Multi-core and Many-core Shared-memory Parallel Raycasting Volume Rendering Optimization and Tuning

E. Wes Bethel*
Lawrence Berkeley National Laboratory, Berkeley CA

Mark Howison†
Brown University, Providence RI

January, 2012

*e-mail: ewbethel at lbl dot gov
†e-mail: mhowison at brown dot edu

Acknowledgment

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET).

Legal Disclaimer

Abstract

Given the computing industry trend of increasing processing capacity by adding more cores to a chip, the focus of this work is tuning the performance of a staple visualization algorithm, raycasting volume rendering, for shared-memory parallelism on multi-core CPUs and many-core GPUs. Our approach is to vary tunable algorithmic settings, along with known algorithmic optimizations and two different memory layouts, and measure performance in terms of absolute runtime and L2 memory cache misses. Our results indicate there is a wide variation in runtime performance on all platforms, as much as 254% for the tunable parameters we test on multi-core CPUs and 265% on many-core GPUs, and the optimal configurations vary across platforms, often in a non-obvious way. For example, our results indicate the optimal configurations on the GPU occur at a crossover point between those that maintain good cache utilization and those that saturate computational throughput. This result is likely to be extremely difficult to predict with an empirical performance model for this particular algorithm because it has an unstructured memory access pattern that varies locally for individual rays and globally for the selected viewpoint. Our results also show that optimal parameters on modern architectures are markedly different from those in previous studies run on older architectures. And, given the dramatic performance variation across platforms for both optimal algorithm settings and performance results, there is a clear benefit for production visualization and analysis codes to adopt a strategy for performance optimization through auto-tuning. These benefits will likely become more pronounced in the future as the number of cores per chip and the cost of moving data through the memory hierarchy both increase.

Keywords: parallel volume rendering, performance optimization, auto-tuning, multi-core CPU, many-core GPU.

1 Introduction

For the past two decades, a majority of high performance parallel computing research, including much visualization research, has focused on the large, COTS-based distributed memory system as the primary platform. These systems have become more powerful due to a combination of faster processors, faster memory, and faster interconnect. Up until recently, growth in processor speed and capacity more or less followed Moore's Law: transistors shrank, allowing more of them to be packed into a given unit area of silicon, and clock rates increased. As heat and power constraints may limit future gains, multi-core processors have emerged as industry's solution.

In computational science research communities, recent research has focused on the problem of adapting algorithms to make effective use of this "new" platform, the multi-core processor. Achieving optimal algorithm performance typically involves a careful examination of how an algorithm moves data through the memory hierarchy. Unfortunately, the cache hierarchies and memory management systems evolve at a very fast pace. Increasingly, there is often a non-obvious relationship between algorithmic parameters like blocking strategies, their impact on memory utilization, and in turn the relationship with runtime issues. For example, on GPUs, performance can be highly sensitive to the degree to which threads in a thread block all execute the same code: performance will degrade when there is conditional branching and threads execute different code paths. Recent work has shown there is not necessarily a clear correlation between the amount of memory traffic and absolute runtime [7], a result confirmed by our study here. At best, multi-processor algorithmic performance models, which are derived using a combination of architectural knowledge and empirical runtime observations, can predict performance bounds, but don't always offer advice on how algorithms should be tuned to produce optimal performance, particularly in light of architecture-specific characteristics, like thread divergence on GPUs.

As a result, the computational science community has evolved a technique known as "auto-tuning", which is methodology for finding the combinations of tunable algorithmic parameters that result in the best performance of an algorithm on a particular platform for a particular problem size. This approach has been used with success to optimize performance of stencil-based codes on multi-core CPUs and many-core GPUs [39, 17, 21, 7, 5, 6]. Auto-tuning is based upon the premise that one can enumerate all possible tunable configurations of an algorithm. As the number of potential permutations of configuration can be quite large, search strategies have emerged as a research area to avoid having to perform an exhaustive search of all possible configurations.

The contribution of this work is an application of auto-tuning principles – the systematic exploration of tunable algorithmic parameters, known algorithmic optimizations, and memory layout strategies – to study the relative performance gain of a staple visualization algorithm, raycasting volume rendering, on two multi-core CPUs and one many-core GPUs by measuring absolute runtime as well as memory cache utilization (L2 cache misses via hardware performance counters). The comparison of hardware performance counter and runtime data helps to provide deeper insight into the relationship between memory traffic, cache utilization, and absolute runtime for volume rendering using known algorithmic optimizations. Our test results reveal a wide variation in runtime performance across all platforms and datasets, as much as 254% on the CPU and 265% on the GPU. The settings that produce the best performance vary from problem to problem and platform to platform, often in a non-obvious way. For example, our results indicate the best-performing configurations on the GPU occur at a crossover between memory utilization and thread divergence, a result that is likely to be extremely difficult, if not impossible, to predict by with an empirical performance model.

The auto-tuning methodology, which is familiar to the computational science community, is relatively unexplored in the visualization community. The settings and algorithmic optimizations that result in "best performance" will change as platforms evolve: our results suggest much different settings than earlier work due to evolution in memory and processor technology. The focus of this work is a thorough study of the performance and memory utilization characteristics of known algorithmic optimizations on modern platforms. The findings reveal insights about their performance on modern platforms, including the fact that an optimization (Z-order memory layout) initially designed to be "cache friendly" on decade-old single-core technology is surprisingly beneficial on modern GPUs. The auto-tuning methodology will likely prove useful in suggesting new algorithmic optimizations and evolution on future platforms where core density increases along with the cost of moving data through the memory hierarchy.

2 Previous Work

Levoy's formulation for raycasting volume rendering is the process of casting rays through a 3D volume and integrating color and opacity along the ray's length to produce a final pixel color [24]. A common approach for parallelizing this staple visualization algorithm produces what is known as "hybrid volume rendering" [26, 37, 25, 1], which is a two-stage algorithm where data is partitioned across processors to be rendered with raycasting [24, 33, 11, 38]. Each processor generates a partial image, and these are combined in a second algorithmic stage using compositing [11, 24, 38].

Nieh and Levoy [29] describe a parallel ray tracing volume rendering algorithm that runs on a shared-memory platform and uses a task-queue image partitioning technique where screen pixels are partitioned amongst processors. Palmer et al. [31] studied two parallel partitioning and dynamic load balancing algorithms to explore the tradeoffs between their memory hierarchy performance. They suggest that image-order decomposition strategies suffer from a lack of locality in accessing the 3D volume data: during the ray integration loop – the most resource intensive part of the raycasting volume rendering algorithm – a given ray may need to access any voxel within the source volume. This lack of locality can result in "cache thrashing," which is a relatively low level of cache reuse. They report that object-parallel partitionings scale well, and this form of partitioning has been adopted as the basis for parallel work decomposition in many subsequent works (e.g., [4, 35, 40, 10, 20]). In contrast, our work here is a more comprehensive, systematic exploration of the relationship between algorithmic optimization and tunable algorithmic parameters – image tile size, work assignment strategy, and alternative memory layouts for the source data, and algorithmic optimizations – and their impact on algorithm performance in terms of runtime and cache utilization measured via hardware performance counters.

Looking more closely at the impact of memory accesses and algorithmic performance, Grimm et al. [16] explored the performance impact resulting when varying the size of data blocks allocated to processors in object-order parallel volume rendering on a shared-memory, multi-core CPU platform. Their objective is to minimize the number of repeated voxel loads, so their approach is to use an "advancing ray-front" approach [23] combined with small data blocks that will fit within cache so as to exploit spatial locality. In contrast, our approach is to exploit temporal, rather than spatial coherence, in an image-parallel approach that is portable to multi-core CPUs and many-core GPUs, as well as comparing performance of alternative memory layouts.

With the aim of exploring a memory layout that is more "cache friendly," Pascucci and Frank [32] use an indexing scheme based upon the Lebesgue, or Z-order, space filling curve to improve and optimize progressive visualization algorithms. This data layout has desirable spatial locality properties: at any point in the mesh, nearby points are nearby in memory or storage. In our study, we wish to compare the performance impact of this type of data layout in shared-memory parallel raycasting volume rendering with array-order memory layout with interactions of other algorithmic optimizations and tunable algorithmic parameters.

Whereas CPU-based volume rendering techniques span both object- and image-order parallel approaches, GPU-based volume rendering algorithms tend to exploit image-level parallelism. Examples including implementing volume rendering as a fragment program with raycasting [22, 34, 15, 13]. The basic idea with these implementations is to draw a proxy geometry where the fragment program, invoked during rasterization, performs the actual raycasting volume rendering. The degree of parallelism is a function of the number of fragment programs that can be run at once by a given GPU. With this approach, there is no direct opportunity for tuning algorithm performance in terms of parameters like block size, which may provide the opportunity for performance gains through improved temporal cache locality.

Marsalek et al. [27] implement a raycasting volume rendering kernel in CUDA. As with our CUDA implementation, theirs is an image-parallel approach where each CUDA thread performs raycasting on an image pixel. Their results suggest that CUDA kernels perform at a rate commensurate with earlier fragment-based approaches. They achieve a performance boost by a process of manual experimentation with thread block size – the number of threads per thread block – to find a good balance between register and shared memory use. In contrast, we systematically explore and report on the performance impact of varying the thread block size as well as alternative memory layouts.

