

ABSTRACT

Title of Dissertation: EFFICIENT RENDERING OF
LARGE 3-D AND 4-D SCALAR FIELDS

Jusub Kim, Doctor of Philosophy, 2008

Dissertation directed by: Professor Joseph JaJa
Department of Electrical and Computer Engineering

Rendering volumetric data, as a compute/communication intensive and highly parallel application, represents the characteristics of future workloads for desktop computers. Interactively rendering volumetric data has been a challenging problem due to its high computational and communication requirements. With the consistent trend toward high resolution data, it has remained a difficult problem despite the continuous increase in processing power, because of the increasing performance gap between computation and communication. On the other hand, the new multi-core architecture trend in computational units in PC, which can be characterized by parallelism and heterogeneity, provides both opportunities and challenges. While the new on-chip parallel architectures offer opportunities for extremely high performance, widespread use of those parallel processors requires extensive changes in previous algorithms to take advantage of the new architectures.

In this dissertation, we develop new methods and techniques to support interactive rendering of large volumetric data. In particular, we present a novel method to layout data on disk for efficiently performing an out-of-core axis-aligned slicing

of large multidimensional scalar fields. We also present a new method to efficiently build an out-of-core indexing structure for n-dimensional volumetric data. Then, we describe a streaming model for efficiently implementing volume ray casting on a heterogeneous compute resource environment. We describe how we implement the model on SONY/TOSHIBA/IBM Cell Broadband Engine and on NVIDIA CUDA architecture. Our results show that our out-of-core techniques significantly reduce the communication bandwidth requirements and that our streaming model very effectively makes use of the strengths of those heterogeneous parallel compute resource environment for volume rendering. In all cases, we achieve scalability and load balancing, while hiding memory latency.

EFFICIENT RENDERING OF
LARGE 3-D and 4-D SCALAR FIELDS

by

Jusub Kim

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Joseph F. JáJá, Chair/Advisor
Professor Amitabh Varshney
Professor David Mount
Professor Manoj Franklin
Professor Ankur Srivastava

© Copyright by
Jusub Kim
2008

Acknowledgements

I would like to first thank my advisor, Professor Joseph JaJa for giving me an invaluable opportunity to work on exciting and challenging projects during my graduate study. He has always been kind and patient, and pushed me to fill in the one last detail to elevate the level of my thinking and work. I sincerely admire his integrity as a researcher and also as a person. I would like to also thank Professor Amitabh Varshney. Without his constant support for this research, I would not have had the opportunity to work on this exciting problems. My thanks also goes to Professor David Bader at Georgia Tech. He has allowed me to freely use the Cell processors at Georgia Tech and kindly invited me to the Cell workshops. The knowledge that I obtained at the workshops tremendously helped me to win the IBM Cell University challenge in 2007. I would like to also thank Dr. Stephen Lockett at NIH, who was my supervisor when I had an internship at NIH in 2006 Summer. His kindness and passion for biomedical imaging has inspired me to continue this research. I would like to also deeply thank my committee members Professor David Mount, Professor Manoj Franklin, and Professor Ankur Srivastava for agreeing to serve on my dissertation committee, sparing their invaluable time reviewing the manuscript, and providing insightful feedback.

I would like to also thank the many friends that I have associated with throughout my graduate career. I was fortunate to have many friends to discuss and share

thoughts. Xingzhi Wen has been always the best friend to discuss parallel research issues. Qin Wang has been always my dependable colleague since my graduate study began. I would like to thank Youngmin Kim in CS for his kind help with graphics research issues and also Derek Juba for his help with CUDA. Many thanks also goes to Chi Cui, Sangchul Song, Soobum Lee, Seungjong Baek, Minkyong Cho and Donghoon Park for their constant encouragement and support. I would like to also acknowledge all UMIACS staff members for their prompt help and support.

