)
- Simulation Code (3 options)
- MPI Tasks (up to 7 options, depending on architecture)
- Image Resolution (many options)
- Data Size (many options)

We discuss each of the factors in the following subsections.

Architectures

Our study was performed on LLNL's Surface cluster. Each node contains two Intel Xeon E5-2670 (Sandy Bridge) and two NVIDIA K40m GPUs. We used the following configurations:

- CPU1: 1 MPI task per node
    - Simulation code: 16 OpenMP threads
    - Strawman: 16 TBB threads
- GPU1: 2 MPI tasks per node
    - Simulation code: 8 OpenMP threads per task

    * Strawman: 1 K40m per task

Rendering Algorithms

For the study, we implemented the following algorithms using VTK-m:

- Ray Tracing: a ray tracer that only casts primary rays and generates images similar to rasterization.
- Structured Volume Rendering: a ray caster for regular grids.
- Rasterization: an implementation based on sampling using barycentric coordinates.

Simulation Codes

Our study used the three physics simulation codes included with Strawman. Each of these codes are "proxy apps," meaning they are lightweight and meant to be accessible for computer science research, and figure 10 shows images from each of the simulations. The codes are:

- Lulesh: a Lagrangian shock hydrodynamics simulation on a 3D unstructured mesh
- Kripke: a deterministic neutron transport solver on a 3D structured mesh
- Cloverleaf3D: an Euler hydrodynamics simulation on a 3D structured mesh

MPI Tasks

We varied the number of MPI tasks to study the effects of active pixels per task as concurrency increased. The number of MPI tasks for each architecture were:

- CPU1: 1, 2, 4, 8, 16, and 32 tasks
- GPU1: 1, 2, 4, 8, 16, 32, and 64 tasks

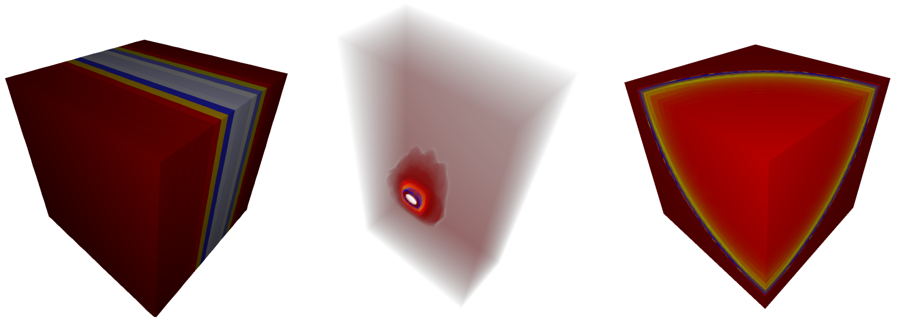

FIGURE 10. Images from each of the simulation codes. From left to right, a rasterized image from Cloverleaf3D, a volume rendering from Kripke, and a ray-traced image from Lulesh.

Image Resolution

There are many possible choices for image resolution, and we wanted our models to predict values for the widest range of possible combinations that are appropriate for tightly-coupled *in situ*. For each test, we used stratified sampling, similar to Latin hypercube sampling, to choose image resolutions between 512^2 and 2880^2, which is equal to the total number of pixels in a standard 4K image (3840×2160). The data sets were cubic, so we chose to use square images to minimize the amount of unused pixels.

Data Set Size

As with image resolution, there are many possible choices of data set size, and using stratified sampling, we chose data set sizes ranging from 128^3 to 320^3 cells per node. We consider these sizes to be representative of the per node cell count of a large multi-physics simulation. The problem size per task was weakly scaled, so that the total number of cells over all processors increased proportionally to the number of MPI tasks. For the rasterizer and ray tracer, we used an external faces operation to generate triangles on each MPI task, and all tasks were assigned equal amounts of geometry. We used external faces to control the amount of geometry, but we believe it is representative of other visualization operations such as slicing and contouring, which takes $O(N^3)$ cells and creates $O(N^2)$ geometry.

Tests and Measurements

For the study, we ran the cross product of the study variables with 25 combinations on data set size and image resolution, and while not all combinations made sense (e.g., structured volume renderer with an unstructured data set), we ran 1350 total tests. Each test consisted of running the simulation code for 10 cycles, and we generated an image for each cycle. Once all of the data were gathered, we discarded the first data point because it proved to be unreliable. We gather data for each frame and for each rank. Since rendering is only as fast as the slowest MPI task, we only considered the slowest MPI task from each of the 1350 tests. Further, for this slowest MPI task, we took its average time over the last nine cycles.

Finally, although we wanted to study rendering with GPUs, the simulation codes we studied only ran on CPUs. Our solution was to run the simulation code on the CPU, and then transfer its data to the GPU. Our measurements in this case only reflect the rendering times (i.e., after the data was transferred), which we feel would be representative of scenarios where simulations run directly on the GPU. While we did not consider it in our study, we feel that data transfer time could be added to our models without much effort.

5.5 Single Node Performance Model

This first subsection defines the models we used for each rendering type, while the second subsection describes our results evaluating our models' accuracy.

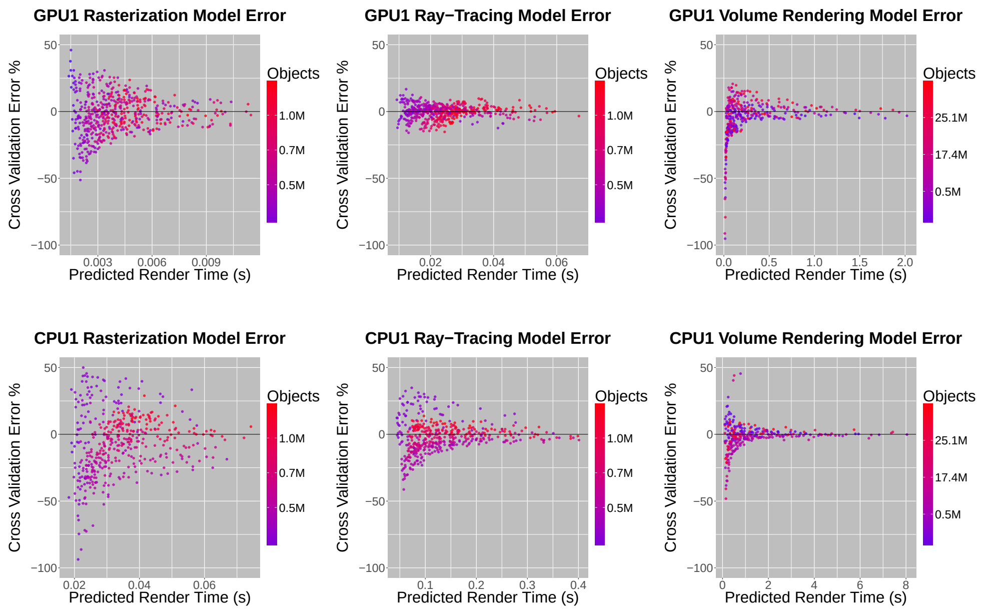

FIGURE 11. The 3-fold cross-validation error plots for all six models. Renderers are paired in columns, while architectures go along rows (GPU1 on top, CPU1 on bottom). For each fold, the models are trained with a partial set of the data, then the models predict the total render time with the remaining data points. Error percentage is calculated as $100 \times (time_{total} - time_{predicted})/time_{total}$. A value of zero represents a perfect prediction. Each test is colored by the number of objects it renders (i.e., triangles for rasterization and ray-tracing and cells for volume rendering).

