Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures

Mikhail Smelyanskiy, David Holmes, Jatin Chhugani, Alan Larson, Douglas M. Carmean, Dennis Hanson, Pradeep Dubey, Kurt Augustine, Daehyun Kim, Alan Kyker, Victor W. Lee, Anthony D. Nguyen, Larry Seiler, and Richard Robb

Abstract—Medical volumetric imaging requires high fidelity, high performance rendering algorithms. We motivate and analyze new volumetric rendering algorithms that are suited to modern parallel processing architectures. First, we describe the three major categories of volume rendering algorithms and confirm through an imaging scientist-guided evaluation that ray-casting is the most acceptable. We describe a thread- and data-parallel implementation of ray-casting that makes it amenable to key architectural trends of three modern commodity parallel architectures: multi-core, GPU, and an upcoming many-core Intel® architecture code-named Larrabee. We achieve more than an order of magnitude performance improvement on a number of large 3D medical datasets. We further describe a data compression scheme that significantly reduces data-transfer overhead. This allows our approach to scale well to large numbers of Larrabee cores.

Index Terms—Volume Compositing, Parallel Processing, Many-core Computing, Medical Imaging, Graphics Architecture, GPGPU.

1 Introduction

The past two decades have seen unprecedented growth in the amount and complexity of digital medical image data collected on patients in standard medical practice. The clinical need to accurately diagnose disease and develop treatment strategies in a minimally-invasive manner has required developing new image acquisition methods, high resolution acquisition hardware, and novel imaging modalities. All of these place computational burdens on the ability to synergistically use the image information. With increasing quality and utility of medical image data, clinicians are under pressure to generate more accurate diagnoses or therapy plans. The challenge is to provide improved health care efficiently, which is complicated by the magnitude of the data. Despite the availability of several general purpose and specialized rendering engines, volume visualization has not been widely adopted by the medical community except in certain specific cases [2, 5].

There are several barriers to adopting volume visualization in the clinic, including the quality of visualization and overall performance of rendering engines *on commodity hardware*. While real-time volume rendering has been shown using GPU [15], performance is usually gained at the cost of image quality. Custom rendering hardware solutions have been developed to provide high-quality rendering, but the cost associated with these systems limits their wide-spread adoption. Examples include the VolumePro [24], which uses a custom rendering chip, and Nvidia Tesla [23], which uses a standard graphics chip in a special purpose product. To ease adoption, it is desirable to provide medical-quality rendering using commodity hardware. The purpose of this work is to evaluate *medical-quality* volume rendering on modern commodity parallel hardware including a general purpose CPU (Intel® architecture code-named Nehalem), a GPU (Nvidia GeForce GTX280), and a many-core architecture (Intel Larrabee).

Main Contributions: Our contributions are as follows:

- Through an image scientist-guided evaluation, we demonstrate ray-casting to be the most acceptable of three volume rendering techniques for high-fidelity requirements of medical imaging.
- We map, evaluate and compare performance of two ray-casting implementations on three modern parallel architectures. We optimize our implementation to take full advantage of each architecture's salient hardware features.
- We demonstrate that our sub-volume based implementation of ray-casting, designed for low memory bandwidth and high SIMD efficiency, achieves best performance on all three architectures.
- To mitigate the overhead of data transfer and take advantage of wide SIMD units, we propose and evaluate robust lossless compression schemes with fast SIMD-friendly decompression.

Results Summary: Our parallel implementation of ray-casting delivers close to 5.8x performance improvement on quad-core Nehalem over an optimized scalar baseline version running on a single core Harpertown. This enables us to render a large $750x750x1000$ dataset in 2.5 seconds. In comparison, our optimized Nvidia GTX280 implementation achieves from 5x to 8x speed-up over the scalar baseline. In addition, we show, via detailed performance simulation, that a 16-core Intel Larrabee [26] delivers around 10x speed-up over single core Harpertown, which is on average 1.5x higher performance than a GTX280 at half the flops. At higher core count, performance is dominated by the overhead of data transfer, so we developed a lossless SIMD-friendly compression algorithm that allows 32-core Intel Larrabee to achieve a 24x speed-up over the scalar baseline.

The remainder of the paper is organized as follows. We describe a clinical study of three volume rendering algorithms in Sec. 2. Section 3 discusses challenges in mapping ray-casting to modern parallel architectures and presents our implementation of sub-volume algorithm to address these challenges. Section 4 describes the three architectures and the medical datasets used in the evaluation, followed by the architectural characteristics of ray-casting in Sec. 5 and detailed performance analysis in Sec. 6. We summarize our findings in Sec. 7.

2 Evaluation of Volume Rendering Techniques

There are several approaches to direct volume rendering. Each balances performance and quality. In this section we compare three classic approaches to volume rendering and demonstrate that ray-casting is the preferred method for diagnosis due to its quality.

2.1 Methods for Direct Volume Rendering

The most direct approach to volume rendering is ray-casting [6]. The traditional implementation of ray-casting, which is used in our baseline application, traces rays through a volume in the viewing direction.

- *Mikhail Smelyanskiy, Jatin Chhugani, Douglas M. Carmean, Pradeep Dubey, Daehyun Kim, Alan Kyker, Victor W. Lee, Anthony D. Nguyen and Larry Seiler are with Intel Corporation, Contact E-mail: mikhail.smelyanskiy@intel.com.*
- *David Holmes, Alan Larson, Dennis Hanson, Kurt Augustine and Richard Robb are with Mayo Clinic.*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

The volume is segmented: each voxel is labeled with the ID of an object the voxel belongs to. When a ray hits a voxel, the object ID is used to determine whether the voxel is visible. If so, it is shaded using a surface normal, which is generated by calculating a gradient from the voxel data. This is done using a 3x3x3 filter, which has fewer aliasing artifacts than a smaller filter [11]. The surface normal is combined with the color of the voxel as well as the color of the segmented object to generate a new composited color. This method is generally recognized as the slowest of the methods.

In order to overcome the high computational complexity of ray-casting, several alternative approaches to volume rendering were developed. Splatting [30] projects individual voxels onto the screen. As the result, the voxels are visited in contiguous fashion and only once. This reduces computational complexity compared to ray-casting and takes advantage of the SIMD architecture found in modern processors. Techniques like voxel over-sampling and depth normalization [21] are used to reduce the rendering artifacts. We implemented a custom splatting algorithm for this evaluation.

The shear-warp method by Lacroute et al. [16] shears the volume data in order to generate distorted intermediate images. Ray-casting is then applied to the data in order to generate the final rendering. Shearing the data allows a one to one mapping between volume slice and the image plane and as a result is also SIMD-friendly. Techniques such as interpolation of intermediate slices and use of smoother opacity transfer functions [27] aid in reducing the resulting artifacts. We used VolPack library for shear-warp rendering.