My deepest thanks to my family who have always stood by me and supported me. Words cannot express the gratitude that I owe to my mom and my wife Hyun-Jung for all the love they have given me through this journey.

And I thank God who has brought me to this blessed journey and made me strong through the harsh tests for greater things.

TABLE OF CONTENTS

List of Tables	vi
List of Figures	vii
1 Introduction	1
1.1 Contribution	5
1.2 Outline	7
2 Related Work	8
2.1 Rendering	8
2.1.1 Isosurface Rendering	9
2.1.2 Volume Rendering	13
2.1.3 n-D TO 2-D Mapping	19
2.2 Representation	20
2.2.1 Out-of-Core Algorithms	20
2.2.2 Multiresolution	22
2.2.3 Compression	23
2.3 Parallel Rendering	26
2.3.1 Algorithms for Clusters of Processors	28
2.3.2 Hardware Acceleration	31
3 Component Based Layout for Large Volumetric Data	32
3.1 Efficient Data Layout for Slicing Queries	32
3.1.1 Case I: $k=2$	35
3.1.2 Case II: $k > 2$	38
3.2 Analysis	40
3.2.1 Case I ($k = 2$)	42
3.2.2 Case II ($k > 2$)	43
3.3 Experimental Results	43
3.4 Discussion	45
3.5 Conclusions	47
4 Efficient Out-of-Core Indexing Structure	53
4.1 Information-Aware 2^n tree	56
4.1.1 Dimension Integration	56
4.1.2 Indexing Structure	60
4.2 Tree Traversal and Controllable Delayed Fetching	61
4.3 Scalable Rendering	62
4.4 Experimental Results	63
4.5 Conclusions	66

5	Volume Ray Casting on Cell Broadband Engine	71
5.1	Cell Broadband Engine Overview	73
5.2	Primary Work Decomposition and Allocation	74
5.3	Implementation of Acceleration Techniques	76
5.3.1	Streaming model for acceleration	77
5.3.2	Techniques for filling performance gap between heterogeneous cores	81
5.4	Experimental Results	86
5.5	Conclusions	89
6	Volume Ray Casting on CUDA	95
6.1	The CUDA Architecture Overview	98
6.2	CELL v.s. CUDA	101
6.3	Primary Work Decomposition and Allocation	103
6.4	Implementation of the Streaming Model	104
6.4.1	Stage 1: Work List Generation	105
6.4.2	Stage 2: Rendering	107
6.5	Experimental Results	108
6.6	Conclusions	111
7	Conclusion	113
	Bibliography	116

LIST OF TABLES

3.1	Contiguity and effectiveness of Z-order + n-D supercell scheme. . . .	41
3.2	Contiguity and effectiveness in each type of a component.	41
4.1	Query performance comparison between IA 2^n -tree and 2^n -tree for the Richtmyer-Meshkov data set. The results are the average values over various types of slicing and different isovalues.	64
4.2	Query performance comparison between IA 2^n -tree and T-BON for the Richtmyer-Meshkov and the Jet data set. The results are the average values over various types of slicing and different isovalues. . . .	66
5.1	Test Datasets. (Fuel dataset size is originally 64^3 . We enlarged it for better comparison.)	86
5.2	Effects of prefetching (in milliseconds).	88
6.1	Comparison of three different architectures.	102