In recent years, there has been a great deal of activity focusing on optimizing the performance of

codes on parallel computing platforms. Given the diversity of computer architectures, along with their rapid change, a technique known as "auto-tuning" has emerged as a way to tune codes to achieve optimal performance [6, 39, 17]. Auto-tuning is based upon the premise that one can enumerate all possible tunable configurations of an algorithm for a given problem size. As the number of potential permutations of configuration can be quite large, search strategies have emerged as a research area to avoid having to perform an exhaustive search of all possible configurations. Alternately, some recent research, such as that by de la Cruz and Araya-Polo [8], focuses on deriving a predictive performance model for an 3D stencil computation code, which has a uniform and predictable memory access pattern. The problem we are studying here, raycasting volume rendering, is representative of a class of algorithm that performs "unstructured" or "non-uniform" memory access. In particular, since our implementation uses perspective projection during rendering [14], each of the rays through the volume will diverge from one another along their length moving away from the viewpoint. The result is that each ray's memory access pattern is different than all other ray's memory access patterns. Therefore, this application is well suited for auto-tuning. While we perform what amounts to an exhaustive search of the space of tunable algorithmic parameters, interesting future work could focus on using one of several different search strategies to avoid the expensive exhaustive search.

Our work extends previous studies to explore the effect of varying the size of the image-tile partitioning and memory layout options in a shared-memory volume renderer on modern multi-core CPU and many-core GPU platforms. Specifically, we wish to understand the relationship between various tunable algorithmic parameters, namely tile size and shape, and memory layout alternative, with performance on a variety of platforms run on a variety of different problems. Do different size or shape tiles or different memory layouts result in better use of memory hierarchy? What is the difference in performance that results from different configurations? Is the performance and memory utilization sufficiently variable to warrant use of auto-tuning for large, production runs? Does a known algorithmic optimization, like early ray termination, offer a performance gain on a platform like the GPU where there is a performance penalty for thread divergence?

These questions help to better understand how tunable algorithmic parameters can affect performance on multi- and many-core systems. While the work in this study focuses on single-node rather than extreme concurrency configurations, our work here helps to reveal how to improve performance at extreme concurrencies by speeding the shared-memory parallel raycasting volume rendering by establishing algorithmic parameter settings that result in optimal, or near-optimal, performance. These single-socket configurations then comprise building blocks in large-scale multi-core hybrid-parallel and hybrid volume rendering applications executed at extreme concurrency, as in the case of recent work by Howison et al. [18], or in situations where multiple GPUs are employed as part of a distributed-memory parallel implementation [12].

3 System Design and Implementation

Our implementation is a raycasting volume renderer that follows Levoy's formulation [24] and that is parallelized using a shared-memory programming model. In our parallel implementation, each thread is responsible for casting rays into the volume, performing color and opacity integration along its ray, and writing the result into a final image buffer. We are using an image-space decomposition: there is one single copy of the volume data, the work is divided up amongst threads such that each thread is assigned a separate part of the image space. We use this algorithm design pattern and parallelization strategy on both multi-core CPUs (Section 3.1) and many-core GPUs (Section 3.2), along with alternative memory layouts (Section 3.3).

3.1 Multi-Core CPU Implementation

3.1.1 Shared Memory Volume Rendering

For the implementation in this study, our implementation uses POSIX threads [2] as the shared memory programming model and execution environment. Upon startup, the application launches a user-specified number of rendering threads. Each of these threads is a slave that executes an event loop. A master

thread tells all slave threads to execute a particular task, e.g., render the volume. A pair of shared memory barriers serve to synchronize the communication between the master and slave threads in the event loop. Prior to releasing the slave threads for rendering, the master thread will compute a number of values that are common across all slave rendering threads, e.g., inverse of model-view matrix, etc., and store these values into a shared-memory data structure that is visible to all threads.

3.1.2 Shared Memory Parallel Work/Block Decomposition

The approach we use for parallelism in this study is to have each thread independently cast rays into the volume to produce final image pixel; the master thread computes a list of work assignments for each thread, then each thread independently processes the list of work assignments. Each work assignment, or work block, consists of a spatially disjoint region of the final image, and a single thread will perform raycasting over all pixels in its assigned work block, then move on to the next work block. We explore two methods for assigning work blocks to threads: (1) a static assignment, where the master assigns blocks to threads in a round-robin fashion, and (2) a dynamic assignment where threads request work blocks from a single work queue. The user specifies an arbitrary tessellation of the output image that forms the basis for the work assignments: blocks of size 4×4, 8×8, 512×1, 1×512, etc., pixels.

We explored a configuration, called "block synchronization," in which all threads encounter a shared barrier after completing work on a given block. The idea is to force all threads to process blocks in lock-step, with the intent of inducing better memory utilization by having the multiple worker threads accessing nearby regions of the source data. For the tests run in this study, we use the dynamic work assignment algorithm on multi-core CPU implementation, as in Nieh et al. [29]. We determined through empirical testing that it performs better than the static work assignment approaches. Those results are not shown here due to space limits, and are consistent with earlier findings [29, 31] that suggest static work assignment algorithms can suffer from load imbalance.

3.1.3 Concurrency

To explore the effects of varying thread concurrency and of non-uniform memory access, we ran tests across lower concurrency levels than the number of available CPU cores. But, to maintain the same image and data volume size in these tests, we used a hybrid parallelism approach that combines the image-tile decomposition described above with a domain decomposition to distribute the data volume across groups of threads. In this way, we still ran as many threads as the number of cores by using multiple groups at the given concurrency level, with each group allocating its own memory buffer disjoint from the other groups (see Table 1). This functionality was already implemented for tests we conducted at large scale on distributed-memory systems using a hybrid parallelism approach [19]. For the CPU tests, we invoked the `-Mconcur=numa` flag in the PGI compiler to link in a thread affinity library that prevented threads from migrating across cores.

Platform	Threads per Buffer	Disjoint Buffers	Buffer Size
Intel/Nehalem	1	8	$256 \times 256 \times 256$
	2	4	$256 \times 256 \times 512$
	4	2	$256 \times 512 \times 512$
	8	1	$512 \times 512 \times 512$
AMD/MagnyCours	6	4	$256 \times 256 \times 512$
	24	1	$512 \times 512 \times 512$

Table 1: Data decomposition for varying concurrency levels.

3.2 Many-Core GPU Implementation

Our GPU raycasting kernel implementation is essentially the same as the CPU version. However, the way work blocks are assigned to GPU threads differs from the CPU implementation owing to the data-parallel nature of the CUDA language: the image is considered as a 2D CUDA grid that is divided into

CUDA thread blocks. Each thread block corresponds to an image tile, and each individual CUDA thread in each thread block performs raycasting of a single pixel in the image. Therefore, an $M \times N$ thread block corresponds to an $M \times N$ pixel image tile. The CUDA runtime manages the dynamic scheduling of thread blocks on the GPU, assigning up to eight blocks at a time to each of the GPU's multiprocessors [30]. The GPU uses hardware context switches between the blocks within a multiprocessor to hide memory latency.

We used version 3.2 of the CUDA compiler and runtime and version 270.41.19 of the NVIDIA driver. ECC support was enabled. During compilation, we targeted CUDA "compute capability" 2.0 [30] to take advantage of additional optimizations available on the Fermi-series architecture. We also configured shared memory to be used as L1 cache for global memory requests.

3.3 Memory Layout

We explore two alternative ways of laying out source data in memory. In the array-order layout, a 3D source volume S is indexed such that a data value at $S[i, j, k]$ is at memory location $i + j * xsize + k * xsize * ysize$. The Z-order method is essentially a way of laying out a 2^n tree in memory in 1D fashion. Unlike array-order indexing, Z-order indexing has the desirable property that at any $[i, j, k]$ location, accessing a point that is nearby in index space is also nearby in physical memory location. We use the formulation described by Pascucci and Frank [32] where we compute the Z-order index through a series of bitwise operations.

To optimize algorithm runtime for both memory layouts, we implement a data structure that supports rapid memory index computation for both layout schemes. During volume rendering, we compute the index of a $S[i, j, k]$ location for array-order indexing using two table lookups and three additions, and for Z-order indexing, using three table lookups and two logical OR operations. In our GPU implementation, these lookup tables are stored in the GPU's constant memory for fast access. In both implementations, we found that a single lookup operation was significantly faster than the $12n$ bitwise operations ($2n$ bit shifts, n bitwise ANDs, and n bitwise ORs for each dimension) needed to calculate the Z-order index on the fly, or the two additions and three multiplications to calculate the array-order index.

4 Experiment Results

4.1 Methodology

Given that we are using an image-order algorithm, and that we are dividing up the work by having each thread work on a subset of the total image, we adopt the term "block size" to refer to the size of the image tile assigned to a thread in the CPU version or a CUDA thread block in the GPU version. Our primary research goal is to determine how much performance varies as a function of block size and layout of data in memory by measuring runtime and cache utilization characteristics.

The runtime we measure is the elapsed time required to perform raycasting and store a resulting pixel buffer. It does not take into account I/O time for either storing an image or loading the data, which is known to account for a sizeable fraction of total rendering time in large-data visualization applications. We are not trying to solve the I/O problem, and our focus on rendering only is appropriate for many use cases, such as creating multiple images from the same dataset as is typical of interactive visualization applications. For this same reason, we do not include the time required by our CUDA implementation for memory transfers between host and device.

Our second performance metric is L2 cache misses, which offers a more detailed picture of memory traffic. To obtain this data, we use the Performance Application Programming Interface (PAPI)[1] for the CPU platforms. PAPI requires a kernel module as well as application instrumentation via a simple API to specify which hardware counters to monitor[2]. On the GPU platform, we used the profiler available

[1] PAPI website: `http://icl.cs.utk.edu/papi/index.html`

[2] For the Intel/Nehalem we measured L2 misses as the sum of the hardware counters `MEM_LOAD_RETIRED:LLC_MISS + MEM_LOAD_RETIRED:LLC_UNSHARED_HIT + MEM_LOAD_RETIRED:OTHER_CORE_L2_HIT_HITM`. For the AMD/MagnyCours, we used the counter `L2_CACHE_MISS:DATA`.