While alternative algorithms change image quality, there are several alternative optimizations which ensure the quality of the data while still improving the performance. We divide these optimizations into two categories. In the first category there are algorithms designed for specific hardware features. These include hardware-accelerated pre-integration [25], coherent ray tracing [29], and interval SIMD arithmetic [13]. In some cases, these methods require pre-processing which can increase the initial rendering time or the memory required by the method [10]. On the other hand there are algorithms that are designed to efficiently manage image data. For example, ray-casting algorithms can adopt a presence acceleration and/or early termination strategies [18] and [4] to avoid unnecessary computation. Multi-resolution trees can store the compressed data for efficient retrieval of contributing voxels [14].

2.2 Quality Comparison

In order to determine which rendering methods generate clinically useful renderings, we rendered high-resolution CT Angiography (CTA) data using these three methods and conducted a blind comparison. Each CTA was acquired isotropically with resolution of .742 mm on a side and covered the entire pelvis. A transfer function was chosen to clearly delineate the inferior epigastric artery (IEA) – a vessel that is critical to the outcome of certain reconstructive surgeries. The IEA can be difficult to visualize because of its size and position in the pelvis.

After selecting the transfer function, we used each method to generate volume renderings spanning 180 degrees around the data. Our blind comparison application presented pairs of corresponding renderings to the user in a random order. Each of the rendering methods was paired with the other rendering methods during the comparison. Five different individuals reviewed the relative quality of the paired images. Quality was determined by the fidelity of the image in terms of visibility of the branches of the IEA and sharpness of the vessel.

Table 1 shows the preference of each observer for one method over another. We pooled the results and compared them using a Fisher exact statistical significance test [7] for categorical data. The Fisher exact test specifically excludes the cases where the two images are considered equivalent; the number of comparisons that were deemed equivalent is also shown.

Ray-casting was statistically preferred over either of the other methods ($p<0.0001$ for both comparisons). Splatting was preferred over shear-warp ($p<0.001$). Although statistically different, splatting and ray-casting had the most number of equivalent renderings. Statistical analysis were not conducted for each observer seperately because of the low sample size, but according to the results, Observer B preferred the Shear Warp renderings. Following the evaluation, the observers commented that the shear-warp images were more blurry when viewed at a non-orthogonal angle, and splatting showed an artifact in some of the non-orthogonal views. Representative images of orthogonal and non-orthogonal renderings are shown in Figure 1. As a result of this blind comparison, it was determined that only the ray-casting method has acceptable quality for medical applications.

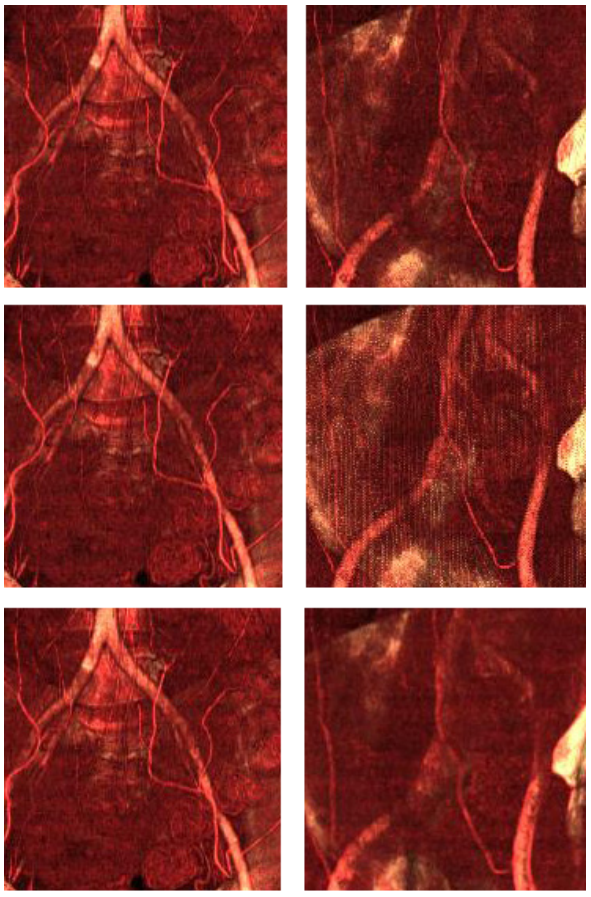


Fig. 1: **Comparison of three rendering methods.** Ray-casting (top), Splatting variant (middle), and Shear-warp (bottom). When viewed from an orthogonal angle (left side), the results are consistent. When viewed obliquely (right side), artifacts and blurring is present.

3 Ray-casting implementation

This section describes challenges of mapping ray-casting to modern many-core architectures and proposes two approaches to overcome these challenges. This work focusses on volume compositing [17] and orthographic projection, which are both widely used clinically (for example, planning pedicle screw placement in the spine [1]).

3.1 Challenges in Mapping to Modern Architectures

Modern processors are equipped with vector execution units and multiple cores. Vectorization and parallelization are necessary to harness the massive computing capability in these processors. Parallelizing ray-casting is relatively trivial: every cast ray can be traced through the volume independently from every other ray. However, vectorizing ray-casting is not as simple because of irregular data access and control flow divergence. Moreover, some modern parallel systems require high data transfer overhead over a slow interface line, such as PCIe. When accounted for, this overhead can significantly limit performance of such systems.

In ray-casting, as the rays traverses through the volume, they access voxels with a non-constant stride. While caches capture some temporal and spatial locality in the neighboring accesses, the *irregular*

Table 1: **Results of Observer Study.** In a blinded manner, each observer selected a "preferred" rendering. If the two renderings were equivalent, the observer could select equivalent. The Fisher exact test was used to determine significance.

Observer	Casting/Splatting	Casting/Shear	Splatting/Shear
A	10/2	20/0	13/7
B	11/5	3/10	4/12
C	12/9	18/4	15/6
D	6/1	19/0	17/2
E	15/2	21/1	14/2
Equivalent	37	14	18
Total	54/19	81/15	63/29
	$p<0.0001$	$p<0.0001$	$p<0.001$

access pattern makes it difficult for hardware to prefetch data to reduce access latency. To improve data locality, our implementation traces a packet of rays that are spatially adjacent in the 2D image plane [28].

To exploit the vector (a.k.a. SIMD) architecture, the packets of 2D rays are processed in SIMD fashion. However, the control flow for each ray is data-dependent and often incoherent, which lowers SIMD efficiency. For example, some rays may exit the volume earlier than other rays. Such rays become inactive and should not be processed. We use predicated SIMD execution with vector masks to address this problem. As ray terminate, the corresponding SIMD lane is masked off; this prevents it from doing computation or accessing memory. A disadvantage of predicated execution is that as rays terminate, the SIMD efficiency drops. Re-packing active rays is done to maintain good SIMD efficiency [3].