LIST OF FIGURES

3.1	Data access patterns for a slicing query in a 2-D case for three different data layouts. Grey blocks correspond to the disk blocks satisfying the slicing query $y=\beta$	34
3.2	A 2-D example of the component-based layout for fast slicing at every other value. Components C1 and C2 are grouped into 1-D supercells and stored in the required lexicographical orders while C0 and C3 are grouped into 2-D supercells and stored according to the Z-order. Note that a dotted box indicates each supercell.	35
3.3	A 2-D example of the component-based layout for fast slicing at every third value. Note that the only change is the element size of each component.	39
3.4	Contribution of each type of components to total time for X, Y, and $Z=\alpha$ queries.	48
3.5	Performance comparison for loading $X=\alpha$ slices (1216×800).	48
3.6	Performance comparison for loading $Y=\alpha$ slices (2048×800).	49
3.7	Performance comparison for loading $Z=\alpha$ slices (2048×1216).	49
3.8	Performance comparison for loading $X=\alpha$ slices at half resolution (608×400).	50
3.9	Performance comparison for loading $Y=\alpha$ slices at half resolution (1024×400).	50
3.10	Performance comparison for loading $Z=\alpha$ slices at half resolution (1024×608).	51
3.11	Sample slice images of the test volumetric data at X, Y, and $Z=\alpha$. (1216×800 , 2048×800 , and 2048×1216 from top to bottom.)	52
4.1	Entropy estimation in each dimension. Note that the y dimension has almost zero entropy in this example.	57
4.2	Different supercell sizes and corresponding hierarchical indexing structures for the data of Figure 4.1: (a) standard supercell; (b) information-aware supercell.	58

4.3	A sampled subvolume along time steps. Entropy computation of x dimension is performed on the line l_{yz} and averaged over all (y,z) values, while the one of t (time) dimension on l_{xyz} and averaged over all (x,y,z) values.	59
4.4	Controllable delayed data fetching. Disk accesses for the active supercells in the subvolume corresponding to the grey node are delayed until the traversal revisits the grey node.	61
4.5	Spatio-temporal entropy ratios computed at uniformly selected 100 reference time steps among the 2000 time steps in the Five Jets data. Each dashed box corresponds to a time region.	65
4.6	Isosurface of the Richtmyer-Meshkov instability data set ($1024 \times 1024 \times 960$) rendered at isovalue=70 and T=139.	67
4.7	Isosurface of the Richtmyer-Meshkov instability data set ($1024 \times 1024 \times 960$, T=100-139) cut by isovalue=70 and Y=300.	68
4.8	Zoomed image of Fig. 4.7.	69
4.9	Isosurface of the Richtmyer-Meshkov instability data set ($1024 \times 1024 \times 960$, T=100-139) cut by isovalue=70 and Z=500. (Time axis is orthogonal to the paper.)	69
4.10	Zoomed image of Fig. 4.9.	70
5.1	Cell Broadband Engine Overview [IBM06].	75
5.2	Work decomposition and assignment to the SPEs.	76
5.3	Our streaming model for acceleration techniques.	78
5.4	Main algorithms in PPE and SPE.	79
5.5	Approximation technique.	83
5.6	The case of missing non-empty subvolumes. In the figure, the shaded region is not checked by any of rays using the approximation technique.	84
5.7	Proof of proposition 1.	84
5.8	Rendered images from four datasets throughout the tests.	90
5.9	Processing time in PPE and SPE for three different combinations of approximation and refining techniques.	91

5.10	Load balance among eight SPEs.	92
5.11	Speed up with respect to the number of SPEs.	92
5.12	Performance with respect to the volume size.	93
5.13	Performance with respect to the screen size.	93
5.14	Performance comparison with Intel Xeon dual processor 3GHz with SSE2.	94
6.1	CUDA Hardware Architecture [NVI07].	99
6.2	CUDA Programming Model [NVI07].	100
6.3	Work decomposition and assignment on CUDA. A tile consists of x by y block of threads and is dispatched into one of the multiprocessors.	103
6.4	Approximation technique on CUDA.	106
6.5	The streaming model for CUDA. Note that the streaming unit is a set of tiles compared to one tile in the case of Cell processor.	107
6.6	Load balance between CPU and CUDA.	108
6.7	Performance with respect to the volume size.	109
6.8	Performance with respect to the screen size.	109
6.9	Performance comparison (CPU v.s. Cell v.s. CUDA).	110