Platform	Cores/MPs	L1 Cache	L2 Cache	L3 Cache
AMD/MagnyCours	24	64 KB / core	512 KB / core	6 MB / 6-cores
Intel/Nehalem	8	64 KB / core	256 KB / core	8 MB / 4-cores
NVIDIA/Fermi	14	64 KB / MP	768 KB / 14-MPs	(none)

Table 2: The cache hierarchies of our test platforms.

with the CUDA Toolkit version 3.2[3]. Conventional thinking is that there is a strong correlation between lower memory traffic levels and increased performance, however this is not always the case [5].

4.1.1 Platforms and Datasets

For this study, we make use of several different multi- and many-core systems. We ran our tests on a single node of each of the following systems, each having core/cache configurations shown in Table 2:

- `carver.nersc.gov` has 400 nodes, each consisting of dual quad-core 2.67 GHz Intel 5550 Nehalem processors and between 24 GB and 48 GB of DDR3 RAM.
- `hopper.nersc.gov` has 6384 nodes, each consisting of dual twelve-core 2.1 GHz AMD 6172 MagnyCours processors and 32 GB of DDR3 RAM.
- `oscar.ccv.brown.edu` has two GPU test nodes, each consisting of dual 2.5 GHz Intel 5630 Nehalem processors and 24 GB of RAM. The GPU accelerator is a NVIDIA Fermi M2050, a many-core GPU with 448 "CUDA cores" grouped as 14 "multiprocessors" [30] sharing 3GB of GDDR5 memory. The M2050 is installed as a PCI-E x16v2 add-on card in these IBM iDataPlex nodes.

Our dataset is a 512^3 volume that we flattened from an adaptively-refined mesh produced by a combustion simulation. Figure 1 shows the rendering of the dataset across the 10 representative views we used for our tests. We chose these views to span a diverse range of memory stride and access patterns during the ray integration stage.

Our shading calculations require a gradient field with three elements per every element in the scalar data set. This gradient field can be precomputed and cached in memory, but it increases the memory footprint by a factor of three. Alternatively, the gradient field can be computed as needed inside the shading calculation. This choice poses a classic trade-off of storage space vs. compute time. We experimented with both methods, using central differences in both cases to compute the gradient vector, and found that on-the-fly computation was faster for both the CPU and GPU implementations by an average of 8.2% on the Intel/Nehalem, 19.4% on the AMD/MagnyCours, and 80.7% on the NVIDIA/Fermi. The additional cache pressure of loading in precomputed gradients appears to be more detrimental than the six-point stencil operation needed to recompute the gradient at each ray integration step. The stencil operation has significant spatial and temporal locality, however, since the scalar values that are needed are either spatially nearby (especially in the case of Z-ordered memory), or are loaded at a similar time by the previous or consequent integration step. Conversely, this locality is not exploited when loading precomputed gradients from a separate memory buffer.

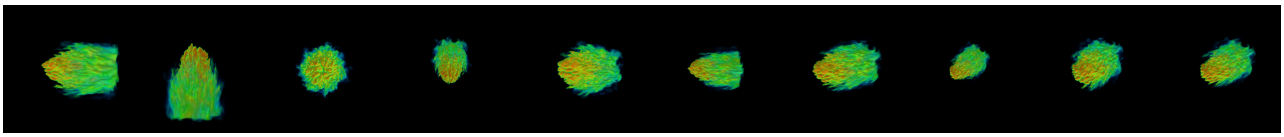


Figure 1: Output from our shared-memory parallel raycasting volume renderer, showing the 10 views we averaged our results over. Each frame is a 512×512 image rendered from a 512^3 dataset of a combustion simulation output. Simulation data courtesy J. Bell and M. Day, Lawrence Berkeley National Laboratory.

[3]We measured the `l2_subp0_read_sector_misses` counter available to compute capability 2.0 devices.

4.1.2 Parameter Sweep/Auto-tuning

The term "auto-tuning" has come to mean the process of evaluating the performance of all potential implementations of a kernel, potentially using some advanced search heuristics to avoid performing an exhaustive search of all possible permutations. In our case here, it means sweeping through tunable algorithmic parameters to find the combination resulting in the best performance. Unlike a "production" auto-tuning system, which would include a code generation phase followed by a lengthy run, we are performing only the parameter sweep to obtain performance measurements for each such configuration.

In our CPU tests, we explore 64 different image tile sizes where width and height both vary over $\{1, 2, 4, 8, 16, 32, 64, 128\}$. A complete test battery consists of approximately 5120 tests on the AMD/MagnyCours: 10 views, 64 block sizes, with and without early ray termination (ERT), two memory layouts, and two concurrency levels, and only the dynamic work assignment. The test battery on the Intel/Nehalem was similar, with three work assignment methods but only 8-way concurrency, for a total of 7680 tests. In addition, we reran another 7680 tests on the Intel/Nehalem at 1-way, 2-way, and 4-way concurrency, but with only the dynamic work assignment, in order to measure the runtime variation across concurrency levels reported in Table 3.

On the GPU, we ran 880 tests over twenty-two different thread block sizes, with width and height varying over $\{1, 2, 4, 8, 16\}$ (excluding the 1×1, 1×2, and 2×1 blocks), across two memory layouts, and with and without ERT. We excluded the blocks with fewer than four threads because the NVIDIA/Fermi requires at least four threads per block to saturate the computational throughput of a warp of execution (as described in more detail later in Section 4.5).

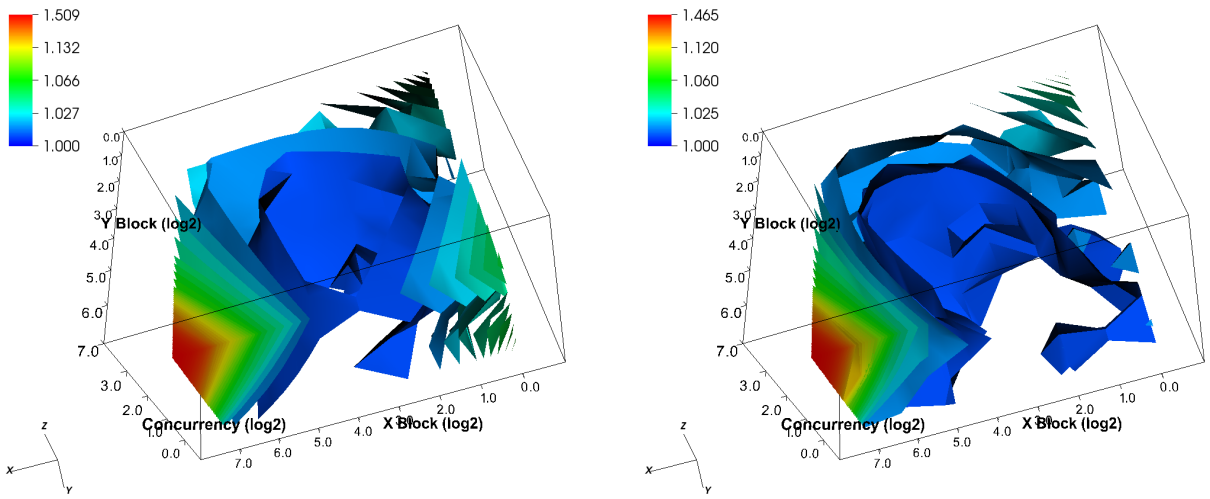
Platform	Concurrency	Array-Order	Z-Order	Array + ERT	Z + ERT
Intel/Nehalem	1	10.2%	7.7%	2.8%	2.6%
	2	7.4%	10.1%	9.7%	9.1%
	4	32.8%	28.7%	30.3%	27.7%
	8	55.1%	52.8%	50.9%	46.5%
AMD/MagnyCours	6	102.7%	100.7%	93.6%	92.4%
	24	247.2%	254.1%	241.5%	242.5%
NVIDIA/Fermi	-	74.8%	255.0%	78.9%	265.1%

Table 3: Percent variation in runtime (averaged over 10 views) across all block sizes, broken down by levels of concurrency, memory layout, and ERT. Block size has a dramatic impact on performance, particularly as concurrency increases to all available CPU cores. On the GPU, variation was greatest for Z-ordered memory, because the performance of the fastest block configurations increased considerably with Z-ordered access.

4.2 Variation in Runtime

To begin, we first wished to determine if block size impacts performance, and if so, by how much, as measured overall by percent variation. The summary results, shown in Table 3, indicate that block size can have a dramatic impact on performance. From Table 3, we draw several conclusions. First, the amount of variation increases with increasing concurrency on the CPUs. This result is expected since there is increased opportunity for reuse of data that is shared among more threads. Block sizes that better utilize the cache hierarchy can perform much better, increasing the variation in performance when compared to less optimal block sizes. Second, block size has a nearly equal effect on variation for both array- and Z-ordered layouts in CPU memory. Although Z-ordering improves the spatial locality of data within the same 3D neighborhood, it is still sensitive to changes to spatial locality caused by different block sizes. Third, variation is greater for Z-ordering in GPU memory by more than a factor of three when compared to array-ordering (see Section 4.7 for more details).