Non-contiguous data accesses pose another challenge to SIMD processing. As we mentioned earlier, rays access non-contiguous locations in memory. To utilize SIMD execution, these data must be packed from memory to SIMD registers. Such operation is known as *gather* . Gathering non-contiguous data often requires multiple memory accesses which reduces SIMD efficiency. In volume compositing, large number of gathers comes from gradient estimation which requires accesses to each of 26 neighbors by each ray in a packet. It results in 26 gather operations and can be extremely costly for architectures with little or no hardware support for gather.

Lastly but not least, modern systems tend to have special processing units to accelerate operations. For example, volume rendering can be offloaded to accelerators such as Nvidia GTX280 [22], or more specialized hardware such as [24]. Such accelerators are connected to the host's main CPU via a slow interface line, such as PCIe, which can deliver at most 10GB/s bandwidth. In cases, where the time to transfer data to accelerators exceeds the speed of volume rendering computation, performance is limited by low bandwidth of the interface.

3.2 Sub-volume-based Rendering

To address the challenges in our base-line implementation of ray-casting, we adapt an implementation based on partitioning the full volume into sub-volumes. Each sub-volume is rendered independently into its local sub-image, which are combined to form one final image in a front-to-back order. Correctness follows from the linear and associative nature of the color/opacity computation. While this sub-volume based implementation has been proposed before [19], we are the first to evaluate this idea in the context of a many-core architecture.

The sub-volume approach provides solutions to the key problems of ray-casting. It significantly improves cache locality because the size of the sub-volume can be chosen to fit into available on-die memory. Pre-computing the gradients before processing each sub-volume replaces gathers with regular accesses that map well onto SIMD, at the cost of a small amount of temporary memory. Pre-computing gradients for the entire volume would require a prohibitive amount of memory. Finally, the sub-volume also mitigates data communication overhead over slow links, since it allows rendering one sub-volume to proceed in parallel with transferring the next sub-volume.

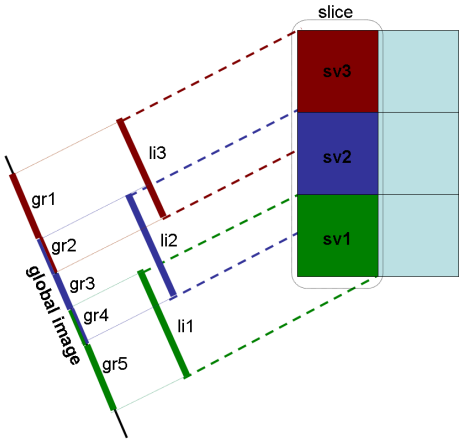


Fig. 2: **Sub-volume based rendering**. Three highlighted sub-volumes $sv1$, $sv2$ and $sv3$ are rendered in parallel into three local images $li1$, $li2$, and $li3$. Local images are combined to produce a single global image. Combining is performed in parallel on non-intersecting regions $gr1$ through $gr5$. Each region is computed by combining corresponding local images in front to back order.

In our SIMD implementation we project all sub-volumes in parallel into local images, which we then combine into the global image. The boundaries of the local images define regions in the global image. Each such region combines its local images sequentially in front to back order. Due to the fact that global regions do not intersect, multiple regions are combined in parallel. Figure 2 illustrates sub-volume algorithm with an example. Here three sub-volumes, $sv1$, $sv2$, and $sv3$, project into local images $li1$, $li2$, and $li3$ in parallel. The global image consists of five regions, $gr1$ through $gr5$. For example, global region $gr1$ is produced from $li3$, while $gr2$ combines $li3$ and $li2$. All five regions are computed in parallel. Our parallel combining algorithm renders one slice at a time in front-to-back order. Moreover, slices are chosen in the direction that is most perpendicular to the viewing plane. This assures that the projection of any sub-volumes within a slice will only overlap with the projection of at most eight of its immediate neighbors. This significantly reduces inter-thread communication as well as overhead of combining within overlapped regions.

3.3 Volume Rendering With Lossless Compression

Since the introduction of Computed Tomography (CT) in 1972, volumetric datasets have been continually growing in size. The first commercial CT scanners produced volumetric resolutions of 64^3 voxels. Datasets in current clinical practice can consist of 512^3 voxels and are expected to increase more than sixteen time to 2048^3 in the next few years. As mentioned in Section 3.1, transferring this massive amount of data may result in significant overhead.

One important technique to reduce this overhead is data compression. There are two key requirements for a compression algorithm for medical imaging (1) It should be lossless because we cannot tolerate any image quality degradation. (2) It should be fast enough to be real-time to support interactive clinical usage. To meet these requirements, we propose and implement application-specific data compression mechanisms for the two most important data structures. Our specialized algorithms compress much better than generalized compression schemes and achieve significant bandwidth savings. Although there exist other schemes like lossless JPEG and lossless wavelet compression [31], their corresponding compression/decompression times are prohibitively large. Finally, our scheme takes advantage of the wide SIMD units in today's processors.

Raw Volume Compression

A volume data structure is a 3D array containing raw data values. Each data element is a 16-bit integer. Our compression mechanism is based on the value locality property of medical volume datasets. We observe that data elements with spatial proximity exhibit similar

values. Therefore, instead of storing each piece of volume data using 16 bits, we store only the difference from a nearby data point. This value difference takes fewer bits due to the value locality. Specifically, an N^3 compression block is divided into multiple N^2 2D planes. For each 2D plane, we first store a base value, and then the difference of each data point from the base. We use the same number of bits to store the differences within each block, and hence the number of bits per element is dictated by the maximum difference computed at any volume element within the 2D slice. Another variant is to store difference from its direct neighbor instead of the base. Since the 2D planes can be oriented in any of the X, Y or Z directions, we choose the orientation that minimizes the length of the resultant compressed stream. A value of N equal to 4 produced the best compression ratios for the datasets we tested on.

The compression/decompression scheme described above maps well to a SIMD implementation. During decompression, we essentially load the data for multiple elements (*eight* in the case of 128-bit SSE), and expand the stored differences into 16-bit values. These values are then added to the base value to obtain the final values. Since different blocks may have different number of bits per element, we pre-compute and store the different shift patterns in a separate table, and at run-time simply look up the appropriate data shuffle patterns to expand the data into 16-bit values. In practice, the overhead of lookups is negligible. Section 6.2.1 has detailed analysis on the achieved compression ratios and decompression run-times.