Chapter 1

Introduction

Rendering volumetric data visually is one of the best ways to explore volumetric data [HJ05], and it is critical to be able to do it “interactively”. The rendering process is essentially a mapping from an n -dimensional data set to a 2-dimensional screen, which entails dealing with occlusion problems because n is typically higher than 2. To investigate the occluded parts of the data, we need to perform some operations such as rotating the volumetric data, zooming in and out, and changing some transparency parameters. We should be able to perform such operations “interactively” for two reasons. First, the number of possible combinations of viewing conditions is too large, thus we can not generate all the images off-line. Second, even if such an approach is possible, it is not the best way for humans to explore data sets because the sequential video automatically generated by some rules can never match humans’ intelligent way of investigation.

However, interactively rendering volumetric data is a very challenging problem due to its high demands on computation and communication requirements. For example, rendering a volumetric data of 1024^3 size at 30 frames per second will require 30 GBytes/sec memory bandwidth and about 3 Teraflops, assuming that roughly 100 instructions are needed per voxel for projection. Compared with these

requirements, the latest desktop computer using Intel Core 2 Duo processor is rated at around 100MBytes/sec maximum disk transfer rate and 8GBytes/sec memory bandwidth with around 25 GFLOPS computing power at peak performance, which obviously falls far short of the required performance.

On the other hand, there is a consistent trend in science, engineering, and medicine toward increasingly generating higher resolution data as computing power steadily increases and sensor and imaging instruments get more refined. High resolution data sets are often generated and stored as volumetric data of 3-D, 4-D or even higher dimension. For example, scientists and engineers often study physical phenomena by simulating their mathematical models on supercomputers, thereby generating time-varying volumetric data sets of sizes ranging from hundreds of gigabytes to tens of terabytes. Biomedical equipments such as CT, MRI, and 3-D confocal microscopy are now capable of providing very high resolution 3-D or 4-D volumetric data. Higher resolution data facilitate the understanding and analysis of complex phenomena that scientists, engineers, or medical practitioners are interested in investigating, and can lead to new important scientific discoveries which were not possible before.

However, this trend makes it harder to achieve interactive rendering. The reason is that the computational power and memory bandwidth are evolving at different rates. Although the number of transistors on chip has been doubling every 18 months, memory bandwidth has not increased at this rate and the gap is in fact widening. Therefore, in the situation that the data size generated increases according to computational power increase rate, computer systems would not be able to

match the performance required for rendering the data because rendering volumetric data requires high performance combining computational power and memory bandwidth. Thus, it has become critical to develop techniques for reducing memory bandwidth requirements in order to enable rendering of larger volumetric data.

On the other hand, current trends in PC architectures can be characterized by parallelism and heterogeneity. First, multi-core CPUs have replaced single core CPUs. The Intel Pentium 4 line has been replaced by Core 2 Duo and Core 2 Quad processors. Intel plans to ship even 80-core chips, which can perform a trillion instructions per second in peak performance. These multi-core CPUs may provide task-parallelism that is very difficult to get from GPUs.

Second, the programmable GPUs are rapidly evolving. For example, the latest nVIDIA 8800 GTX is capable of around 345 GFLOPS peak performance and 86.4 GBytes/sec maximum bandwidth to graphics memory, which is about an order of magnitude superior to the latest CPUs. With the computation power doubling every year, they now offer a much more flexible memory and programming model, making it an increasingly attractive choice for running general purpose compute-intensive applications.

The heterogeneous and parallel computing architecture began to appear on gaming processors. For example, Cell processors, which are shipped with Sony's new PlayStation 3 game console, are composed of one 'Power Processor Element' (PPE) which is a conventional CPU (a variant of IBM PowerPC) and multiple 'Synergistic Processing Elements' (SPE) that are SIMD-type simple cores. The PPE and SPEs are linked together by an internal high speed bus having a theoretical

peak bandwidth of 204.8 GBytes/sec. Microsoft's XBox 360 uses an unified memory architecture in which a CPU and GPU share the memory providing ~ 20 GBytes/sec memory bandwidth. Both architectures can achieve very high peak computational performance (~ 200 GFLOPS). We will see more and more heterogeneous parallel compute resources combined together, each exploiting a different type of parallelism.