Figure 2 amplifies the results shown in Table 3 for the Intel/Nehalem platform. The parameter sweep through block sizes at a single concurrency produces a 2D array of performance data. However, we are interested in the patterns that may emerge across concurrency levels, so by "stacking" the 2D arrays at



(a) Array-Order

(b) Z-Order

Figure 2: Runtimes (averaged over 10 views, then normalized across concurrency level relative to the best runtime) from the Intel/Nehalem across four levels of concurrency presented in 3D using isocontours. Two of the axes are block sizes, while the third axis is concurrency. Larger and smaller block sizes correspond to poorer performance, particularly at higher concurrency levels, while medium-sized blocks produce relatively better performance at all concurrency levels.

each concurrency level, we have a 3D dataset to which we can apply traditional visualization techniques for inspection. The corners enclosed by the red contour correspond to block sizes that performed worse relative to others at that concurrency level. The innermost blue contour shows the region of best performance, which is comprised of medium-sized blocks and is consistent across concurrency levels. We see relatively lower variation at lower concurrency levels, and the largest relative outliers (the red contour) at 8-way concurrency. Again, both memory layouts exhibit similar performance characteristics in terms of variation across block size.

It is likely that larger block sizes suffer from poor load balancing since the granularity of the image tiles is so much larger. In this application, the amount of work each thread performs while processing an image tile is a function of scene characteristics, and there is not necessarily the same amount of actual work per block. In addition, the raycasting algorithm employs an "empty-space skipping" [28] optimization that skips pixels that are predetermined not to lie within the bounding box of the oriented 3D volume after it has been projected onto the 2D view plane. Therefore, image tiles that have less coverage of the 3D volume can end up calculating substantially fewer ray integration steps.

4.3 Intel/Nehalem

On the Intel/Nehalem, medium-sized blocks show the best performance in terms of both faster runtime and lower levels of L2 cache misses (Figure 3) regardless of work assignment algorithm or memory layout. With smaller block sizes, we also see consistently higher runtimes. This result is likely due to a relatively lower level of temporal cache coherence, as evidenced by the relatively higher rate of L2 cache misses associated with smaller block sizes.

With the static work assignment algorithm, we conducted an experiment aimed at inducing temporal cache coherency. This algorithmic option, which appears in Figures 3 as "sync," forces all threads synchronize at the block level. The intent is to force all threads to access nearby memory locations in a synchronous fashion. This configuration performed so poorly that we abandoned further investigation. The poor performance is most likely due to load imbalance. In contrast, the dynamic work assignment shows better overall performance due to its better load balancing characteristics.

The Z-ordered memory layout results in consistently better performance due to better use of the memory hierarchy. We see consistently lower levels of L2 cache misses when using the Z-ordered layout.

		Array Order																Z Order													
		No ERT								ERT								No ERT								ERT							
		1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
Static	1	2.53	2.45	2.44	2.41	2.43	2.85	4.00	3.78	2.46	2.37	2.27	2.23	2.25	2.64	3.69	3.54	2.25	2.21	2.18	2.16	2.19	2.75	3.56	3.28	2.19	2.06	2.00	1.98	2.01	2.52	3.40	3.09
	2	2.44	2.46	2.39	2.38	2.42	2.85	4.00	3.75	2.32	2.31	2.26	2.23	2.25	2.67	3.65	3.49	2.30	2.17	2.15	2.12	2.18	2.72	3.61	3.30	2.10	2.01	1.98	1.97	2.01	2.52	3.35	3.08
	4	2.52	2.46	2.35	2.35	2.40	2.82	3.93	3.77	2.32	2.26	2.26	2.23	2.25	2.65	3.75	3.52	2.17	2.13	2.13	2.12	2.16	2.73	3.60	3.36	2.13	1.99	1.98	1.97	2.01	2.57	3.34	3.09
	8	2.53	2.47	2.36	2.36	2.38	2.84	4.01	3.78	2.29	2.23	2.22	2.22	2.24	2.64	3.68	3.55	2.30	2.15	2.13	2.11	2.16	2.71	3.60	3.34	2.01	1.97	1.98	1.97	2.00	2.50	3.31	3.14
	16	2.57	2.52	2.39	2.34	2.39	2.84	4.02	3.86	2.22	2.21	2.23	2.21	2.24	2.66	3.70	3.53	2.29	2.20	2.14	2.11	2.15	2.71	3.61	3.38	1.97	1.99	1.96	1.97	2.00	2.51	3.34	3.12
	32	2.56	2.51	2.39	2.35	2.38	2.84	3.99	3.80	2.21	2.23	2.21	2.20	2.25	2.64	3.71	3.52	2.32	2.17	2.13	2.11	2.15	2.72	3.59	3.43	2.02	1.99	1.96	1.96	2.00	2.55	3.34	3.13
	64	2.73	2.49	2.38	2.35	2.38	2.83	3.95	3.79	2.24	2.21	2.20	2.21	2.24	2.64	3.66	3.58	2.35	2.19	2.14	2.12	2.15	2.73	3.57	3.39	1.97	1.98	1.96	1.96	2.00	2.50	3.31	3.15
	128	2.58	2.45	2.38	2.36	2.39	2.86	4.01	3.85	2.23	2.22	2.21	2.21	2.25	2.64	3.67	3.56	2.39	2.20	2.15	2.13	2.16	2.72	3.64	3.43	2.02	1.97	1.97	1.96	2.00	2.52	3.31	3.18
Static/Sync	1	4.83	4.19	4.35	3.41	3.70	4.42	4.40	4.07	4.73	4.57	4.16	3.52	3.78	4.14	4.00	3.89	4.06	4.23	3.78	3.46	3.60	4.07	3.96	3.71	4.04	4.24	3.91	3.52	3.41	3.76	3.74	3.45
	2	4.82	3.68	3.17	3.60	3.54	4.25	4.29	4.09	4.63	3.37	3.01	3.36	3.41	4.06	3.87	3.76	4.22	4.07	3.44	3.05	3.34	3.88	3.89	3.71	4.00	3.67	3.13	3.09	3.21	3.68	3.64	3.40
	4	4.09	3.14	2.94	3.27	3.57	4.24	4.23	4.18	3.96	3.31	2.88	3.26	3.41	3.98	3.98	4.04	3.48	3.60	2.82	2.96	3.23	3.89	3.80	3.70	3.44	3.43	2.57	2.87	3.15	3.64	3.63	3.53
	8	3.72	3.10	3.13	2.98	3.41	4.19	4.29	4.17	3.39	3.36	3.01	2.82	3.27	3.96	4.03	3.99	3.77	2.97	2.76	2.86	3.26	3.78	3.93	3.61	2.97	2.72	2.75	2.69	3.09	3.62	3.65	3.47
	16	3.87	2.88	2.88	2.93	3.46	4.17	4.26	4.20	3.44	3.17	2.70	2.78	3.22	3.92	3.96	3.86	3.22	3.15	2.55	2.69	3.25	3.80	3.90	3.67	3.22	2.78	2.42	2.55	3.05	3.59	3.63	3.48
	32	2.94	2.62	2.66	2.91	3.45	4.17	4.13	4.37	2.82	2.57	2.62	2.74	3.24	3.90	3.94	4.03	3.22	2.46	2.48	2.67	3.22	3.75	3.85	3.90	2.96	2.31	2.32	2.56	3.02	3.57	3.54	3.64
	64	2.72	2.60	2.68	2.86	3.43	4.17	4.18	4.77	2.61	2.49	2.56	2.71	3.21	3.93	3.81	4.40	2.35	2.34	2.42	2.69	3.20	3.72	3.76	4.22	2.30	2.25	2.32	2.51	3.00	3.54	3.54	3.99
	128	2.60	2.57	2.65	2.90	3.39	4.13	4.12	5.86	2.49	2.44	2.50	2.70	3.17	3.89	3.87	5.36	2.46	2.28	2.40	2.61	3.10	3.62	3.66	5.09	2.20	2.18	2.30	2.50	2.94	3.46	3.47	4.64
Dynamic	1	2.48	2.41	2.38	2.36	2.35	2.35	2.35	2.35	2.24	2.17	2.14	2.13	2.12	2.12	2.12	2.12	2.35	2.29	2.25	2.24	2.23	2.23	2.23	2.23	2.08	2.03	2.00	1.99	1.99	1.99	1.99	1.99
	2	2.41	2.36	2.34	2.33	2.33	2.34	2.34	2.35	2.17	2.14	2.12	2.11	2.10	2.11	2.11	2.12	2.29	2.26	2.23	2.22	2.21	2.21	2.22	2.24	2.04	2.01	1.99	1.98	1.97	1.97	1.97	1.99
	4	2.37	2.34	2.32	2.31	2.31	2.33	2.34	2.36	2.14	2.11	2.10	2.09	2.09	2.10	2.11	2.13	2.26	2.24	2.22	2.21	2.20	2.20	2.21	2.25	2.02	2.00	1.98	1.97	1.96	1.96	1.97	2.00
	8	2.36	2.33	2.31	2.30	2.31	2.32	2.34	2.39	2.12	2.10	2.09	2.08	2.09	2.10	2.12	2.15	2.23	2.22	2.21	2.20	2.19	2.19	2.21	2.28	2.00	1.99	1.97	1.96	1.96	1.96	1.97	2.02
	16	2.37	2.33	2.31	2.30	2.30	2.32	2.36	2.41	2.13	2.10	2.09	2.08	2.08	2.10	2.13	2.18	2.22	2.21	2.20	2.19	2.19	2.19	2.22	2.30	1.99	1.98	1.96	1.96	1.96	1.96	1.97	2.04
	32	2.39	2.33	2.32	2.30	2.31	2.33	2.39	2.48	2.16	2.11	2.09	2.08	2.09	2.11	2.16	2.24	2.21	2.20	2.19	2.19	2.18	2.19	2.24	2.35	1.98	1.97	1.96	1.95	1.95	1.96	2.00	2.09
	64	2.42	2.35	2.33	2.31	2.32	2.37	2.46	2.71	2.18	2.13	2.10	2.09	2.10	2.14	2.24	2.51	2.21	2.20	2.19	2.18	2.19	2.22	2.30	2.57	1.97	1.97	1.96	1.95	1.96	1.99	2.07	2.34
	128	2.44	2.37	2.34	2.32	2.33	2.42	2.67	3.57	2.20	2.15	2.12	2.10	2.11	2.20	2.44	3.14	2.21	2.20	2.19	2.19	2.20	2.27	2.48	3.34	1.97	1.96	1.96	1.95	1.97	2.03	2.23	2.86