Object Compression

An object data structure is a 3D array of the segmented object IDs. Each data element is represented by an 8-bit unsigned value. To compress object datasets, we exploit the knowledge that there are very few objects within a small region, often only one. For each N^3 compression block, we identify unique object IDs, for which we create a dictionary and assign indices. Then, instead of storing the actual 8-bit object IDs, we store the corresponding dictionary indices. The size of the dictionary index is usually smaller than the size of the object ID. We use the same number of bits for each dictionary index within a block, and hence the number of bits is dictated by the number of distinct object IDs within the block. Above all, about 50% of the blocks (for $N = 4$) in our datasets are completely contained inside a single object, and hence consist of a single distinct object ID. We optimize such cases by storing only one 8-bit value without any per-element dictionary index. The SIMD implementation for object data decompression is similar to the raw volume data decompression described above.

4 Experimental Environment

In this section, we describe our experimental setup, followed by the relevant hardware characteristics of the evaluated systems and present an overview of the analyzed medical datasets.

4.1 Experimental Testbed

We performed our experiments on the following three architectures.

Intel Quad-core CPU

Intel Nehalem is an x86-based multi-threaded multi-core architecture that offers four cores on the same die, running at 3.2 GHz. Nehalem cores feature an out-of-order superscalar microarchitecture, with newly added 2-way hyper-threading. In addition to scalar units, it has 4-wide SIMD units that support a wide range of SIMD instructions called SSE4 [12]. Each core is backed by a 256KB L2 cache. All four cores share an 8MB L3 cache.

Intel Larrabee x86-based Architecture

Larrabee [26] is a homogeneous many-core architecture based on small in-order IA cores. Each core is a general-purpose processor, which has a scalar unit based on the Pentium processor design, as well as a vector unit that supports 16 32-bit float or integer operations per clock. Vector instructions can be predicated by a mask register that controls which vector lanes are written. To improve SIMD efficiency, Larrabee has packed load and store instructions to enable bundling together sparse strands. This enables efficient re-packing of active rays. Larrabee also includes hardware support for gather and scatter operations, which allow loading from and storing to 16 non-contiguous memory addresses.

Larrabee has two levels of cache: low latency 32KB 1st level data cache and larger globally coherent 2nd level cache that is partitioned among the cores. Each core has a 256KB partition. To further hide latency, each core is augmented with 4-way multi-threading. Since Larrabee has fully coherent caches, standard programming techniques such as pthreads or OpenMP can be used for ease of programming.

We measure workload performance in terms of Larrabee units. A Larrabee unit is defined to be one Larrabee core running at 1 GHz, corresponding to a theoretical peak throughput of 32 GFLOPS, counting fused multiply-add as two operations. The machine configuration is chosen solely for ease of calculation. Real devices would ship with a variety of core counts and clock rates. Performance data were obtained from detailed simulations of 16, 32, and 64 core configurations. We performed these simulations on a cycle accurate system simulator, which is used and validated in designing Intel multi-core CPUs.

Nvidia GeForce GTX280 [22]

GTX280 is composed of an array of 30 multiprocessors. Each multiprocessor has 8 scalar processing units running at 1.3 GHz. The hardware SIMD structure is exposed to programmers through thread warps. Although it can handle thread divergence within a warp, it is important to keep such divergence to a minimum for best performance. To hide memory latency, GTX280 provides hardware multi-threading support that allows hundreds of thread contexts to be active simultaneously. To alleviate memory bandwidth, the card includes various on-chip memories such as multi-ported software-controlled 16KB memory and small non-coherent read-only caches. GTX280 can be programmed using the CUDA environment. It allows programmers to write a scalar program that is automatically organized into thread blocks across a set of parallel threads. Because GTX280 has very limited support for memory consistency and inter-thread synchronization, programmers should synchronize kernel invocations manually through the host to assure data consistency across parallel regions.

4.2 Datasets and Viewing Directions

Three sets of human 16-bit CT data are used in our evaluation. Table 2 shows image resolution and the size of each. The original data are collected at 1*mm* x 1*mm* x 1*mm* voxel resolution. The objects are segmented into bones, skin, blood pool, etc. Datasets ds1, ds3 and ds5 are rasterized with one transfer function, the remaining datasets are rasterized with another. Figure 3(a, b) shows representative renderings of ds1 and ds2. Performance characteristics such as memory access pattern vary drastically with viewing direction, as well as with changes in the transfer function. We report the average over a range of angles obtained by uniformly sampling a unit sphere for each dataset.

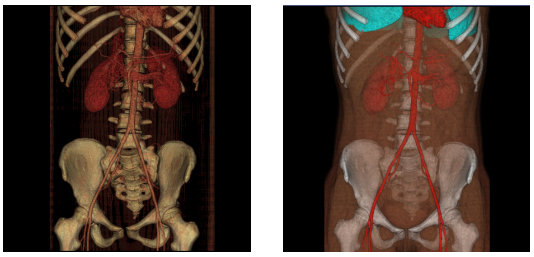
Table 2: **Dataset characteristics**. Six diverse datasets were used in evaluation. Each dataset consists of raw data grid (2 bytes per voxel) and object data grid (1 byte per voxel). Each grid is rendered into an image of comparable resolution.

Dataset	Data Resolution		Data Size (MB)	
	Grid	Image	Raw	Object
ds1&2	300x300x443	443x443	76	38
ds3&4	512x512x756	756x756	378	189
ds5&6	750x750x1107	1107x1107	1188	594

5 High-Level Characteristics of Volume Rendering

To gain insights into performance delivered by each evaluated platform, we provide high-level characteristics of two implementations of ray-casting algorithms: (1) the original, full-volume based and (2) the sub-volume based algorithm described in Section 3.2.

Execution Time Breakdown: Table 3 shows execution time breakdown for the full-volume implementation gathered on Harpertown platform. We see that ds1, ds3 and ds5, spend over 70% of execution time in



(a) (b)

Fig. 3: **Example of data renderings used in our evaluation:** (a) volume compositing rendering of torso, (b) volume compositing rendering with segmented lungs.

Dataset	ds1	ds3	ds5	ds2	ds4	ds6
Gradient Calc.	29%	24%	19%	61%	61%	61%
Compositing	71%	76%	81%	39%	39%	39%

Table 3: **Time breakdown of full-volume implementation**. Due to differences in transfer function, execution time breakdown differs across datasets.

the compositing function, whereas ds2, ds4 and ds6 spend most of their time in gradient calculation. This is due to the fact that transfer function used in ds2, ds4 and ds6 visualizes more objects. The sub-volume implementation incurs some overhead due to maintaining ray transitions between sub-volumes. It also performs more updates to the global image than full-volume due to combining local images. This overhead decreases with increasing sub-volume size. For 16^3 sub-volumes, the overhead ranges from 20% to 48% and for 64^3 sub-volumes, the overhead is only 4% to 18%. The sub-volume overhead is further amortized for images with more visual objects (ds2, ds4 and ds6).