Programming the heterogeneous parallel PC architecture in which different types of many-cores are working together provides significant technical challenges. For several decades, we have been enjoying performance increase simply by scaling frequency. Thus, without any algorithmic change in our applications, we could get performance increases simply by upgrading to newer CPUs. However, as chip makers decide to go toward parallel architectures, it has become almost impossible to increase sequential algorithm performance. Now almost every algorithm has to be changed to make use of the parallel architecture, which requires fundamental algorithmic changes.

Moreover, programmers now have to deal with heterogeneous cores. Conventional CPUs are designed to perform well in a type of program where each data may be processed differently, and thus the memory system is optimized to provide low latency to each operation with the help of multi-layered caches. On the other hand, GPUs are designed to achieve the maximum performance in data-parallel processing, in which homogeneous computation is applied to each element of the data, and the memory systems are more optimized to provide high throughput rather than low latency. Designing new parallel algorithms that make the best use of the heterogeneous parallel architectures is a challenging problem although it provides

new opportunities for applications such as volume rendering that demands high computational power.

1.1 Contribution

This dissertation makes several contributions to the areas of out-of-core data management and parallel graphics.

- Out-of-Core Data Management
 1. We provide a new efficient data layout scheme for an out-of-core axis-aligned slicing of large multidimensional scalar fields. We show that the typical Z-order or Hilbert-Peano order is not optimal for the axis-aligned slicing queries and introduce a novel *component-based layout* scheme providing more efficiency in a specialized problem domain, in which it is only required to provide fast slicing at every k -th value, $k > 1$. The features of our component-based data layout scheme are the following. First, it provides much faster processing time for any type of axis-aligned slicing queries at every k -th value, $k > 1$. Second, it does not duplicate data. Last, the data layout can be generalized to any high dimension.
 2. We provide a new multidimensional indexing structure - Information-Aware (IA) 2^n tree - that can provide higher indexing structure efficiency than previous methods. While a typical 2^n tree recursively subdivides the n-D volume into 2^n almost equal subvolumes, our IA- 2^n tree determines

the dimensions of the subvolume based on the information embedded in the data so that subvolumes can be as coherent as possible along each dimension. Our IA- 2^n tree is particularly useful when dealing with temporal volumetric data. The benefit of our IA- 2^n tree is the reduced amount of time for both tree traversal and data reads, given the same requirement on indexing structure size. We compare our tree with the best previous indexing structure, and achieve almost an order of magnitude higher indexing structure efficiency.

- Parallel Graphics

1. We present a streaming model based efficient parallel implementation of volume ray casting on the Cell processor. In particular, we achieve an optimized implementation of two main acceleration techniques for volume ray casting [Lev90] - empty space skipping and early ray termination - on the heterogeneous multi-core processor. Our streaming model based scheme provides the following two key benefits. First, we essentially remove the overhead caused by traversing the hierarchical data structure by overlapping the empty space skipping process with the actual rendering process. Second, using prefetching, we essentially remove memory access latency, which has been the main performance degradation factor that is due to the irregular data access patterns. In addition to these two key benefits, we can also achieve better SIMD utilization in the SPEs because the SPEs know the sampling voxels to process in advance and thus they

can pack them into SIMD operations.

2. We present a streaming model based volume ray casting scheme for the nVIDIA CUDA architecture. We extend the streaming model used on the Cell processor to the PC environment where CPU and GPU are collaborating for volume ray casting. We show that our model essentially removes the tree traversal optimization overhead. Also, we compare the results against those achieved on the Cell processor and Intel Xeon processor.