(a) Runtime (s)

		Array Order																Z Order													
		No ERT								ERT								No ERT								ERT							
		1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
Static	1	28.7	23.1	20.7	18.9	18.4	21.7	30.6	29.0	15.8	9.6	6.7	5.4	4.8	5.1	7.4	6.5	23.7	21.1	18.5	17.2	16.7	21.3	28.0	25.7	14.1	8.7	6.0	4.7	4.2	4.7	6.3	6.2
	2	24.4	23.5	19.6	18.5	18.4	21.9	30.4	28.6	10.2	6.8	5.0	4.3	4.5	5.0	7.4	6.8	24.6	18.7	18.0	16.9	16.8	21.4	28.0	25.7	9.3	6.0	4.1	3.5	3.8	4.7	6.3	5.9
	4	24.7	21.5	18.6	18.4	18.4	21.8	30.7	28.7	8.7	5.2	4.1	3.9	4.3	5.0	7.4	7.6	20.0	18.0	17.3	16.8	16.8	21.3	28.1	26.0	8.4	4.7	3.3	3.1	3.6	4.6	6.3	6.2
	8	24.1	20.9	18.7	18.3	18.4	21.8	30.8	28.6	7.4	4.7	3.4	3.6	4.2	4.9	7.4	6.9	21.6	18.2	17.4	16.6	16.7	21.2	28.3	25.7	6.2	3.8	2.7	2.8	3.5	4.6	6.4	5.8
	16	22.4	21.4	19.0	18.3	18.3	21.9	30.7	29.0	6.1	4.3	3.2	3.4	4.2	4.9	7.4	6.7	20.6	19.0	17.3	16.7	16.7	21.3	28.1	25.8	5.1	3.5	2.6	2.7	3.4	4.6	6.3	5.8
	32	23.4	20.3	19.1	18.3	18.3	21.8	30.7	29.1	5.9	4.0	3.2	3.4	4.2	4.9	7.4	6.7	20.6	18.2	17.0	16.4	16.7	21.3	28.0	25.6	5.3	3.6	2.4	2.6	3.5	4.6	6.3	5.9
	64	24.5	20.0	18.5	18.0	18.2	21.8	30.7	29.0	6.1	3.9	3.0	3.3	4.1	4.9	7.4	6.8	21.6	18.1	16.9	16.3	16.6	21.2	28.2	25.9	4.9	3.3	2.5	2.5	3.4	4.6	6.3	5.8
	128	22.6	19.0	18.8	17.9	18.2	21.8	30.6	30.1	5.5	3.8	3.1	3.2	4.1	4.9	7.4	6.8	21.3	17.6	16.7	16.2	16.5	21.3	28.1	27.1	4.9	3.2	2.3	2.5	3.4	4.5	6.3	6.0
Static/Sync	1	28.7	25.2	22.0	19.6	19.6	22.8	30.8	30.7	15.5	10.6	8.3	6.4	5.6	5.7	7.9	8.6	25.0	23.0	20.1	18.1	17.6	22.1	28.5	26.2	13.0	9.1	7.4	5.6	5.0	5.4	6.5	6.7
	2	25.2	23.3	20.9	19.4	19.2	22.4	30.8	30.3	11.8	8.2	6.0	5.2	4.9	5.4	7.7	7.0	23.0	20.9	19.0	17.6	17.2	21.7	28.2	27.1	9.4	7.1	5.1	4.3	4.2	5.0	6.4	6.6
	4	23.7	21.6	19.8	18.7	18.7	22.1	30.5	29.7	9.4	6.5	4.9	4.3	4.6	5.2	7.6	7.3	21.4	20.2	18.3	17.2	17.1	21.6	28.3	25.9	7.7	5.6	3.9	3.5	3.9	4.8	6.4	6.7
	8	22.5	20.4	19.6	18.4	18.5	22.1	30.9	29.3	8.8	5.7	4.1	3.9	4.4	5.0	7.7	7.1	20.1	18.7	17.9	16.9	17.0	21.4	28.3	25.6	7.0	4.7	3.4	3.0	3.6	4.6	6.3	6.0
	16	22.2	19.8	19.2	18.2	18.5	21.9	30.7	29.2	7.7	4.8	3.6	3.6	4.3	5.0	7.6	6.9	19.6	18.4	17.3	16.8	16.8	21.4	28.3	25.8	6.3	4.2	2.9	2.8	3.5	4.6	6.5	5.9
	32	21.7	19.8	18.6	18.3	18.4	21.8	30.6	29.2	6.8	4.6	3.3	3.4	4.2	5.0	7.5	6.8	19.7	17.7	17.1	16.6	16.7	21.3	28.2	25.9	5.9	3.8	2.5	2.5	3.4	4.6	6.4	5.8
	64	21.1	19.4	18.6	18.1	18.3	21.8	30.6	29.0	6.4	4.0	3.1	3.3	4.2	5.0	7.5	6.8	19.1	17.9	16.9	16.5	16.7	21.3	28.2	26.0	5.4	3.4	2.4	2.5	3.4	4.6	6.3	5.9
	128	21.6	19.1	18.5	17.9	18.3	21.8	30.6	30.0	6.2	3.8	2.9	3.3	4.2	4.9	7.4	6.9	19.0	17.7	16.9	16.4	16.7	21.2	28.0	27.4	4.8	3.2	2.2	2.5	3.4	4.5	6.3	5.8
Dynamic	1	23.3	20.6	19.0	18.1	17.7	17.6	17.6	17.8	21.0	18.7	17.2	16.5	16.1	16.0	16.2	16.2	12.4	8.3	6.2	5.2	4.7	4.5	4.4	4.4	11.3	7.4	5.5	4.6	4.1	3.9	3.9	3.8
	2	20.7	19.0	18.1	17.7	17.6	17.6	17.7	17.7	18.8	17.3	16.5	16.2	16.0	16.0	16.2	16.3	8.2	5.6	4.3	4.1	4.3	4.4	4.5	4.4	7.4	4.9	3.7	3.4	3.6	3.7	3.8	3.8
	4	19.7	18.2	17.8	17.6	17.5	17.7	17.8	18.3	17.8	16.6	16.2	16.0	16.0	16.0	16.2	16.5	6.3	4.3	3.4	3.5	4.0	4.3	4.4	4.4	5.6	3.7	2.8	2.8	3.3	3.6	3.7	3.9
	8	19.0	17.9	17.6	17.5	17.5	17.7	17.8	18.3	17.2	16.3	16.1	16.0	16.0	16.1	16.4	16.7	5.4	3.6	2.9	3.2	3.9	4.2	4.3	4.6	4.7	3.1	2.4	2.6	3.2	3.6	3.7	3.9
	16	18.8	17.8	17.7	17.6	17.6	17.8	18.2	19.0	17.0	16.2	16.1	16.0	16.0	16.1	16.7	17.3	4.9	3.4	2.8	3.1	3.8	4.2	4.3	4.7	4.3	2.8	2.2	2.4	3.2	3.5	3.7	4.0
	32	18.9	17.9	18.0	17.7	17.7	17.9	18.4	19.5	17.1	16.3	16.3	16.1	16.2	16.3	16.9	17.8	4.7	3.3	2.7	3.1	3.8	4.1	4.3	5.0	4.1	2.7	2.2	2.4	3.2	3.6	3.7	4.2
	64	19.3	18.2	18.1	17.8	17.9	18.2	19.9	21.7	17.3	16.4	16.4	16.2	16.2	16.9	17.8	20.5	4.7	3.3	2.7	3.1	3.9	4.2	4.5	5.4	4.1	2.8	2.2	2.4	3.2	3.7	3.9	4.8
	128	19.3	18.8	18.2	17.9	17.8	18.9	21.5	28.3	17.7	17.1	16.5	16.2	16.3	17.2	20.1	25.6	4.7	3.4	2.8	3.2	3.9	4.3	4.8	6.5	4.2	2.9	2.3	2.4	3.3	3.7	4.2	5.7

(b) L2 cache misses (millions)

Figure 3: Runtime (a) and L2 cache misses (b) averaged across 10 views for 8-way concurrency on the Intel/Nehalem platform with varying block size, memory layout, work assignment method, and ERT.

4.4 AMD/MagnyCours

The AMD/MagnyCours shows similar trends to the Intel/Nehalem: medium-sized blocks performed best, and there are benefits for both ERT and Z-ordering (see Figure 4).