Memory Accesses Characteristics: To understand memory access locality, we perform working set analysis by varying the size of the cache from 16KB to 32MB and measuring the miss rate. Figure 4 shows an analysis of four datasets. The top curve shows the working set for the original full-volume algorithm, while the other two curves show the working set for the sub-volume algorithm, sizes 16^3 and 32^3, respectively. We only present data for ds1, ds2, ds3 and ds4: ds5 is similar to ds3 and ds6 is similar to ds4. Several levels of working set are observed at the distinct knees in the curves. As cache size increases, eventually all data structures fit entirely in the cache and only compulsory misses remain. We make the following observations. First, for full-volume, ds1 and ds3 have larger working set than ds2 and ds4. This is due to the fact ds2 and ds4 spend more time in gradient calculation which has better locality than compositing. Second, the sub-volume algorithm requires a smaller working set than the full-volume algorithm. This is due to the fact that data accesses are localized to the sub-volumes which are much smaller than original data. For 16^3 sub-volumes, a 256KB cache is enough to capture most of the working set for all datasets, while for 32^3 sub-volumes, a 1MB cache is enough, since in both cases the miss rate is reduced to under 0.5%.

SIMD Characteristics: To better understand SIMD scalability, we characterize various SIMD overheads due to control incoherence and gathers. Rows two and three of Table 4 show percentage of control incoherence for SIMD width 4, 16 and 32. We report data for ds3 and ds4 - other datasets exhibit similar patterns. Ray packets of size 4, 16 and 32 are traced simultaneously. As long as all rays in the packet follow the same control flow path, they execute in lock step. When some of the rays diverge, they serialize, which reduces SIMD efficiency. We see that for 4-wide SIMD the loss of efficiency is only a few percent. While efficiency drops further for wider SIMD, it remains higher than 70% for all datasets on 32-wide SIMD. Overall volume rendering exhibits good control coherence even on architectures with wide SIMD.

Rows 5 and 6 shows the fraction of the total number of dynamic

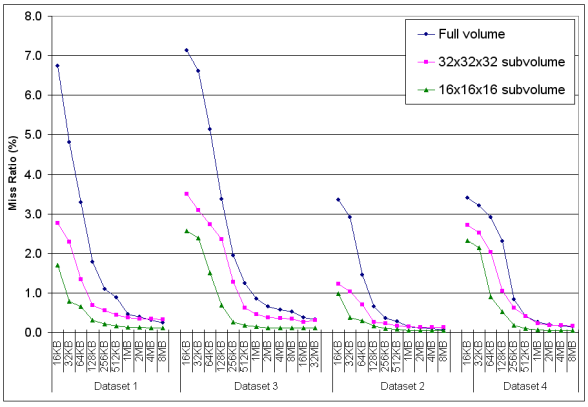


Fig. 4: **Working Set Analysis**. Sub-volume has smaller working set than full-volume. 256K cache reduces miss rate down to 0.5% when 16^3 sub-volume is used.

Table 4: **Characteristics of SIMD Level Parallelism**. Relatively small control incoherence even for 32-wide SIMD. Up to 6.8% of dynamic instructions are vector gathers. Half of the accesses within each gather are to a single 64-byte block. Tiling improves block locality.

Row	SIMD Width	4-wide	16-wide	32-wide
1	Control Incoherence			
2	ds3	7.7%	19.2%	27.9%
3	ds4	2.6%	10.2%	10.9%
4	% of data gathers			
5	ds3	4.3%	4.5%	4.0%
6	ds4	6.8%	6.8%	6.4%
7	Data Incoherence (average 64-byte blocks / gather)			
8	ds3	2.54	6.96	11.30
9	ds4	2.86	8.63	15.04
10	Data Incoherence after tiling (average 64-byte blocks / gather)			
11	ds3	2.03	4.14	7.20
12	ds4	2.29	5.13	8.91

instructions that are gathers from non-contiguous memory locations. We see that 7% of dynamic instructions are gathers. This can have significant performance impact on SIMD efficiency for architectures without hardware support for vector gathers.

To get further insight into gather overhead, rows 8 and 9 show the average number of contiguous 64-byte memory blocks accessed by each gather. These statistics are important because some architectures, such as Nvidia GTX280 and Intel Larrabee, can coalesce memory accesses from a single gather if they belong to same block. This reduces number of memory accesses and improves the performance. We see however that the average number of blocks per gather is quite high: roughly half of accesses within each gather are to distinct cache lines. This is expected, because in each traversal step rays in the packet touch voxels which are far apart in memory. On architectures with a limited number of memory ports or that cannot overlap gathers with other computation, this can significantly reduce SIMD efficiency. However, the average number of blocks can be reduced by tiling volume and object data, which captures spatial 3D locality that is absent in linear order. The last two rows show that tiling reduces the average number of blocks accessed per gather by 20% for 4-wide SIMD and 40% for 16- and 32-wide SIMD.

6 Analysis on Three Hardware Architectures

We mapped, optimized and analyzed full-volume and sub-volume implementations on three parallel architectures. Our optimizations take full advantage of each architecture's most relevant hardware features. Table 5 shows absolute performance (in ms) of our optimized baseline

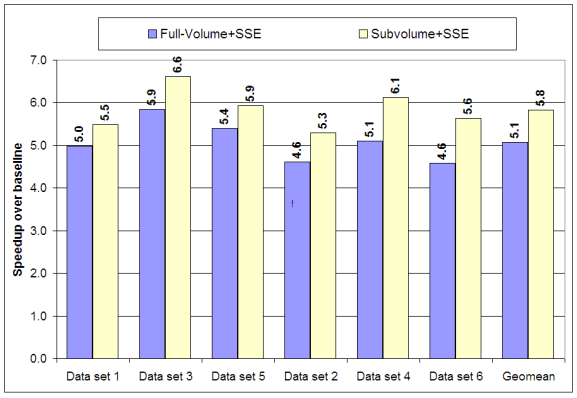


Fig. 5: **Performance on quad-core CPU**.Sub-volume (right bar) is faster than full-volume (left bar) due to improved cache locality and fewer gathers.