Model Definition

We describe our performance models for each of the three rendering techniques. Each of the three models use input variables that describe the rendering task to predict the corresponding execution time. The models described in this section are the final models we developed for each rendering technique, realized after exploring many alternative options. They represent the best fit to the data from our experiments, evaluated using methodology discussed in Section 5.3.

The models describe only the time to do local rendering (i.e., a single MPI task rendering its own data); this model is extended in Section 5.6 to include parallel rendering. When running in parallel (i.e., more than one MPI task), each node has its own value for each of the input variables. For this phase of our model, we are only concerned with the process of making a sub-image (which is embarrassingly parallel) and not with the compositing (which does require parallel communication). So our model can predict the individual execution times for each of the MPI tasks, given each of their individual input variables.

Ray-Tracing

Our performance model for ray-tracing execution time (T) is:

$$T_{RT} = (c_0 * O + c_1) + (c_2 * (AP * log_2(O)) + c_3 * AP + c_4) \tag{5.1}$$

The intuition behind this model is that ray-tracing consists of three parts: building the acceleration structure ($c_0 * O$), tracing the rays ($c_1 * (AP * log_2(O))$), and shading the intersection points ($c_2 * AP$). We timed each of these three phases separately, which enabled us to solve for constants c_i for each of the three separate linear regression models. Finally, we separated the term first term (i.e.,$c_0 * O + c_1$), from the rest of the terms. This allows us to consider cases where the BVH acceleration structure is built once, and then multiple renderings use that BVH afterwards.

There are multiple ways to implement ray-tracing, and our model matches our particular implementation. Specifically, our acceleration structure is a variant of a Linear Bounding Volume Hierarchy (LBVH) (66), which has a build-time complexity of $O(n)$. Tracing rays is a function of the number of pixels (AP), and traversal through the acceleration structure, which is a binary tree, is a function of the number of objects (i.e., $log_2(O)$). It is worth noting that the shape of the binary tree and average traversal depth is related to the spatial distribution of the objects and camera position. Finally, the shading term is a function of the number of pixels.

Rasterization

Our performance model for rasterization execution time is:

$$T_{RAST} = c_0 * O + c_1 * (VO * PPT) + c_2 \tag{5.2}$$

The intuition behind this model is that rasterization consists of two parts: culling objects that are not visible ($c_0 * O$), and then rasterizing the visible objects ($c_1 * (VO * PPT)$).

Volume Rendering

Our performance model for structured volume rendering is:

$$T_{VR} = c_0 * (AP * CS) + c_1 * (AP * SPR) + c_2 \tag{5.3}$$

The four main tasks in volume rendering are locating cells that contain sample points, loading scalar values into memory, interpolating scalars, and compositing colors. Our implementation consists of a single component that samples by casting rays into the volume, but the four tasks fall into two distinct groupings: cell frequency ($AP * CS$) calculations and sample frequency ($AP * SPR$) calculations. The performance model groups the calculations into terms of the same frequency.

Cell frequency calculations involve locating scalar values for a cell and loading them into memory. The total amount of cell locations is the number of active pixels (AP) multiplied by the number of cells that span the ray's path (CS) (i.e., ($AP * CS$)). Additionally, our implementation calculates interpolation constants that are re-used when sampling the same cell multiple times.

TABLE 12. R^2 values for our performance models. R^2 captures the percentage of variance in run-time captured by the model and is an indicator of model fit.

	R^2	
Renderer	CPU1	GPU1
Ray Tracing	0.9695	0.9728
Volume	0.9981	0.9978
Rasterization	0.6677	0.9425

TABLE 13. Model accuracy summary from the result of the 3-fold cross validation analysis for our six models (three rendering techniques times two architectures). For each fold, two thirds of the data is used to train the model and the remaining one third is used to test the prediction. Each percentage represents the number of predicted values, from all folds, that are within the corresponding error percentage.

	CPU1				
Renderer	50%	25%	10%	5%	Average %
Ray Tracing	100.0	93.3	65.6	35.6	9.6
Volume	100.0	95.4	81.3	63.0	6.3
Rasterization	96.0	70.2	35.8	18.4	19.1

	GPU1				
Renderer	50%	25%	10%	5%	Average %
Ray Tracing	100.0	100.0	94.8	71.7	3.8
Volume	97.8	92.8	64.7	46.0	10.0
Rasterization	99.8	90.2	53.2	26.8	11.5

Sample frequency calculations consist of interpolation and compositing. This happens exactly $AP * SPR$ times, which is equal to the total number of samples for the render.

Model Evaluation

Table 12 shows the R^2 values for the six architecture-rendering pairings. In our case, five of the six models had values over 0.94, which indicates a good fit.

Our 3-fold cross-validation study involved creating a model using a subset of the data, and then evaluating the model with the remainder of the data to prevent overfitting. The results of this study are plotted in Figure 11, which shows the error for each test, and is summarized in Table 13, which shows model accuracy over all tests. The model errors in Figure 11 shows that we are increasingly accurate as render time goes up. These errors are also seen in Table 13, which shows the average error percentage for our models were at worst 19.1% incorrect on average, and at best 3.8% incorrect on average. In terms of model accuracy, Table 13 shows that all six models were predictive within 50% error at a very high rate — the worst model was still able to predict within 50% error for 96% of the time, and for three of the models, we were within 50% error all of the time.

The model that had the poorest fit was CPU1+rasterization. This reflects the high variability in observed run-time in its tests. Our reported values were an average of nine cycles; for every one of our CPU1+rasterization experiments, at least one of the nine tests was more than 10% above or below the test average. In total, 24% of the predictions fell within the measured variation in the run-time. We believe that the higher variation was due to our implementation which used hardware specific features, but in the future, we would like verify if the performance of OpenSWR has the same characteristics as our implementation. That said, despite this low R^2 score, 70% of the predictions were within 25% of the measured values and 96% were within 50%.

The x-axis for Figure 11 is predicted render time, and it is clearly visible that our tests skew toward shorter run times. This does not represent bias in our tests, but rather the effects of increased concurrency, which then reduces the number of active pixels per MPI task. For all models, it is these lower render times that contribute the majority of the error occurs, where both timer resolution and memory hierarchy factors more heavily influence execution time.

One clear example of the influence of the memory hierarchy is the volume rendering model for GPU1. As render times approach zero, our model consistently over-estimates the cost in a small part of the distribution. The majority of the data points have more active pixels when sampling the volume saturates GPU memory throughput, and the model coefficients reflect the majority of the data points. With a small number of active pixels, GPU threads do not have to wait long for memory requests to be satisfied. The effects of the memory hierarchy on both the CPU and GPU are complex and occur at too fine-grain a level to be useful to our models.