We ran tests at two different concurrencies to look for effects due to non-uniform memory access (NUMA), where a core on one socket accesses memory with affinity to a different socket. Although our MagnyCours test system has two twelve-core processors in two physical sockets, these 24 cores actually share four memory controllers, so we use the term "NUMA socket" to refer to the set of six cores that

		Array Order																Z Order													
		No ERT								ERT								No ERT								ERT							
		1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
6 Threads	1	1.58	1.49	1.42	1.39	1.38	1.38	1.38	1.38	1.54	1.45	1.39	1.36	1.35	1.34	1.34	1.34	1.50	1.42	1.39	1.35	1.33	1.32	1.32	1.32	1.46	1.37	1.34	1.31	1.29	1.29	1.29	1.29
	2	1.51	1.45	1.40	1.38	1.37	1.37	1.37	1.38	1.47	1.41	1.37	1.34	1.33	1.33	1.34	1.34	1.45	1.40	1.36	1.33	1.31	1.31	1.32	1.32	1.41	1.36	1.32	1.30	1.28	1.28	1.28	1.29
	4	1.48	1.44	1.39	1.37	1.36	1.36	1.37	1.38	1.44	1.40	1.36	1.33	1.33	1.33	1.33	1.35	1.42	1.38	1.34	1.32	1.30	1.31	1.32	1.33	1.38	1.35	1.31	1.29	1.27	1.27	1.29	1.29
	8	1.46	1.43	1.39	1.36	1.35	1.36	1.37	1.40	1.43	1.39	1.35	1.33	1.32	1.32	1.33	1.35	1.40	1.36	1.33	1.31	1.30	1.30	1.32	1.34	1.35	1.31	1.29	1.27	1.27	1.27	1.28	1.30
	16	1.45	1.42	1.38	1.36	1.35	1.36	1.38	1.43	1.41	1.38	1.35	1.33	1.32	1.32	1.34	1.38	1.39	1.34	1.32	1.30	1.30	1.30	1.33	1.37	1.34	1.32	1.28	1.27	1.27	1.27	1.29	1.33
	32	1.43	1.40	1.37	1.36	1.36	1.36	1.41	1.51	1.40	1.37	1.34	1.32	1.32	1.34	1.38	1.44	1.37	1.33	1.31	1.30	1.30	1.31	1.36	1.45	1.33	1.30	1.28	1.27	1.27	1.29	1.33	1.38
	64	1.42	1.40	1.37	1.36	1.36	1.39	1.49	1.87	1.39	1.36	1.33	1.32	1.32	1.37	1.45	1.74	1.36	1.32	1.31	1.30	1.31	1.34	1.43	1.79	1.33	1.29	1.27	1.27	1.27	1.32	1.39	1.66
	128	1.43	1.40	1.37	1.37	1.38	1.46	1.75	2.74	1.39	1.36	1.34	1.33	1.34	1.41	1.69	2.55	1.34	1.32	1.31	1.31	1.33	1.40	1.67	2.61	1.31	1.29	1.28	1.28	1.29	1.36	1.62	2.43
24 Threads	1	1.78	1.63	1.54	1.48	1.44	1.43	1.43	1.43	1.59	1.45	1.37	1.32	1.30	1.29	1.28	1.28	1.32	1.22	1.18	1.16	1.14	1.15	1.14	1.15	1.24	1.14	1.09	1.06	1.03	1.03	1.03	1.03
	2	1.69	1.58	1.51	1.46	1.43	1.43	1.43	1.43	1.52	1.42	1.36	1.31	1.28	1.28	1.28	1.28	1.25	1.17	1.15	1.14	1.13	1.13	1.14	1.14	1.12	1.06	1.03	1.02	1.01	1.02	1.02	1.03
	4	1.65	1.56	1.50	1.45	1.42	1.43	1.43	1.42	1.47	1.39	1.34	1.30	1.27	1.28	1.28	1.27	1.20	1.14	1.12	1.11	1.11	1.12	1.13	1.14	1.09	1.03	1.01	1.01	1.00	1.02	1.02	1.03
	8	1.63	1.55	1.49	1.45	1.41	1.41	1.40	1.40	1.44	1.37	1.32	1.29	1.27	1.27	1.26	1.26	1.18	1.13	1.11	1.10	1.10	1.12	1.13	1.14	1.06	1.02	1.00	0.99	0.99	1.01	1.01	1.03
	16	1.60	1.53	1.48	1.44	1.41	1.41	1.41	1.47	1.44	1.38	1.33	1.29	1.26	1.26	1.27	1.35	1.17	1.12	1.10	1.10	1.10	1.11	1.14	1.18	1.05	1.01	1.00	0.99	0.99	1.01	1.03	1.09
	32	1.59	1.53	1.48	1.43	1.41	1.42	1.50	1.92	1.43	1.38	1.33	1.29	1.25	1.27	1.37	1.71	1.16	1.12	1.10	1.10	1.10	1.12	1.19	1.52	1.05	1.01	0.99	0.99	0.99	1.01	1.10	1.39
	64	1.57	1.50	1.47	1.44	1.43	1.50	1.90	2.81	1.40	1.35	1.32	1.29	1.28	1.38	1.73	2.45	1.16	1.12	1.10	1.10	1.11	1.18	1.50	2.19	1.05	1.01	1.00	0.99	1.00	1.10	1.39	1.97
	128	1.58	1.49	1.45	1.46	1.52	1.72	2.70	4.86	1.40	1.33	1.31	1.30	1.36	1.58	2.40	4.29	1.16	1.12	1.11	1.11	1.17	1.34	2.11	3.84	1.05	1.01	1.00	1.01	1.05	1.24	1.89	3.39

(a) Runtime (s)

		Array Order																Z Order													
		No ERT								ERT								No ERT								ERT							
		1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
6 Threads	1	7.4	5.7	3.9	2.9	2.4	2.2	2.1	2.1	7.2	5.5	3.8	2.8	2.4	2.1	2.0	2.0	5.1	3.3	2.3	1.8	1.6	1.5	1.5	1.5	4.9	3.2	2.2	1.8	1.6	1.5	1.5	1.4
	2	4.9	3.8	2.6	1.9	1.7	1.8	1.9	2.0	4.7	3.7	2.5	1.9	1.6	1.7	1.9	1.9	3.2	2.1	1.5	1.2	1.1	1.1	1.3	1.4	3.2	2.1	1.5	1.2	1.1	1.1	1.3	1.4
	4	3.5	2.8	2.0	1.5	1.3	1.6	1.9	2.0	3.4	2.7	1.9	1.4	1.3	1.5	1.8	1.9	2.3	1.5	1.1	0.9	0.8	0.9	1.2	1.4	2.3	1.5	1.1	0.9	0.8	0.9	1.2	1.4
	8	3.0	2.4	1.6	1.2	1.2	1.5	1.8	2.0	2.9	2.3	1.6	1.1	1.1	1.5	1.7	1.9	1.9	1.2	0.9	0.7	0.7	0.8	1.2	1.4	1.8	1.2	0.8	0.7	0.6	0.8	1.2	1.3
	16	3.2	2.2	1.5	1.1	1.1	1.5	1.8	2.0	3.0	2.1	1.4	1.0	1.0	1.4	1.7	1.9	1.7	1.1	0.8	0.6	0.6	0.8	1.2	1.4	1.6	1.1	0.7	0.6	0.6	0.7	1.1	1.4
	32	3.6	2.2	1.4	1.1	1.0	1.5	1.8	2.1	3.4	2.1	1.4	1.0	1.0	1.4	1.7	1.9	1.6	1.0	0.7	0.6	0.5	0.8	1.2	1.5	1.5	1.0	0.7	0.6	0.5	0.7	1.2	1.4
	64	3.7	2.2	1.4	1.1	1.0	1.5	2.0	2.8	3.5	2.1	1.4	1.0	1.0	1.4	1.9	2.6	1.6	1.0	0.7	0.6	0.5	0.8	1.3	1.8	1.5	1.0	0.7	0.6	0.5	0.7	1.3	1.8
	128	3.7	2.2	1.4	1.0	1.0	1.6	2.4	4.2	3.6	2.1	1.4	1.0	1.0	1.5	2.3	3.9	1.6	1.0	0.7	0.6	0.5	0.8	1.6	2.9	1.6	1.0	0.7	0.5	0.5	0.8	1.5	2.7
24 Threads	1	11.8	10.3	9.5	9.0	8.7	8.5	8.4	8.3	10.6	9.1	8.3	7.8	7.6	7.5	7.5	7.4	4.2	2.7	2.0	1.7	1.5	1.5	1.4	1.4	3.8	2.4	1.8	1.5	1.3	1.3	1.2	1.2
	2	10.5	9.5	9.0	8.8	8.7	8.6	8.5	8.4	9.4	8.5	8.0	7.8	7.6	7.5	7.5	7.4	2.8	1.9	1.5	1.3	1.3	1.3	1.4	1.4	2.5	1.7	1.3	1.2	1.1	1.1	1.2	1.3
	4	9.8	9.2	8.9	8.7	8.7	8.7	8.6	8.4	8.8	8.1	7.8	7.6	7.5	7.5	7.5	7.3	2.1	1.4	1.2	1.1	1.2	1.3	1.4	1.4	1.9	1.3	1.0	0.9	1.0	1.1	1.2	1.2
	8	9.5	9.0	8.7	8.7	8.5	8.6	8.4	8.3	8.4	7.9	7.7	7.6	7.5	7.6	7.5	7.4	1.8	1.2	1.0	1.0	1.1	1.2	1.3	1.4	1.6	1.1	0.9	0.8	0.9	1.1	1.2	1.2
	16	9.3	8.8	8.6	8.6	8.6	8.6	8.6	8.8	8.3	7.9	7.6	7.6	7.5	7.6	7.7	7.7	1.6	1.1	1.0	0.9	1.0	1.2	1.3	1.4	1.4	1.0	0.8	0.8	0.9	1.0	1.2	1.3
	32	9.1	8.8	8.6	8.6	8.7	8.7	9.0	10.9	8.1	7.8	7.6	7.6	7.5	7.5	8.0	9.9	1.5	1.1	1.0	1.0	1.0	1.3	1.5	1.8	1.4	0.9	0.8	0.7	0.9	1.0	1.3	1.6
	64	9.0	8.4	8.4	8.5	8.7	9.1	11.3	16.0	8.0	7.4	7.5	7.6	7.7	8.2	10.1	14.1	1.5	1.1	1.0	1.0	1.0	1.3	1.8	2.6	1.3	0.9	0.8	0.7	0.8	1.1	1.6	2.3
	128	9.0	8.3	8.3	8.6	9.4	10.3	15.7	28.5	7.9	7.3	7.3	7.6	8.2	9.4	14.0	24.9	1.5	1.1	1.0	0.9	1.3	1.6	2.6	4.5	1.3	1.0	0.8	0.8	0.8	1.3	2.2	3.9

(b) L2 cache misses (millions)

Figure 4: Runtime (a) and L2 cache misses (b) averaged across 10 views on the AMD/MagnyCours platform with varying block size, memory layout, concurrency level, and ERT.

share a memory controller.