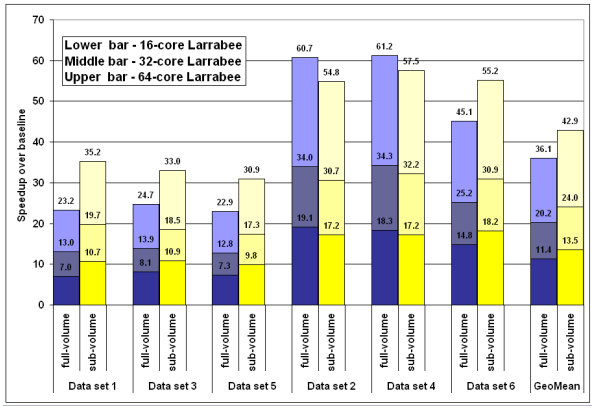


Fig. 6: **Performance on Larrabee without data transfer time**. Scaling is linear with 16 (lower bar), 32 (middle bar) and 64 (upper bar) cores. Hardware gather enables 2x-3.5x improvement due to SIMD.

implementation, which implements full-volume ray-casting and was measured on one core of Intel Harpertown running at 3.2 GHz. The rest of this section reports performance as the speed-up over this baseline. We also use geomean to average performance across datasets.

Table 5: **Absolute performance of baseline volume rendering.** Runtime in ms of scalar full-volume running on single core of the CPU.

Dataset	ds1	ds3	ds5	ds2	ds4	ds6
Time(ms)	311	1,566	4,663	1,017	4,889	14,218

6.1 Intel Nehalem

Our parallel implementation of full-volume on a quad-core CPU partitions the ray packets among threads. The sub-volume implementation partitions sub-volumes among threads. We also mapped both algorithms to SSE4. To map control statements, we used *_mm_blendv_ps* instruction, which blends two SSE registers based on the value of a source mask. The quad-core CPU does not have hardware support for gather, so we hand-coded an optimized instruction sequence to gather from 4 non-contiguous memory locations into a SSE register. Our sequence contains 13 instructions, in comparison to 20 instructions in compiler generated code.

Figure 5 shows speed-up results for all six datasets measured on the quad-core CPU over single core baseline. For each dataset we show two bars. The left bar shows the speed-up achieved by our SSE4-optimized full-volume implementation (speed-up of 4.6x-5.9x). The right bar shows the speed-up for our SSE4-optimized sub-volume implementation (speed-up of 5.3x-6.61x). Full-volume gets almost no benefit from SSE, mainly due to the overhead of gathers. Quad-core CPU super-linear speed-up over single core baseline comes primarily from hyper-threading as well as improved micro-architecture and memory sub-system of Intel Nehalem compared to the Intel Core2 microarchitecture used for the single core baseline.

Sub-volume achieves a 1.1x-1.2x improvement over full-volume. For ds1, ds3 and ds5 the improvement mainly results from improved cache locality (Figure 4). For ds2, ds4 and ds6 the improvement is due to gradient precomputation, which eliminates large fraction of gathers and therefore improves SIMD efficiency.

6.2 Intel Larrabee

Since Larrabee is an x86-based cache-coherent many-core architecture, the parallelization mechanism is similar to quad-core CPU. However, a single sub-volume is rendered by 4 hardware threads, where each thread traces a subset of rays. This avoids cache thrashing, which could occur if each thread processed its own sub-volume.

Larrabee also features a 16-wide SIMD. To exploit SIMD, we have coded both algorithms using Larrabee SIMD vector instructions. In addition to hardware masks we take advantage of pack/unpack instructions to re-pack the rays across multiple packets, when some of the rays terminate earlier. This improves SIMD efficiency and is similar to on-the-fly ray re-ordering [3], except that it is done using hardware support available on Larrabee. Gathers are mapped to Larrabee's special gather-scatter unit. We have not used texture sampler hardware on Larrabee to sample along the ray, because our implementation uses nearest neighbor interpolation, and therefore cannot benefit from interpolation hardware.

Figure 6 shows Larrabee's speed-up over the single core baseline implementation. Here we ignore the overhead of data transfers. The left bar for each dataset shows full-volume, while the right bar shows sub-volume. Each bar is further broken into three sub-bars. Lower, middle and top sub-bars correspond to 16-, 32- and 64-core Larrabee configurations, respectively, each simulated at a nominal 1 GHz clock rate. Each sub-bar shows an incremental improvement over the previous sub-bar. For example, for ds1, sub-volume achieves speed-ups of 10.7x, 19.7x and 35.2x on 16-, 32- and 64- core configurations, respectively. We also observe that a single CPU core requires from 2x-3.5x more clock cycles than a single Larrabee core. This shows the effectiveness of the Larrabee instructions and wide SIMD. In particular, this is due to the fact that the Larrabee hardware gather support significantly improves SIMD performance.

Sub-volume outperforms full-volume because of smaller working sets and better cache locality. We see that sub-volume benefits all datasets except for ds2 and ds4. For these two datasets, the majority of the execution time is spent in gradient estimation, which already has very good cache locality. We also observe that the code scales nearly linearly with the number of cores. Large on-die caches and high memory bandwidth enable Larrabee to achieve high scalability even for large number of cores.

6.2.1 Reducing Transfer Overhead With Compression

The lower sub-bars of Figure 7 account for the data transfer overhead that is omitted from Figure 6. Here we only discuss sub-volume implementation. Left, middle and right sub-bars correspond to 16-, 32- and 64-core Larrabee configurations, respectively. Comparing to Figure 6, we observe that transfer overhead is very small on 16-core Larrabee and can be mostly overlapped with computation. On average, 16-core configuration achieves speed-up of 13.5x without data transfer and 10.2x with data transfer. Data transfer overhead significantly degrades performance of 32-core and 64-core configurations. For example, ds5 achieves almost 17x speed-up on 32 cores when transfer overhead is ignored. However, with transfer overhead is included, the speed-up is reduced to 6.1x – 3-fold performance loss.

We now show how the lossless compression scheme described in Section 3.3 can reduce the transfer overhead. We evaluate performance of our compression algorithm against ZLIB [8], which is a commonly used lossless compression technique. Other competing schemes like lossless variants of JPEG and wavelet compression [31] have a runtime complexity of O(NlogN) for N data points, resulting in larger runtimes as compared to ZLIB. Table 6 summarizes average compression ratio and decompression time. For raw volume datasets, our scheme