Memory Usage

In addition to timing rendering, we captured memory usage information during our experiments. Our data set dimensions and image sizes for individual tests were essentially random, so we could not directly compare memory usage statistics. On the whole, rasterization used the least amount of memory, and ray tracing used the most since it uses an acceleration structure. On average, rendering increased the simulation memory overhead by less than 1%, and peaked at 5% when image size was large and data set size was small.

5.6 Multi-Node Performance Model

In a distributed memory setting, there are two factors that contribute to the overall rendering time: single node rendering time and image compositing time. The first factor is the maximum run-time out of the set of all MPI tasks, and we can predict the bottleneck task using the performance models from section 5.5. The second factor is the performance of compositing in which the images produced by each task are merged into a final image based on depth (e.g., rasterization and ray-tracing) or visibility ordering (volumes). In equation form, our total rendering model is:

$$T_{total} = max_{tasks}(T_{LR}) + T_{COMP} \tag{5.4}$$

where T_{LR} is the time for local rendering, i.e. T_{RT}, T_{VR}, or T_{RAST} based on the rendering technique used. Further, although it is not the primary focus of this paper, we need a basic model for image compositing (T_{COMP}) to answer questions about distributed rendering performance. Establishing T_{COMP} is the focus of the remainder of this section.

Observations

Figure 12 shows a histogram of compositing performance in our study as a function of MPI Rank and Pixels. The dominant trend in the histogram is that more pixels lead to slower times. The secondary trend is counterintuitive, which is that more tasks make compositing faster. This is because increasing MPI tasks causes the simulation domain to be divided up among more tasks, reducing the number of active pixels. This trend is likely to reverse as MPI tasks increase further, since the active pixels will decrease less quickly, but the amount of communication will increase. Unfortunately, our experimental study was limited in scope with respect to MPI tasks, and we were not able to capture these effects from our corpus of 1,350 experiments. A more thorough performance analysis at massive scales can be found in (98) and (110).

Model Definition

Our performance model for compositing is:

$$T_{COMP} = c_0 * ave_{tasks}(AP) + c_1 * Pixels + c_2 \tag{5.5}$$

where Pixels is the (full) image resolution.

Image compositing proceeds in a hierarchy of rounds exchanging pixel information, and the main factors are the number of pixels in the image and the number

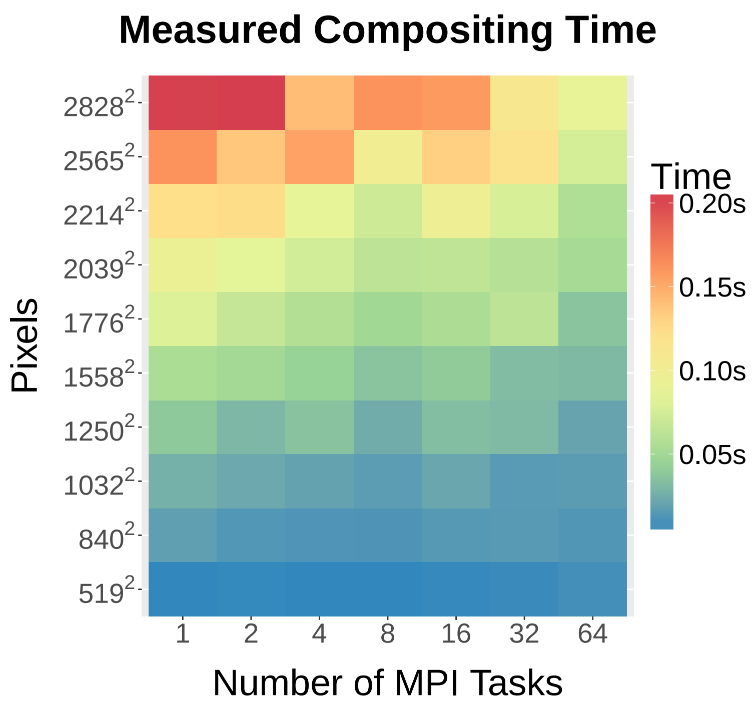

FIGURE 12. Histogram of image compositing time.

of MPI tasks. Our model uses image resolution for the number of pixels, and average number of active pixels captures the amount of work performed, since the active pixels is related to the number of cores and pixels that can contribute to the final image. As noted in Section 5.6, large numbers of MPI tasks will slow down compositing. We did not include that term in our model, since our regression analysis latched onto the decrease in execution time as MPI tasks went up (because of the decrease in active pixels) — adding that input variable using our corpus led to a model where increasing MPI tasks led to faster and faster compositing times. Again, while we believe we have the correct corpus for our single-node performance model, we understand our corpus is too limited for performance modeling of compositing.

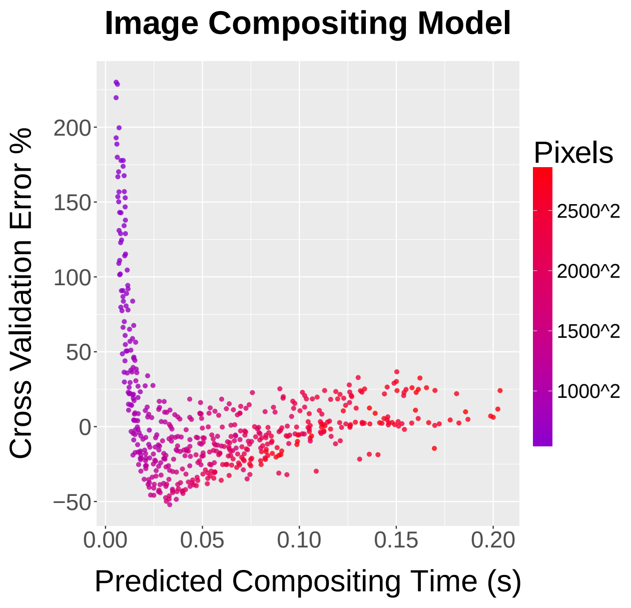

FIGURE 13. 3-fold cross-validation error plots for compositing. For each fold, the models are trained with a partial set of the data, then the models predict the total render time with the remaining data points. Error percentage is calculated as $100 \times (time_{total} - time_{predicted})/time_{total}$. A value of zero represents a perfect prediction. Each test is colored by the number of pixels composited.

Model Evaluation

The performance of our model is summarized in Figure 13, which shows the results of the cross-validation analysis, and table 14, which shows the percentage of predictions that fall within percentiles. The performance model underestimates the cost of compositing images of low resolution, but the model is fairly accurate for higher resolutions to provide a reasonable estimate (within the range of concurrencies we consider).

TABLE 14. Model accuracy summary from the result of the 3-fold cross validation analysis for our compositing model. For each fold, two thirds of the data is used to train the model and the remaining one third is used to test the prediction. Each percentage represents the number of predicted values, from all folds, that are within the corresponding error percentage.