1.2 Outline

We begin the dissertation by reviewing related work in Chapter 2, covering three categories: rendering, representation and parallelization. In Chapter 3, we describe our new data layout scheme for an out-of-core axis-aligned slicing of large multidimensional scalar fields. We then describe our new multidimensional indexing structure which can provide higher indexing structure efficiency than previous methods in Chapter 4. In Chapter 5, we start with the introduction of the Cell processor and describe our streaming model based parallel volume ray casting algorithms for the multi-core processor. We then describe how we apply the model to the CUDA architecture in Chapter 6. The CUDA architecture is also introduced at the beginning of the chapter. Finally, we reiterate our contributions and conclude this dissertation in chapter 7.

Chapter 2

Related Work

Rendering large 3-D/4-D volumetric data efficiently can be handled in three principal ways: the acceleration of the rendering computation, compact representation of volumetric data, and parallelization. While rendering is about how to make a meaningful image from the 3-D/4-D data, representation is concerned with reducing the amount of data that flows through the rendering pipeline so that the data set can be efficiently managed with available computing resources. Compression and multiresolution techniques are typically employed for that purpose. Due to the fast growing size of volumetric data sets, parallel rendering has also become a necessary step to achieve interactive frame rates and to implement a scalable rendering system. In this chapter, we review known major techniques to handle each of the areas mentioned above.

2.1 Rendering

In this dissertation, we are primarily interested in visualizing 3-D/4-D scalar fields, which represents the most common type of volumetric data. Visualizing fields can typically be done at low-level by voxel-based techniques, where no human knowledge is incorporated into the target of the visualization and visualization is

based only on the voxel values. However, visualizing fields can also be done at a higher level by feature-based techniques. In feature-based techniques, features based on domain knowledge are extracted, tracked and visualized [PVH⁺03].

There are generally two different approaches for visualizing 3-D discretely sampled data sets using the voxel-based techniques: isosurface rendering and volume rendering. In isosurface rendering, we typically generate an explicit geometric representation of the surface defined by a certain density value, using a polygonal mesh to approximate the surface [LC87]. The polygonal mesh is rendered using a graphics hardware. In volume rendering, every sample value is initially mapped to an opacity and a color, which is typically done using a 1-D transfer function that can be a simple ramp, a piecewise linear function, or more complicated interpolation functions. The color and opacity values are then composited in one of several ways. Ray casting [Lev90], splatting [Wes90], shear-warping [LL94] and texture mapping [WE98] are the most representative techniques for creating the final image. Volume rendering is a powerful technique that enables an understanding of spatial relationships between different structures embedded in the 3-D space.

2.1.1 Isosurface Rendering

An isosurface can be defined as the set of 3-D points satisfying $f(x,y,z) = \alpha$, for some density value α , called isovalue. Lorensen and Cline's Marching Cubes algorithm [LC87] has been the most widely used strategy for extracting the polygonal approximation of an isosurface. This algorithm uses a case table of edge intersections

to describe how a surface cuts through each cube in a 3D data set. Since the introduction of this strategy, a variety of more efficient algorithms have appeared in the literature, which either use geometric partitioning such as the octree [WG92], or use value space partitioning such as the span space [LSJ96, SHLJ96] and the interval tree [CMM⁺97], or take a hybrid approach such as the seed propagation technique [BPS96]. These algorithms use various types of indexing structures to avoid the exhaustive volume scan in the marching cubes algorithm. Sutton et al. [SHS*99] provides a detailed review and performance comparison of isosurfacing algorithms on 3-D scalar fields. Note that isosurface visualization can also be achieved by ray casting, in which no explicit geometric representation is extracted [PSL⁺98, Shi05, WFM⁺05]

In the following, we review known techniques for visualizing 4-D (time-varying 3-D) scalar fields in more detail, grouping them into two different categories depending on how they deal with the temporal dimension. In 3-D+time approach, the fourth temporal dimension is treated as a special dimension, while in the 4-D approach, time is treated just as another dimension.