In the 6-way concurrency test, we allocated four disjoint memory buffers and used the first-touch policy of the Linux kernel to set the affinity of each buffer to its own NUMA socket. In this test, we executed four groups of six threads, with each group scheduled on its own NUMA socket. Our theory is that this configuration should lead to more uniform memory access, and therefore better performance. Surprisingly, we found the opposite to be true. At 24-way concurrency, with only a single memory buffer shared by threads executing across all 24 cores, performance was better across all conditions: block size, memory layout, and with and without ERT, even though this meant that 18 of those cores were accessing memory on a different NUMA socket.

Moreover, Figure 4 shows that the block sizes that exhibited the fewest L2 cache misses are in the 6-way concurrency condition, even though those block sizes had longer runtimes than their corresponding trials in the 24-way concurrency condition. We suspect that there are other important effects at the L3 cache level that we are not capturing with our L2 cache measurements. Exploring this issue further and refining our measurement of the cache hierarchy will be the subject of future work.

4.5 NVIDIA/Fermi

On the GPU, we see a wide variation in runtime across different block sizes (Figure 5). Both Z-ordering and ERT show benefits for the more optimal block sizes. The best configurations on the NVIDIA/Fermi

		Array Order					Z Order				
		1	2	4	8	16	1	2	4	8	16
NoERT	1			1.00	0.77	0.74			0.90	0.58	0.44
	2		0.98	0.71	0.63	0.68		0.86	0.51	0.35	0.30
	4	1.04	0.70	0.59	0.61	0.74	0.89	0.51	0.32	0.26	0.33
	8	0.81	0.63	0.61	0.68	0.73	0.56	0.34	0.25	0.27	0.32
	16	0.83	0.71	0.72	0.68	0.71	0.42	0.30	0.27	0.27	0.32
ERT	1			0.92	0.70	0.66			0.83	0.53	0.40
	2		0.90	0.64	0.57	0.60		0.79	0.47	0.32	0.27
	4	0.95	0.64	0.53	0.54	0.65	0.82	0.46	0.29	0.23	0.29
	8	0.73	0.56	0.54	0.60	0.63	0.51	0.31	0.23	0.24	0.28
	16	0.74	0.62	0.62	0.59	0.61	0.38	0.27	0.24	0.24	0.28

(a) Runtime (s)

		Array Order					Z Order				
		1	2	4	8	16	1	2	4	8	16
NoERT	1			50	115	481			25	28	42
	2		36	57	238	632		17	17	22	67
	4	52	59	179	517	907	14	14	16	43	240
	8	139	243	527	839	873	14	17	43	182	202
	16	636	724	909	827	834	29	63	176	166	166
ERT	1			43	95	401			22	24	35
	2		29	48	193	530		15	15	19	56
	4	38	47	144	429	773	12	12	14	37	198
	8	106	192	436	711	741	12	14	37	153	163
	16	506	596	765	696	701	24	53	148	137	130

(b) L2 cache misses (millions)

Figure 5: Runtime (a) and L2 cache misses (b) averaged over 10 views for different thread block sizes on the NVIDIA/Fermi with varying memory layout and ERT. Grey boxes indicate thread blocks with too few threads to fill a warp of execution.

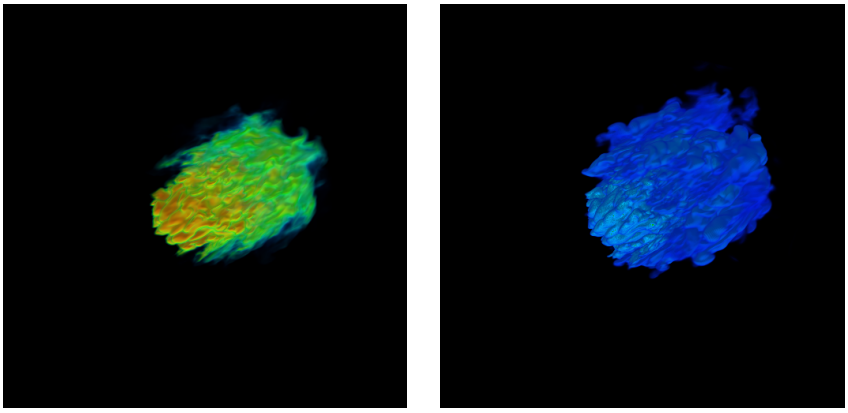
are the 4×8 and 8×4 thread blocks with Z-ordering and ERT. Performance appears to vary with the total number of threads in a block, leading to the diagonal striping in Figure 5.

Even though CUDA thread blocks require a minimum of 32 threads to saturate computational throughput, many of the block sizes with 16 or fewer threads still perform well because of the branching nature of our algorithm, which causes divergence among CUDA threads. A thread block is executed in a single-instruction-multiple-thread (SIMT) fashion in which "warps" of 32 threads are executed across four clock cycles in subsets of eight threads that share a common instruction. If those eight threads do not share a common instruction, such as when conditionals cause branching code paths, the threads diverge and must be executed serially. This situation is prevalent in our algorithm. For example, imagine a thread block owning a region of the image that only partially covers the data volume. Some of the threads immediately exit because of the empty-space skipping optimization in our algorithm, while the other threads proceed to cast rays through the volume. Even the threads that proceed together with raycasting may have rays of different lengths, which will cause divergence and load imbalance.

Since a warp must be scheduled across at least four clock cycles, using fewer than four threads per thread block will guarantee under-utilization, so we excluded those configurations from our sweep. Empirically, the sweet spot for thread block size is 16 or 32 threads, depending on the memory ordering and whether ERT is enabled. Many block sizes with 16 threads perform well even though this is less than the warp size of 32 threads, indicating the complex interaction of the CUDA runtime and warp scheduler in handling branching for this particular algorithm and problem. It is also likely that larger thread blocks exhibited greater load imbalance because the variation in ray lengths tends to increase with block size.

Surprisingly, the small thread blocks that displayed the worst performance also exhibited the fewest L2 cache misses (see Figure 5). Yet, the converse is not true: the most optimal block sizes don't show the most L2 cache misses. Instead, L2 cache misses appear to rise uniformly with the total number of threads in a block, leading to the same diagonal striping as seen in the runtime plot. Therefore, we conclude that achieving the best performance on the GPU is a trade-off between using enough threads to saturate a warp and using few enough to maintain good cache utilization. We also find that, as in the CPU tests, L2 cache misses are systematically less when using the Z-ordered memory layout on the GPU because of the improved spatial locality.

Interestingly, the NVIDIA CUDA Programming Guide [30] says: "The effect of execution configuration on performance for a given kernel call generally depends on the kernel code. Experimentation is therefore recommended." Our experiments show a wide variation in performance depending upon thread block size. While there is little surprise that such variation exists, the amount of variation – as much as 265% – is somewhat unexpected, as is the fact that the optimal block size for one problem is not the same as for another problem when run on the same platform.



(a) Transfer Function "A" (b) Transfer Function "B"

Figure 6: Two different transfer functions have different benefits from early ray termination, but also yield images that accentuate different features of the underlying dataset. For the results presented in this paper, we used transfer function "A."

4.6 Algorithmic Optimization: Early Ray Termination

We tested an algorithmic optimization called early ray termination (ERT), where a conditional within the inner-most loop that iterates along an individual ray tests the condition of the ray reaching full opacity. If it has, the loop is aborted since any further steps will not contribute to the color or opacity of that pixel. This optimization is dependent both on the input data and on the transfer function that determines how data values map to color and opacity. For our specific data set and transfer function, we see approximately 10% fewer integration steps when ERT is enabled, which is directly reflected in our reported runtimes: in cases where ERT is enabled, we see a 10-15% improvement in absolute runtime.

To demonstrate the relationship between the transfer function and the benefits of ERT, we ran an additional test with a"shallower" transfer function (called "B" in Figure 6) that did not penetrate as deep into the volume. With ERT enabled, this transfer function exhibited 19.7% fewer integration steps and ran 19.1% faster, as expected. However, the image rendered with transfer function "B" has important differences in the areas and features of the dataset that are visualized, which may or may not be appropriate according to the application.

The benefits of ERT are highly dependent upon scene characteristics and transfer function, so it is impossible to say with certainty how much ERT will help algorithm performance in general. In earlier work investigating the use of ERT in raycasting volume rendering [22], an opaque volume exhibited a 300% improvement in runtime, while a semi-transparent volume exhibited no improvement in runtime from ERT and instead sustained a 30% penalty from the additional conditional overhead. This conditional can have an adverse impact on performance, particularly for GPU implementations where divergent code paths are costly. In an early test case where there was no reduction in raycasting steps from ERT for a particular scene, we measured a penalty of 5%. In summary, the effects of ERT are highly variable: there are combinations of data sets and transfer function where ERT has no effect, and others where it lead to a dramatic reduction in the number of integration steps.