	50%	25%	10%	5%	Average %
Compositing	88.4	68.3	28.3	15.0	29.3

5.7 Evaluation on Leading-Edge Supercomputer

To further validate our models, we tested on a leading edge supercomputer, ORNL's Titan machine. Each node contains a 16-core AMD processor with 32 GB of memory, and we used the following configuration:

– GPU2: 1 MPI task per node

  * Simulation Code: 16 OpenMP threads
  * Strawman: 1 K20 GPU per task

The architecture-specific coefficients (c_i) changed on this machine, so we ran a small number of experiments to calibrate.

We used Cloverleaf3D as the simulation code, and ran between 20 and 31 experiments for each renderer, using the same methods as described in Section 5.4. Next, we performed a large-scale run, using 1024 nodes of GPU2 for each renderer with an image resolution of 2048^2 and over 16 billion total elements, and tested the models predictions against the measured run-time.

Table 15 shows the results of the runs including actual render time, predicted render time, percentage error, and number of samples used to train the model for each renderer. For ray-tracing and rasterization, our render time predictions were within 6% and 18% of the actual runtime, which was in line with our expectations. The prediction

TABLE 15. We trained our models using a minimal number of data points using Cloverleaf3D and between 1 and 32 MPI tasks on Titan. We used the same sampling strategy as the first phase of our study. We then increased the number of MPI tasks to 1024 with 16 billion total cells at an image resolution of 2048^2, and tested the run-time against the model predictions.

Rendering Technique	Actual Time	Predicted Time	Difference	Sample Points
Ray-Tracing	0.0165s	0.0176s	-6.0%	31
Volume Render	0.00638s	0.0136s	-78.6%	30
Rasterization	0.00275s	0.00224s	18.5%	20

for volume rendering was 78% away from the actual run-time, and this matched the error we saw in the upper-right of figure 11 as the renderer time approached zero.

As discussed in Section 5.6, our compositing model is not appropriate at the scale of 1024 MPI tasks, so we do not present it here.

5.8 Mapping Rendering Configurations to Model Inputs

Visualization experts and domain scientists likely think of rendering configurations in terms of our study options from Section 5.4 (i.e., architecture, rendering technique, number of MPI tasks, image resolution, and data set), and it is doubtful that they would immediately think of their rendering problem in terms of our variable inputs (i.e., O, AP, VO, PPT, SPR, CS, and c_i). To bridge this gap, a mapping from rendering configurations to variable inputs is needed. In general, an exact mapping can be difficult to generate, especially because some variable inputs can be data-specific. For example, the number of triangles in an isosurface is not known until you apply an isosurfacing algorithm. That said, it is typically easy to provide upper bounds for the terms, which in turn can be fed into our model, even for arbitrary rendering configurations. Further, since all coefficients c_i are positive, as the upper bound becomes larger and larger, then our prediction will get larger and larger. This is a desirable property, since it means that overestimates lead to conservative results.

For our study, we use the following mappings:

- Objects (O): varies based on rendering type
  * For rasterization and ray-tracing and a data set of size N^3, this is $12 \times N^2$, since we are taking external faces (each of the six exterior faces of the domain has N^2 quadrilaterals, which is two triangles).
  * For volume rendering, this is N^3.
- Active Pixels (AP): $55\% \times \frac{1}{\#MPITasks^{1/3}} \times Pixels$

  Our camera positions filled about 60% of pixels by default. Each time the number of blocks went up by N^3, the number of active pixels for a rank dropped to be approximately $\frac{1}{N^{th}}$ as much. So, with 8 blocks, each MPI task would have about 30% of all available pixels as active.
- Visible Objects (VO): min(AP, O)

  If there are fewer triangles than active pixels, then all triangles are visible. Otherwise, the number of visible objects is approximately the same as the number of active pixels.
- Pixels Per Triangle (PPT): $AP \times 4$

  For our external faces, active pixels on average have two overlapping triangles (front face and back face). Further, an additional two triangles (the triangles that "complete the quadrilateral" with the original two triangles) will still consider these pixels, but will fail their inside-out tests.
- Samples Per Ray (SPR): $373 \times \frac{1}{\#MPITasks^{1/3}}$

  Like Active Pixels, we found the baseline value for a single core, and then considered behavior as the number of MPI tasks increased.

– Cells Spanned (CS): N (from an N^3 mesh)

  The maximum value is the number of cells that span the diagonal of the data set and the minimum is zero. We found N to be a good estimate.

Again, the mappings above were appropriate for our study (and we validated them by checking against experimental results), but would have to be re-thought for camera configurations other than those inspired by the Cinema specification. The purpose for showing them was (1) to provide background for our next section exploring rendering feasibility questions and (2) to demonstrate an example mapping for those considering other rendering configurations.

Validation of Mapping

To validate our mapping procedure, we considered six rendering configurations. For each configuration, we applied our mapping to obtain variable inputs, and then used our performance model. For those same six rendering configurations, we also retrieved the observed variable inputs from the experiments and used those variable inputs to predict execution time with the performance model. Finally, we also compared with the observed execution time for each rendering configuration. The results of this validation are in Table 16. It shows that our mapping does a reasonable job of predicting variable inputs (for our randomly selected test cases), and also that our conservative estimates do lead to slower prediction times in some cases.

5.9 In Situ Viability

In this section, we use our performance models to ask rendering feasibility questions. We utilize the experimentally-determined coefficients from Sections 5.5 and 5.6 in our model, and list those coefficients in Table 17. We also make use of the

TABLE 16. This table contains five groupings of information that collectively provide evidence that our mapping from rendering configuration to variable inputs is valid. The first grouping describes six randomly chosen test configurations from the study, one for each of our performance models. The middle three groupings show the values for variable inputs in our performance models for the three renderers. The final grouping shows the execution time from a prediction using our mapping (labeled Mapping), from a prediction using our observed variable inputs from an experiment (labeled Experiment), and the actual execution time from the tests (labeled Actual).

Test #	Arch.	Rendering Technique	Mesh Size	Image Size	MPI Tasks
0	CPU	Volume Rendering	206^3	2375^2	4
1	CPU	Ray-Tracing	224^3	1598^2	32
2	CPU	Rasterization	185^3	1712^2	8
3	GPU	Volume Rendering	226^3	2322^2	2
4	GPU	Ray-Tracing	155^3	1688^2	64
5	GPU	Rasterization	177^3	2838^2	16

Test (Description)	Type	Active Pixels	Samples Per Ray	Cells Spanned
0 (CPU / Vol.Ren.)	Predicted	1.96M	235	206
0 (CPU / Vol.Ren.)	Observed	1.84M	217	206
3 (GPU / Vol.Ren.)	Predicted	2.35M	296	262
3 (GPU / Vol.Ren.)	Observed	2.24M	281	262

Test (Description)	Type	Objects	Active Pixels
1 (CPU / Ray-Tracing)	Predicted	442K	339K
1 (CPU / Ray-Tracing)	Observed	602K	338K
4 (GPU / Ray-Tracing)	Predicted	228K	232K
4 (GPU / Ray-Tracing)	Observed	288K	232K

Test (Description)	Type	Objects	Visible Objects	Pixels Per Triangle
2 (CPU / Rasterization)	Predicted	406K	406K	7.94
2 (CPU / Rasterization)	Observed	410K	408K	7.1
5 (GPU / Rasterization)	Predicted	371K	371K	18.9
5 (GPU / Rasterization)	Observed	376K	375K	18.0

Test #	Mapping	Experiment	Actual
0	2.47s	2.075s	2.02s
1	0.14s	0.10s	0.08s
2	0.009s	0.006s	0.006s
3	0.97s	0.88s	0.89s
4	0.07s	0.05s	0.05s
5	0.14s	0.13s	0.13s

TABLE 17. Experimentally-determined coefficients for our models.