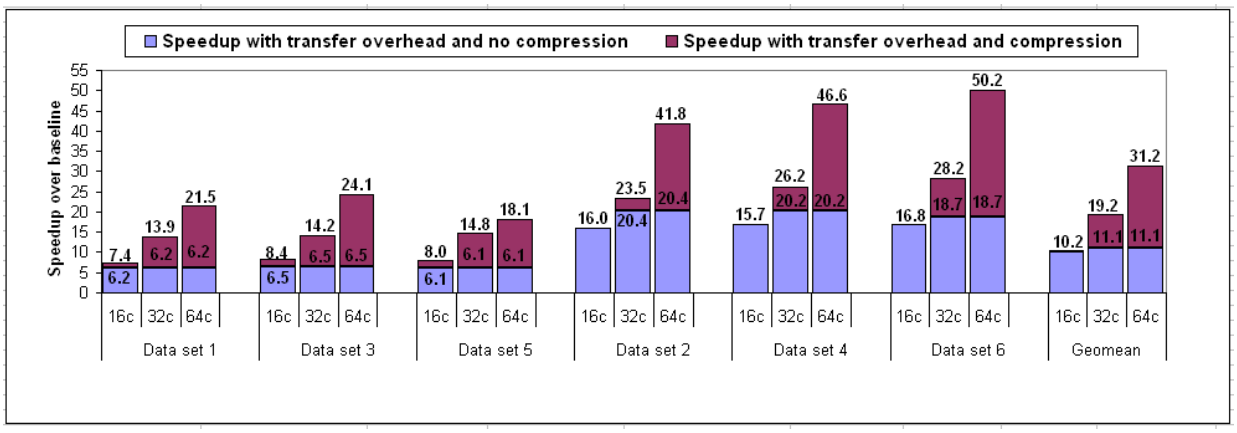


Fig. 7: **Overall performance on Larrabee**. Data transfer overhead is small on 16 cores, but significantly reduces speedup on 32 and 64-cores. Using domain specific compression with fast SIMD-friendly decompression hides at least 60% of this overhead.

Table 6: **Summary of compression results.** Compression ratio, compression/decompression times and SIMD scalability are shown for ZLIB and our compression algorithm.

Row			ZLIB	Proposed
1	Compression Ratio	volume	1.9	2.7
2	(original/compressed)	object	151	29
3	Compression time	volume	364	68
4	(cycles/element)	object	37	23
5	Decompression time	volume	40	34
6	(cycles/element)	object	7	3
7	Decompression SIMD	volume	negligible	6x
8	Scalability	object	negligible	14x

achieves 37% to 44% higher compression ratio than ZLIB and 11% to 24% faster decompression time (row 1 in the table). For the object datasets, ZLIB achieves better compression ratio than our algorithm. However, because raw data is two times larger than object data, overall our scheme achieves up to 1.3x more compression than ZLIB.

In addition to higher compression ratio, our scheme achieves faster decompression times than ZLIB. This is shown in the row 5 and 6. When integrated with our ray-casting, our approach shows 20% decompression overhead, while ZLIB shows 27%. Most importantly, our schemes is significantly more SIMD-friendly than ZLIB. While ZLIB is inherently sequential, our fixed-length decompression scheme maps straightforwardly to wide SIMD architectures. Specifically, as shown in row 7 and 8, we achieve 6x speed-up due to SIMD on raw data and 14x on object data. Finally, our scheme has faster compression times than ZLIB, as shown in row 3 and 4. We see that the overhead of compression is 2x higher than the overhead of decompression for volume data and almost 8x higher for object data. However compression overhead is less important than decompression overhead since compression occurs less frequently than decompression, since raw volume data is compressed once but rendered multiple times.

Figure 7 shows the results of using compression in our sub-volume based algorithm. For each dataset there are three bars which correspond to 16-, 32- and 64-core configurations. The upper sub-bars show the Larrabee speed-up with transfer overhead and compression. We see that compression significantly reduces overhead of data transfer on 32 and 64 cores, while adding a small computational overhead for decompression. Namely, for 32 cores we observe on average only 20%-30% slowdown, compared to the data in Figure 6. For example, for dataset 4 the right bar of Figure 6 shows that on 32 cores sub-volume achieves 32x speed-up when transfer overhead is ignored. However, as seen in Figure 7, after compression, the speed-up is 26x for the same dataset, including both transfer time as well as overhead of decompression. For 64 cores, for some datasets, we cannot completely recover performance loss due to transfer overhead. But even in the worst case, exemplified by ds5, compression hides at least 60% of this overhead.

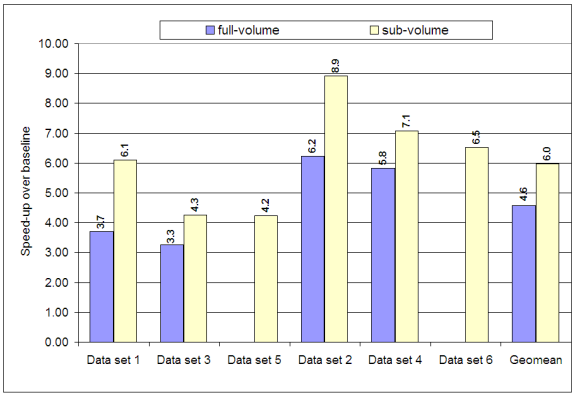


Fig. 8: **Performance on GTX280/CUDA Architecture**. Sub-volume is up to 1.6x faster than full-volume, because its smaller working set fits better in texture cache.

6.3 Nvidia GTX280

We used the CUDA SDK [22] to map volume rendering to the Nvidia GTX280, assigning a packet of rays to each thread block. While coding a CUDA kernel is relatively simple, most of our time was spent optimizing its performance. The key optimizations involved mapping key data structures, such as volume data, object map, and composite table to various available on-die memory types, such as multi-ported shared memory, constant memory, and texture memory. We tried all feasible combinations of memory type and data structures. For full-volume we could only use textures for both object and volume data, because each of these data structures is too big to fit into shared memory. We found that 1D textures performed best for smaller datasets. Due to constraints on the size of 1D textures, we were forced to use 3D textures for larger datasets to be able to run full-volume on GTX280.

Aggressively allocating data to various memories on the card reduces most of global memory accesses but not all, as reported by the CUDA profiler. Some of the remaining accesses come, for example, from the rendered image, which has to be kept in global memory. It is too large to fit into shared local memory and cannot be treated as a texture because it requires write accesses.

Figure 8 shows GTX280 performance for the 6 datasets. The left bar shows performance for the full-volume implementation, while the right bar shows performance for the sub-volume implementation. Full-volume bars are missing for ds5 and ds6 because they are too big to fit into GTX280 memory. We see that sub-volume is between 1.2x to 1.6x faster than full-volume. This is due to the fact that sub-volume has better memory locality, which is likely to result in fewer misses in the texture cache. The CUDA profiler reported many more global memory stores in case of sub-volume than full-volume. Due to the fact

that each sub-volume is projected into a local image (see Section 3.2), the areas of overlap of two projections require double the amount of stores compared to full-volume. However, the impact of extra stores on performance is negligible, compared to global loads. At this level of performance, PCIe transfer time is not a bottleneck. For example, ds2 achieves 9x over single core CPU baseline. As Figure 7 shows, for this dataset to be limited by transfer overhead, its performance has to be at least 20.4x over the baseline.