4.7 Algorithmic Optimization: Z-ordered Memory

Z-ordered memory outperforms array-ordered memory on all platforms and at all concurrency levels, and these performance gains increases with concurrency (see Figure 7). The benefits of Z-ordering at higher concurrency are likely due to the larger penalties for non-contiguous access and cache misses in shared memory systems that service many cores, as on the AMD/MagnyCours and NVIDIA/Fermi. While the Intel/Nehalem and AMD/MagnyCours show a modest gain from Z-ordering, ranging from 3% to 29%, the NVIDIA/Fermi exhibits gains of 146% at 64-way concurrency, i.e. 64 threads per thread block.

As more cores access memory through a shared memory controller and subsystem, the improved locality of Z-ordered memory access and correspondingly lower L2 miss rates becomes increasingly

Figure 7: The performance gains (averaged across thread block size, with ERT) from using Z-ordered memory instead of array-ordered increase with concurrency. The gains are most notable on the NVIDIA/Fermi, where we count the number of threads per block as the concurrency.

beneficial. Thus, the NVIDIA/Fermi, whose memory subsystem must service 14 multiprocessors each with many thread blocks in flight, sees the biggest improvements from Z-ordering. While the slower block configurations improve some with Z-ordering, the best configurations improve greatly, leading to larger variation for Z-ordering than array-ordering.

Because many datasets are not already stored with a Z-ordered layout, there is in practice a cost associated with re-ordering the data buffer, which we have not included. However, in the use case where multiple visualizations will be generated from the same data set, this initial cost is amortized.

5 Conclusions and Future Work

The main point of this study has been to explore the relationship between tunable algorithm parameters and known algorithmic optimizations and the resulting impact on performance of a staple visualization algorithm on modern multi-core and many-core platforms. Our results suggest a wide variation in performance can result – up to 254% on multi-core CPUs and 265% on many-core GPUs. We used the findings of this study to set tunable algorithmic parameters for a set of extreme-concurrency runs [18, 19] that required literally millions of CPU hours; by finding and using optimal settings for tunable algorithmic parameters, we in effect saved millions of additional CPU hours that would have been spent executing an application in a non-optimal configuration.

This work, which uses a well-established methodology for finding optimal performance, shows that such a methodology can be useful for visualization algorithms as well, and that the algorithmic parameters that produce the best performance vary from problem to problem and platform to platform, often in a non-obvious way. Our GPU results underscore this result: the best performance results in what appears to be a crossover point between cache utilization and thread warp size and thread divergence that occurs due to this particular algorithm. This approach helps empirically establish algorithmic parameters that may be difficult, if not impossible, to determine using a performance model that predicts

performance bounds, and has been widely used in the computational science community with great success [21, 7, 5].

The algorithm we study, raycasting volume rendering with perspective projection, uses an unstructured, or irregular memory access pattern. Each ray will, in effect, execute a different traversal path through memory, and the set of paths and the number of computational steps each requires is a function of runtime parameters, such as viewpoint, data set, and color transfer function. Other visualization algorithms exhibit similar memory access characteristics, like parallel streamline computation [3], and could benefit from this performance optimization methodology.

An avenue for future work would be to expand the scope of algorithmic parameters and options and discover how they inform performance within the context of the auto-tuning approach. For example, would using an interleaved memory layout for data/gradients produce better performance than the current non-interleaved approach? Does the cost of branch prediction in the inner ray integration loop outweigh the benefit of an acceleration option like early ray termination? In object-order (bricking) approaches, what is the relationship between brick size/shape and actual observed cache utilization? How well would a bricking strategy work on the GPU, and does its use improve memory access patterns on the GPU? Would an alternate encoding of data in bricks, such as using a space-filling curve layout, result in better or worse utilization of the memory hierarchy, and what is the relationship of that performance with brick size and shape? How does a change in final image resolution alter the choice of block size? Could predictive performance models be enhanced to include some statistical estimation of runtime given known variability in algorithm performance that depends upon data or runtime characteristics? What is the impact of multi-field data, particularly with respect to memory utilization, and do the lessons learned for scalar data apply to complex data types?

Another avenue of future work would be to explore the efficacy of alternative programming environments, like MapReduce [9], for use on multi- and many-core systems. Recent work by Stuart and Owens [36] pursues this line of investigation for several computational kernels. Interestingly, their implementation of a GPU-capable MapReduce library "relaxes" some of the MapReduce semantics to expose GPU-centric features, like thread block size, though it is not clear if this kind of control, which is necessary for auto-tuning, exists outside of Stuart and Owens GPMR library, for example on multi-core CPU implementations. Even though our shared-memory implementation does not have an explicit "reduce" step, one desirable outcome would be the ability to write code once that runs on many different types of platforms. Given that the settings that produce optimal performance vary by platform, dataset, and other runtime attributes, there seems to be a clear benefit to using an auto-tuning methodology to find those that produce optimal performance rather than by-hand coding.

References

[1] C. Bajaj, I. Ihm, G. Joo, and S. Park. Parallel Ray Casting of Visible Human on Distributed Memory Architectures. In *VisSym'99 Joint EUROGRAPHICS-/IEEE TCCG Symposium on Visualization*, pages 269–276, 1999.

[2] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[3] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy. Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17:1702–1713, 2011.

[4] X. Cavin, C. Mion, and A. Filbois. COTS Cluster-based sort-last rendering: Performance Evaluation and Pipelined Implementation. In *Proceedings of the 2005 IEEE Visualization Conference*, 2005.

[5] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, 2009.

[6] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Ar-

chitectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[7] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point Stencil for Multicore. In *4th International Workshop on Automatic Performance Tuning (iWAPT)*, 2009.

[8] R. de la Cruz and M. Araya-Polo. Towards a Multi-level Cache Performance Model for 3D Stencil Computation. In *Procedia Computer Science, Proceedings of the International Conference on Computational Sciences, ICCS*, volume 4, pages 2145–2155, 2011.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[10] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 12, Washington, DC, USA, 2003. IEEE Computer Society.

[11] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, 1988.

[12] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. Bergeron, and P. Hatcher. Large Data Visualization on Distributed Memory Multi-GPU Clusters. In *Proceedings of High Performance Graphics 2010*, pages 57–66, 2010.

[13] T. Fogal and J. Krüger. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, pages 139–146, Nov. 2010.

[14] J. D. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 2. edition, 1990.

[15] E. Gobbetti, F. Marton, and J. A. I. Guitián. A Single-pass GPU Ray Casting Framework for Interactive Out-of-core Rendering of Massive Volumetric Datasets. *The Visual Computer*, 24(7):797–806, 2008.

[16] S. Grim, S. Bruckner, A. Kanistar, and E. Gröller. A Refined Data Addressing and Processing Scheme to Accelerlate Volume Raycasting. *Computers and Graphics*, 5(28):719–729, 2004.

[17] J. Hollingsworth and A. Tiwari. End-to-end Auto-tuning with Active Harmony. In D. H. Bailey, R. F. Lucas, and S. W. Williams, editors, *Performance Tuning of Scientific Applications.* CRC Press, 2010.

[18] M. Howison, E. W. Bethel, and H. Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPVG)*, Norrköping, Sweden, May 2010.

[19] M. Howison, E. W. Bethel, and H. Childs. Hybrid Parallelism for Volume Rendering on Large, Multi- and Many-core Systems. *IEEE Transactions on Visualization and Computer Graphics*, 99(PrePrints), 2011.

[20] W. M. Hsu. Segmented ray casting for data parallel volume rendering. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pages 7–14, New York, NY, USA, 1993. ACM.

[21] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-tuning framework for Parallel Multicore Stencil Computations. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[22] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[23] A. Law and R. Yagel. Multi-frame thrashless ray casting with advancing ray-front. In *GI '96: Proceedings of the conference on Graphics interface '96*, pages 70–77, Toronto, Ont., Canada, Canada, 1996. Canadian Information Processing Society.

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

[25] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *PRS '95: Proceedings of the IEEE symposium on Parallel rendering*, pages 23–30, New York, NY, USA, 1995. ACM.

[26] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 15–22. ACM Press, October 1993.

[27] L. Marsalek, A. Hauber, and P. Slusallek. High-speed Volume Ray Casting with CUDA. In *IEEE Symposium on Interactive Ray Tracing*, 2008. Poster.

[28] C. Müller, M. Strengert, and T. Ertl. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proceedings of Eurographics Parallel Graphics and Visualization*, pages 59–66, 2006.

[29] J. Nieh and M. Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 17–24. ACM Siggraph, October 1992.

[30] NVIDIA Corporation. *NVIDIA CUDA^{TM} Programming Guide Version 3.2 RC*, 2010. `http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html`.

[31] M. E. Palmer, B. Totty, and S. Taylor. Ray Casting on Shared-Memory Architectures: Memory-Hierarchy Considerations in Volume Rendering. *IEEE Concurrency*, 6(1):20–35, 1998.

[32] V. Pascucci and R. J. Frank. Global Static Indexing for Real-time Exploration of Very Large Regular Grids. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 2–2, New York, NY, USA, 2001. ACM.

[33] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *SIGGRAPH Comput. Graph.*, 22(4):51–58, 1988.

[34] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.

[35] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003.

[36] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 1068–1079, May 2011.

[37] R. Tiwari and T. L. Huntsberger. A Distributed Memory Algorithm for Volume Rendering. In *Scalable High Performance Computing Conference*, Knoxville, TN, USA, May 1994.

[38] C. Upson and M. Keeler. V-buffer: visible volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 59–64, New York, NY, USA, 1988. ACM.

[39] S. Williams, K. Datta, L. Oliker, J. Carter, J. Shalf, and K. Yelick. Auto-tuning Memory-Intensive Kernels for Multicore. In D. H. Bailey, R. F. Lucas, and S. W. Williams, editors, *Performance Tuning of Scientific Applications*. CRC Press, 2010.

[40] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.