Technique	Arch	c_0	c_1	c_2	c_3	c_4
Ray-tracing	CPU1	5.39e-8	1.13e-2	1.84e-9	3.46e-8	1.28e-2
Ray-tracing	GPU1	1.32e-8	3.81e-3	3.63e-10	2.13e-9	3.54e-3
Rasterization	CPU1	1.29e-8	1.97e-9	1.75e-2	–	–
Rasterization	GPU1	2.10e-9	3.73e-10	1.36e-3	–	–
Volume	CPU1	3.68e-10	4.46e-9	9.35e-2	–	–
Volume	GPU1	1.43e-10	1.07e-9	9.35e-3	–	–
Compositing	–	1.89-e8	4.73e-5	-3.25e-2	–	–

mapping from Section 5.8 so that we can pose our questions in terms of rendering configurations (as opposed to our performance model's input variables).

We focus on two specific rendering feasibility questions here, to establish the benefit of our performance model and also to provide a demonstration of its use. However, we note that our model can be used to answer many more types of questions.

Maximizing Images Rendered in a Fixed Time Allotment

As mentioned in the introduction, a new image-based paradigm is emerging for *in situ* processing (64; 19; 20). The researchers behind these works are advocating for extracting thousands to millions of images, the rendering of which could significantly impact the overall execution time of a simulation code in a tightly-coupled setting. While we could explore many issues in this context, we consider the following: what is the tradeoff between image size and the total number of images produced? While we cannot know whether a domain scientist prefer to produce many small images or few large ones, we can enable that scientist to make an informed decision.

We approached our example problem as follows: We assumed a fixed configuration for data and number of MPI tasks, since these factors would typically be fixed when running tightly-coupled *in situ*. For our example, we assumed 32 MPI tasks, each with a data set that is 200^3. We also assumed a budget of sixty seconds from the simulation

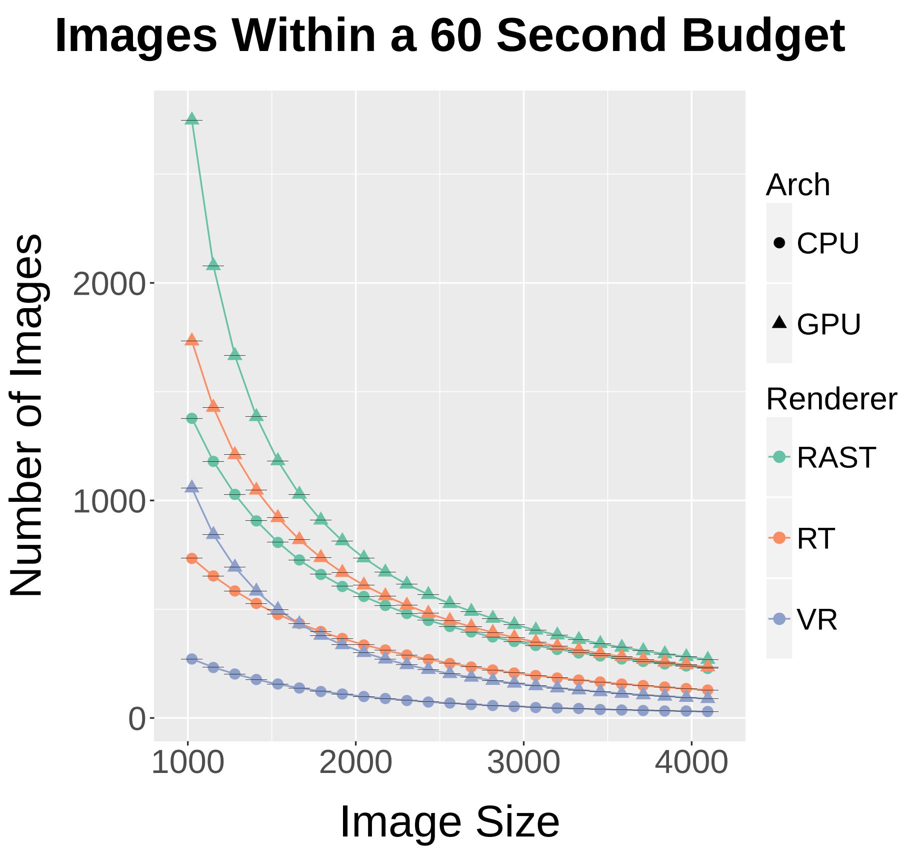

FIGURE 14. Predictions from our performance model on how many images each rendering technique can generate in 60 seconds, as a function of image size. This analysis could allow a domain scientist to make informed tradeoffs between quality (high image resolutions) and quantity (number of viewpoints).

code to perform our visualizations. We then considered all six performance models (two architectures times three rendering techniques) and ran our performance model with images sizes varying from 1024^2 to 4096^2, in steps of 128 in both dimensions. For each image size, we got a predicted rendering time, which we could then use to calculate the number of frames per minute. For example, if a single rendering takes 0.2s, then we could do 300 renderings in one minute. The results are listed in Figure 14.

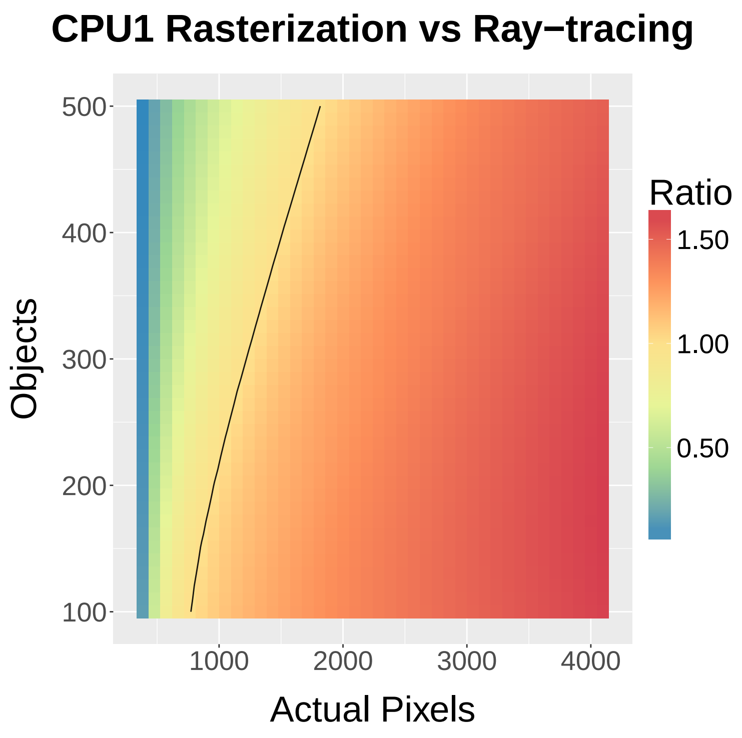

FIGURE 15. Heatmap comparing ray-tracing and rasterization performance. A value of 1.5 means that rasterization is faster, and can produced three images for every two ray-tracings. A value of 0.5 means two ray-tracings can be rendered for every rasterization. While rasterization is faster in a larger proportion of the configurations, ray-tracing has much more significant advantages in the configurations where it is faster.