In contrast, there are several GPU implementations of ray-casting using fragment shaders [9, 15]. Mensmann, et al. compared CUDA and fragment-shader ray-casters, both implemented in Voreen rendering engine (http://www.voreen.org/), and found that their CUDA code was from 18% slower to 40% faster [20]. These results are not directly comparable to our CUDA implementation, due to their using a 6-neighbor gradient, a different transfer function, and other differences. We performed a more direct comparison by importing our medical imaging datasets and transfer functions into the Voreen fragment shader, as well as by modifying it to compute 26-neighbor gradients. Table 7 gives the results in frames-per-second for the four data sets that we were able to run through Voreen. ds5 and ds6 did not fit into the graphics card's texture memory.

Table 7: **Comparison between CUDA and fragment-shader ray-casting.** Voreen was adapted as the fragment-shader implementation.

Dataset	ds1	ds3	ds2	ds4
CUDA (FPS)	19.61	2.75	8.75	1.45
Fragment-shader (FPS)	22.56	3.08	10.76	1.83

We see that the Voreen fragment shader is 1.12x to 1.26x faster than our CUDA implementation, which is comparable to the observations made by Mensmann et al. We believe this is largely due to taking advantage of hardware rasterization to perform empty space skipping [15], which is not an option for our CUDA implementation. As the result, for datasets with many empty voxels (Table 3), such as ds2 and ds4, Voreen achieves speedups of 1.23x and 1.28x, respectively, over our CUDA implementation. For datasets with few empty voxels, such as ds1 and ds3, a significant portion of the execution time is spent in gradient computation. As a result, Voreen achieves smaller speedups of 1.12x and 1.15x, respectively. Overall, we see both CUDA and fragment-shader implementations of ray-casting are comparable in performance.

7 Conclusions

This paper maps and evaluates performance of volume rendering application on three modern parallel architectures: Intel Nehalem, Nvidia GTX280 and Intel Larrabee. Overall our parallel implementation of ray-casting delivers close to 5.8x speed-up on quad-core Nehalem over an optimized scalar baseline version running on a single core Harpertown. This enables us to render a large $750x750x1000$ dataset in 2.5 seconds. In comparison, we achieve 5x to 8x speed-up on Nvidia GTX280 over the scalar baseline. In addition, we show, via detailed performance simulation, that 16-core Larrabee delivers around 10x speed-up over single core Harpertown, which is on average between 1.5x higher performance than GTX280 at half the flops.

For 32-core and 64-core Larrabee the performance is dominated by the overhead of data transfer. When the overhead is ignored, we simulate 24x and 42x speed-up on 32-core and 64-core Larrabee, respectively, over scalar baseline. When transfer time is accounted for, the measured speed-up is reduced to 11x for both configurations. To this end, we have developed a lossless SIMD-friendly compression algorithm which on average compresses our dataset by more than 3x, while decompression overhead is less than 30%. With such compression, we are able to reduce the transfer time overhead to 20%-30% on average. This results in 19x and 31x speed-ups on 32-core and 64-core Larrabee, respectively.

References

[1] K. E. Augustine and et al. Optimization of spine surgery planning with 3D image templating tools. *Proceedings of SPIE*, 6918, 2008.

[2] H. Bjorkman, H. Eklof, J. Wadstrom, L. G. Andersson, R. Nyman, and A. Magnusson. Split renal function in patients with suspected renal artery stenosis: a comparison between gamma camera renography and two methods of measurement with computed tomography. *Acta Radiologica*, 47(1):107–13, 2006.

[3] S. Boulos, I. Wald, and C. Benthin. Adaptive ray packet reordering. *Symposium on Interactive Ray Tracing*, pages 131–138, 2007.

[4] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of Workshop on Volume Visualization*, pages 91–98, 1992.

[5] K. M. Das, A. A. El-Menyar, A. M. Salam, R. Singh, W. A. Dabdoob, H. A. Albinali, and J. Al Suwaidi. Contrast-enhanced 64-section coronary multidetector CT angiography versus conventional coronary angiography for stent assessment. *Radiology*, 245(2):424–32, 2007.

[6] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *SIGGRAPH*, volume 22, pages 65–74. 1988.

[7] R. A. Fisher. On the interpretation of x2 from contingency tables, and the calculation of p. *Journal of the Royal Stat. Society*, 85(1):87–94, 1922.

[8] J.-L. Gailly and M. Adler. ZLIB documentation and sources. *http://www.zlib.net/*.

[9] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski. Advanced illumination techniques for gpu volume raycasting. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–166. ACM, 2008.

[10] Y. Heng and L. Gu. Gpu-based volume rendering for medical image visualization. pages 5145–5148, 2005.

[11] K. Hohne and R. Bernstein. Shading 3D-Images from CT using gray-level gradients. *IEEE Trans Med Imag*, MI-5(1):45–47, 1986.

[12] Intel. SSE4 Programming Reference. 2007.

[13] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. In *Proc. of the 2nd IEEE/EG Symp. on Interactive Ray Tracing*, pages 11–17, 2007.

[14] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *IEEE Symp. on Interactive Ray Tracing*, pages 115–124, 2006.

[15] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *IEEE Visualization 2003*, pages 287–292, 2003.

[16] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94*, pages 451–458, New York, NY, USA, 1994. ACM.

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

[18] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[19] M. Magallón, M. Hopf, and T. Ertl. Parallel volume rendering using PC graphics hardware. In *Pacific Conference on Computer Graphics and Applications*, pages 384–389, 2001.

[20] J. Mensmann, T. Ropinski, and K. Hinrichs. Slab-Based Raycasting: Efficient Volume Rendering with CUDA, High Performance Graphics. *Poster in High Performance Graphics 2009*, August 2009.

[21] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *IEEE Vis*, pages 239–245, 1998.

[22] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.0. 2007.

[23] NVIDIA. NVIDIA Tesla C870 GPU Computing Processor Board, 2008.

[24] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro realtime raycasting system. In *SIGGRAPH*, pages 251–260, '99.

[25] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *EG/IEEE TCVG Symposium on Visualization*, pages 231–238, 2003.

[26] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, and et al. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[27] J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *Symposium on Data Visualisation 2002*, pages 95–104, 2002.

[28] I. Wald and et al. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164, 2001.

[29] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent raytracing. In *Computer Graphics Forum/Proceedings of EUROGRAPHICS 2001*, pages 153–164, 2001.

[30] L. Westover. Footprint evaluation for volume rendering. In F. Baskett, editor, *SIGGRAPH 90*, volume 24, pages 367–376, 1990.

[31] Z. Xiong, X. Wu, S. Cheng, and J. Hua. Lossy-to-lossless compression of medical volumetric data using three-dimensional integer wavelet transforms. *Medical Imaging, IEEE Transactions on*, 22(3):459–470, 2003.