Which is Better?: Ray-Tracing and Rasterization

While rasterization has traditionally been the dominant paradigm in scientific visualization for rendering surface data, researchers have recently been considering if ray-tracing may be superior (34). The intuition for ray-tracing is that it can perform well with large data sets, since its work is proportional to the number of pixels. On the downside, ray-tracing works best when accelerated by efficient spatial search structures (like the LBVH), and these structures take time to build. However, in repeated rendering use cases (like the ones described in the above Sub-Section, the build time is amortized, since it only needs to be done before the first render.

With our performance model, we can consider the ray-tracing versus rasterization question. We supposed an example setting where there were 32 MPI tasks and we wanted to perform 100 renderings. We then varied the image size from 384^2 to 4096^2 in steps of 128 in both dimensions and considered data sizes from 100^3 to 500^3 in steps of 25^3 in all three dimensions. For each of the results 510 configurations, we ran our performance model for both rendering types and considered the ratio between their predicted execution times. Figure 15 plots the results. As suspected, ray-tracing does have a significant advantage when there is a lot of geometry and few pixels. In the most extreme case — 384^2 images and 500^3 data — ray-tracing could produce 16 images for every one rasterization. However, rasterization gained regular advantages when images went above 1024^2, and was a consistent winner by 1920^2. That said, its biggest advantages, when image size was large and geometry was small, was nowhere near as lopsided as ray-tracing's: the top advantage for rasterization over all configurations we considered was to render three images for every two ray-tracings.

5.10 Summary

We established performance models for three rendering methods in order to explore feasibility questions with *in situ* rendering. These feasibility questions will be increasingly important as *in situ* processing becomes more and more common, leading simulation codes and visualization programs to compete for resources. Our models were validated through statistical analyses and shown to have good fit overall. This in turn allowed us to explore a sampling of interesting feasibility questions, which in turn demonstrates the utility of our models. In terms of future work, we would like to explore additional rendering considerations, including the impact of camera angle, varying geometry per MPI task, and compositing at higher scales.

CHAPTER VI

FUTURE DIRECTIONS

In the previous chapter, we demonstrated the effectiveness of our methodology for creating performance models and showed their usefulness by using the models to answer important in situ rendering viability questions. In this chapter, we discuss the future directions that can build on this research.

6.1 Modeling Other Algorithms

Within visualization, rendering is a small but important subset of all visualization algorithms. Our models are useful we considering image-based in situ, but there are use cases where image-based in situ may not be the most appropriate option for analysis of a simulation. Consequently, we need to apply performance modeling to other visualization algorithms to help manage the complex constraints between simulation and visualization in the broader context of all the possible in situ use cases. This is a challenging task because there are hundreds of visualization algorithms. Some visualization algorithms will be low hanging fruit, but there are others that will be more difficult since they may have inputs that are hard to estimate.

Streamlines are an example of an important algorithm that will pose a challenge to model. The process of creating streamlines involves placing particles throughout a vector field, and then advecting the particles to create a path. Depending on the characteristics of the vector field, a particle might terminate immediately or might spin around in a vortex until the maximum amount of advection steps is completed. In a distributed memory setting, the vector field could force particles toward a single domain, creating a large work imbalance. Consequently, estimating the amount of work in a domain at any given time is problematic, and this will make creating a performance model challenging without an analysis of the vector field.

Other algorithms should be straight-forward. For example, slicing extracts a 2-D plane from a 3-D data set, and creating a slicing performance model is likely as simple as estimating the amount of cells intersected by the plane. Further, while modeling the performance of streamlines is challenging and slicing should be easy, other visualization algorithms will fall within the spectrum defined by the two extremes.

6.2 Data Gathering Infrastructure

In order to facilitate performance model creation for other visualization algorithms, we would greatly benefit from a generic data gathering infrastructure. Having such an infrastructure would reduce the effort needed to instrument code and to capture the timing data that is required to generate the models. There are hundreds of visualization algorithms and lowering the barrier-to-entry for model creation will encourage others to apply our methodology. Our methodology starts the simplest possible model and then decomposes each algorithmic input until it accurately predicts the run-time of a particular algorithms. If we create hierarchical annotations for timings gathered within an algorithm, we could automate model creation.

The models in this dissertation were developed off-line. We ran tests to gather data, created the models, and then used the models to answer questions. With automated model creation, we could process data as it is generated and develop models in an on-line fashion. Models would be refined as more data is generated, with model accuracy increasing as the corpus grows.

Another approach would be to use a database of information to create models tightly fitted to the run-time parameters of a simulation. Using a previously created performance model, we could query this database on-the-fly for similar configurations, and we could create a tightly fitting model that would more accurately predict the run-time of the algorithms. At first, the database would be sparse, seeded with a small amount test data. As simulations run, visualization algorithms would deposit small amounts of information every time they run, thus populating the database and increasing model accuracy. Further, we would gain valuable information about what algorithms are used most frequently, which would allow the visualization community to focus their effort where it is needed most.

6.3 Adaptive Infrastructure

Once the visualization community has created models for algorithms, we need to pass model estimates upstream directly to the simulation or to an adaptive in situ infrastructure layer that manages run-time constraints. Such an infrastructure would be responsive to the needs of the simulation and coordinate visualization according to the current simulation state. In this dissertation, we considered the amount of time it takes to render, but run-time is just one of the constraints. The adaptive layer would receive process constraints with memory consumption, power usage, and run-time.

Adaptive infrastructures would require inputs from both the simulation and visualization, and would require a descriptive API to communicate simulation needs and constraints. Above the layer, simulations would register constraints about how much memory is currently available, how much time it is willing to dedicate to visualization, the maximum amount of power it can consume, and what types of data features are interesting to domain scientists. Below the layer, the visualization algorithms need to be able to provide estimates, such as the ones described in this dissertation, on all constraints based on the current state of the simulation such as mesh size and output type.

Finally, the adaptive layer would choose visualization algorithms based on the input from the simulation. While there is much research and development needed as we approach exascale, this dissertation lays down some of the foundation so we can begin to tackle the complex issues that arise from an environment of constraints.

REFERENCES CITED

[1] Apache thrift. https://thrift.apache.org/. [Accessed 12-Aug-2015].

[2] Bson specification. http://bsonspec.org/. [Accessed 12-Aug-2015].

[3] Civetweb project. https://github.com/civetweb/civetweb. [Accessed 16-Aug-2015].

[4] CloverLeaf3D. http://uk-mac.github.io/CloverLeaf3D/. [Accessed 14-Aug-2015].

[5] Conduit documentation. http://scalability-llnl.github.io/conduit/. [Accessed 14-Aug-2015].

[6] A high performance, highly scalable software rasterizer for opengl. http://http://openswr.org/. Accessed: 2016-4-01.

[7] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[8] The industry's foundation for high performance graphics. http://www.opengl.org/. Accessed: 2016-4-01.

[9] Json specification. http://json.org/. [Accessed 12-Aug-2015].

[10] Kripke. https://codesign.llnl.gov/kripke.php. [Accessed 14-Aug-2015].

[11] The mesa 3d graphics library. http://www.mesa3d.org/. Accessed: 2016-4-01.

[12] MFEM: Modular finite element methods. http://www.mfem.org. [Accessed 16-Aug-2015].

[13] Protocol buffers. https://developers.google.com/protocol-buffers/. [Accessed 12-Aug-2015].

[14] Silo. https://silo.llnl.gov. [Accessed 16-Aug-2015].

[15] XDMF: extensible data model and format. `http://www.xdmf.org`. [Accessed 16-Aug-2015].

[16] Performance application programming interface, 2012.

[17] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, Jim Ahrens, E. Wes Bethel, Hank Childs, Jian Huang, Kenneth I. Joy, Quincey Koziol, Jay Lofstead, Jeremy Meredith, Ken Moreland, George Ostrouchov, Mike Papka, Venkat Vishwanath, Matthew Wolf, Nick Wright, and K. John Wu. Scientific Discovery at the Exascale: Report for the DOE ASCR Workshop on Exascale Data Management, Analysis, and Visualization, July 2011.

[18] James Ahrens, Berk Geveci, Charles Law, CD Hansen, and CR Johnson. 36-paraview: An end-user tool for large-data visualization, 2005.

[19] James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H Rogers, and Mark Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434. IEEE Press, 2014.

[20] James Ahrens, John Patchett, Andrew Bauer, Sébastien Jourdain, David H Rogers, Mark Petersen, Benjamin Boeckel, Patrick OLeary, Patricia Fasel, and Francesca Samsel. In situ mpas-ocean image-based visualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Visualization & Data Analytics Showcase*, 2014.

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

[22] Timo Aila, Samuli Laine, and Tero 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.

[23] Hiroshi Akima and Albrecht Gebhardt. *akima: Interpolation of Irregularly and Regularly Spaced Data*, 2015. R package version 0.5-12.

[24] Utkarsh Ayachit, Berk Geveci, Kenneth Moreland, John Patchett, and Jim Ahrens. The ParaView Visualization Application. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 383–400. October 2012.

[25] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition*, 2:359–371, 2011.

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

[27] James Bigler, Abe Stephens, and Steven Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of Interactive Ray Tracing*, pages 187–196, 2006.

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

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

[30] Solomon Boulos, Ingo Wald, and Carsten Benthin. Adaptive Ray Packet Reordering. In *Proceedings of Interactive Ray Tracing*, pages 131–138, 2008.

[31] Ian Bowman, John Shalf, Kwan-Liu Ma, and Wes Bethel. Performance modeling for 3d visualization in a heterogeneous computing environment. *Lawrence Berkeley National Laboratory*, 2004.

[32] C. Brownlee, T. Fogal, and C.D. 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.

[33] C. Brownlee, J. Patchett, L.T. Lo, D. DeMarle, C. Mitchell, J. Ahrens, and C.D. 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.

[34] Carson Brownlee, John Patchett, Li-Ta Lo, David DeMarle, Christopher Mitchell, James Ahrens, and Charles D. 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*. The Eurographics Association, 2012.

[35] Paul Bunyk, Arie Kaufman, and Cláudio T Silva. Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization Conference, 1997*, pages 30–36. IEEE, 1997.

[36] Steven P Callahan, Milan Ikits, João Luiz Dihl Comba, and Claudio T Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):285–295, 2005.

[37] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[38] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 13. ACM, 2011.

[39] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 357–372. October 2012.

[40] Hank Childs, Mark Duchaineau, and Kwan-Liu Ma. A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 153–162, Braga, Portugal, May 2006.

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

[42] Hank Childs, Berk Geveci, Will Schroeder, Jeremy Meredith, Kenneth Moreland, Christopher Sewell, Torsten Kuhlen, and E. Wes Bethel. Research Challenges for Visualization Software. *IEEE Computer*, 46(5):34–42, May 2013.

[43] Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther Weber, and E. Wes Bethel. Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 30(3):22–31, June 2010.

[44] Jonathan D Cohen, Daniel G Aliaga, and Weiqiang Zhang. Hybrid simplification: combining multi-resolution polygon and point rendering. In *Proceedings of the conference on Visualization'01*, pages 37–44. IEEE Computer Society, 2001.

[45] Thomas W Crockett. An introduction to parallel rendering. *Parallel Computing*, 23(7):819–843, 1997.

[46] David Culler, Richard Karp, et al. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, 1993.

[47] Zachary 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.

[48] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 155–163. IEEE, 2012.

[49] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[50] H Carter Edwards and Daniel Sunderland. Kokkos array performance-portable manycore programming model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2012.

[51] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E Jan. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

[52] Ricardo Farias, Joseph SB Mitchell, and Cláudio T Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 91–99. ACM, 2000.

[53] Paul F. Fischer, James W. Lottes, and Stefan G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[54] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *EUROGRAPHICS*, 2010.

[55] William Gropp, Ewing Lusk, and Anthony 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.

[56] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[57] Charles Hansen, E. Wes Bethel, Thiago Ize, and Carson Brownlee. Rendering. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 49–70. October 2012.

[58] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 6:1–6:12, New York, NY, USA, 2011. ACM.

[59] R Hornung, J Keasler, et al. The raja portability layer: overview and status. *Lawrence Livermore National Laboratory, Livermore, USA*, 2014.

[60] Mark Howison, E. Wes Bethel, and Hank Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 1–10, Norrköping, Sweden, April 2010. One of two ***best papers***.

[61] Mark Howison, E Wes Bethel, and Hank Childs. Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1):17–29, 2012.

[62] Mark Howison, E. Wes Bethel, and Hank 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, January 2012.

[63] Curtis L. Janssen, Helgi Adalsteinsson, et al. Using simulation to design extremescale applications and architectures: Programming model exploration. *SIGMETRICS Performance Evaluation Review*, 38(4):4–8, 2011.

[64] Akira Kageyama and Tomoki Yamada. An approach to exascale visualization: Interactive viewing of in-situ visualization. *Computer Physics Communications*, 185(1):79–85, 2014.

[65] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Aug 2013.

[66] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

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

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

[69] Aaron Knoll, Ingo Wald, Paul Navratil, Anne Bowen, Khairi Reda, Michael E Papka, and Kelly Gaither. Rbf volume ray casting on multicore and manycore cpus. In *Computer Graphics Forum*, volume 33, pages 71–80. Wiley Online Library, 2014.

[70] Aaron Knoll, Ingo Wald, Paul A Navrátil, Michael E Papka, and Kelly 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.

[71] Adam Kunen. Kripke: User manual v1.0. Technical report, Aug 2014.

[72] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 79–88. ACM, 2011.

[73] Matt Larsen, Jeremy Meredith, Paul Navrátil, and Hank Childs. Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 279–286, Hangzhou, China, April 2015.

[74] Matthew Larsen, Eric Brugger, Hank Childs, Jim Eliot, Kevin Griffin, and Cyrus Harrison. Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), held in conjunction with SC15*, pages 30–35, Austin, TX, November 2015.

[75] Matthew Larsen, Cyrus Harrison, James Kress, Dave Pugmire, Jeremy Meredith, and Hank Childs. Performance modeling of in situ rendering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016 (To Appear).

[76] Matthew Larsen, Stephanie Labasan, Paul Navrátil, Jeremy Meredith, and Hank Childs. Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 53–62, Cagliari, Italy, May 2015.

[77] Matthew Larsen, Stephanie Labasan, Paul Navrátil, Jeremy Meredith, and Hank Childs. Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 53–62, Cagliari, Italy, May 2015.

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

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

[80] Li-ta Lo, Christopher Sewell, and James Ahrens. PISTON: A portable cross-platform framework for data-parallel visualization operators. pages 11–20. Eurographics Symposium on Parallel Graphics and Visualization, 2012.

[81] Li-ta Lo, Christopher Sewell, and James P Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV*, pages 11–20, 2012.

[82] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[83] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[84] Benjamin Lorendeau, Yvan Fournier, and Alejandro Ribes. In-situ visualization in fluid mechanics using catalyst: A case study for code saturne. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pages 53–57. IEEE, 2013.

[85] Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE*, 29(6):14–19, 2009.

[86] Kwan-Liu Ma, James S Painter, Charles D Hansen, and Michael F Krogh. Parallel volume rendering using binary-swap compositing. *Computer Graphics and Applications, IEEE*, 14(4):59–68, 1994.

[87] John H. Maindonald and W. John Braun. *DAAG: Data Analysis and Graphics Data and Functions*, 2015. R package version 1.22.

[88] AC Mallinson, David A Beckingsale, WP Gaudin, JA Herdman, JM Levesque, and Stephen A Jarvis. Cloverleaf: Preparing hydrodynamics codes for exascale. 2013.

[89] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[90] 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.

[91] 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.

[92] Jeremy S Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. Eavl: the extreme-scale analysis and visualization library. 2012.

[93] Jeremy S. Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. EAVL: The extreme-scale analysis and visualization library. In Hank Childs, Torsten Kuhlen, and Fabio Marton, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012.

[94] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Parallel and Large-Data Visualization and Graphics, 2001. Proceedings. IEEE 2001 Symposium on*, pages 85–154, Oct 2001.

[95] Kenneth Moreland. Icet users guide and reference. Technical report, version 2.0. Technical Report SAND2010-7451, Sandia National Laboratories, 2011.

[96] Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 97–104. IEEE, 2011.

[97] Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu 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.

[98] Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2011.

[99] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, et al. Examples of in transit visualization. In *Proceedings of the 2nd international workshop on Petascale data analytics: challenges and opportunities*, pages 1–6. ACM, 2011.

[100] Kenneth Moreland, Christopher Sewell, William Usher, Li-ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, et al. Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications*, 36(3):48–58, 2016.

[101] Kenneth Moreland, Christopher Sewell, William Usher, Li ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, May/June 2016. (to appear).

[102] Paul A. Navrátil, Hank Childs, Donald S. Fussell, and Calvin Lin. Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 20(6):893–906, June 2014.

[103] Ulrich Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Parallel Rendering Symposium, 1993*, pages 97–104. IEEE, 1993.

[104] Lesley Northam and Rob Smits. Hort: Hadoop online ray tracing with mapreduce. In *ACM SIGGRAPH 2011 Posters*, page 22. ACM, 2011.

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

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

[107] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk. Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints*, March 2004.

[108] Steven G Parker et al. Optix: a general purpose ray tracing engine. In *ACM Transactions on Graphics (TOG)*, volume 29, page 66. ACM, 2010.

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

[110] Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 4. ACM, 2009.

[111] Tom Peterka, Hongfeng Yu, Robert B Ross, Kwan-Liu Ma, et al. Parallel volume rendering on the ibm blue gene/p. In *EGPGV*, pages 73–80. Citeseer, 2008.

[112] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics (Proceedings of SIGGRAPH)*, 31(Annual Conference Series):101–108, August 1997.

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

[114] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

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

[116] Silvio Rizzi, Mark Hereld, Joseph Insley, Michael E. Papka, Thomas Uram, and Venkatram Vishwanath. Performance Modeling of vl3 Volume Rendering on GPU-Based Clusters. In Margarita Amor and Markus Hadwiger, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2014.

[117] A. F. Rodrigues, K. S. Hemmert, et al. The structural simulation toolkit. *SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[118] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. ISBN 978-0-387-75968-5.

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

[120] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. 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*, pages 93–ff. IEEE Computer Society Press, 1996.

[121] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. 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*, pages 93–100. IEEE Computer Society Press, 1996.

[122] Hendrik A Schroots and Kwan-Liu Ma. Volume rendering with data parallel visualization frameworks for emerging high performance computing architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, page 3. ACM, 2015.

[123] Peter Shirley and Allan Tuchman. *A polygonal approximation to direct scalar volume rendering*, volume 24. ACM, 1990.

[124] Cláudio Teixeira Silva, Joao Luiz Dihl Comba, Steven Paul Callahan, and Fabio Fedrizzi Bernardon. A survey of gpu-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA)*, 12(2):9–29, 2005.

[125] Kyle Spafford and Jeffrey Vetter. Aspen: a domain specific language for performance modeling. In *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2012.

[126] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13. ACM, 2009.

[127] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[128] Jeff A Stuart, Cheng-Kai Chen, Kwan-Liu Ma, and John D Owens. Multi-gpu volume rendering using mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 841–848. ACM, 2010.

[129] Nicolaas Tack, Francisco Morán, Gauthier Lafruit, and Rudy Lauwereins. 3d graphics rendering time modeling and control for mobile terminals. In *Proceedings of the ninth international conference on 3D Web technology*, pages 109–117. ACM, 2004.

[130] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[131] Ingo Wald, Carsten Benthin, and Solomon 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.

[132] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):153–164, 2001.

[133] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (proceedings of SIGGRAPH, to appear)*, 2014.

[134] Stfan van der Walt, S. Chris Colbert, and Gal Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[135] Taiyun Wei. *corrplot: Visualization of a correlation matrix*, 2013. R package version 0.73.

[136] Brad Whitlock, Jean M Favre, and Jeremy S Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109. Eurographics Association, 2011.

[137] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009.

[138] Sven Woop, Louis Feng, Ingo Wald, and Carsten Benthin. Embree ray tracing kernels for cpus and the xeon phi architecture. In *ACM SIGGRAPH 2013 Talks*, page 44. ACM, 2013.

[139] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 7–12. IEEE Press, 2002.