3-D+Time Approach: In this approach, research has primarily focused on how to extend the existing indexing structures for static data sets to time-varying data sets while trying to reduce the size of the structure by exploiting temporal coherence, and on how to use out-of-core techniques to deal with the inherently large size of time-varying data sets.

Sutton and Hansen [SH00] present *Temporal Branch-On-Need tree (T-BON)* in which they build a *Branch-On-Need-Octree (BONO)* [WG92] at each time step,

storing general structure of the trees in a single file and achieve speed-up by using the demand-driven paging [CE97] against the in-memory version of the BONO. BONO is a space-efficient octree when the dimensions of a volume are not necessarily power of two. In BONO, each lower subdivision covers the largest power of two instead of exact half, and each node stores the minimum and maximum values of the subvolume covered by the node. As noted in [SH00], this indexing structure does not capture the temporal coherence, and thus its size increases linearly with the number of time steps. Shen [She98] proposes a *Temporal Hierarchical Index Tree* as an extension of the span space structure [SHLJ96] for time-varying fields. In this method, a binary tree is built over the entire time domain, and cells (basic cubic units in a volume) that have a small amount of variation over time are placed in the root node of the tree that covers the entire time span while cells with a larger variation are placed in multiple nodes of the tree multiple times, each for a short time span. They organize the cells at each node using the span space structure [SHLJ96]. Chiang [Chi03] proposes an out-of-core version of the hierarchical temporal tree, by incorporating an out-of-core version of the interval tree indexing structure. He builds a time hierarchy to store the metacells (clusters of cells) whose field values are close enough to each other for the corresponding time interval and used the Binary-Blocked-I/O interval tree (BBIO) [CSS98] as a secondary structure to support I/O optimal interval searches. Bajaj et al. [BSS02] extend their seed set propagation techniques [BPS96] to deal with time-varying data sets. In their method, the isosurface at time t is not extracted from scratch, but refined from the isosurface at $t - 1$ using two steps: temporal propagation of old components and seed set generation of new components.

4-D Approach: Some researchers treat the fourth temporal dimension in the same way as the other dimensions and produced some results that are not possible in the 3-D+Time approach. A 4-D isosurface can be defined as the set of points that satisfy $f(x, y, z, t) = c$, for a given isovalue c . The problem of extracting linear approximations to 4-D isosurfaces from sampling over a structured grid was addressed by Weigle and Banks [WB98] and Bhaniramka et al. [BWC04]. Weigle and Banks [WB98] describe a recursive contour meshing strategy for n-dimensional grids, involving the decomposition of the hypercubes into n-simplices, followed by contouring these simplices into (n-1)-simplices to satisfy a given constraint, and the process is repeated for additional constraints. This strategy is useful to generate an isosurface at a non-integer time step, but very computationally demanding. Bhaniramka et al. [BWC04] present an algorithm for constructing isosurfaces in any dimension. They extend the marching cubes algorithms to a n-dimensional regular grid by presenting an algorithm that can automatically generate the look-up tables for triangulation of isosurfaces in the n-dimensional regular grid. Their algorithm leads to several useful applications involving interval volume and morphing. However, the approach of extracting isosurfaces in four-dimensional space and taking only a slice defined by one or more constraints is computationally prohibitive for large scale data.

2.1.2 Volume Rendering

Volume rendering can be achieved in several ways depending on how the color and opacity values of sampled points are projected onto a 2-D screen. However, all methods can be viewed as approximations of the low-albedo volume rendering integral [Kru90]. This technique analytically computes $I_\lambda(x, r)$, the amount of light of wavelength λ that is received at location x on the image along the ray direction r :

$$I_\lambda(x, r) = \int_0^L C_\lambda(s) \mu(s) e^{-\int_0^s \mu(t) dt} ds,$$

where C_λ is the light of wavelength λ emitted or reflected at location s in the direction r , L is the length of ray r , and $\mu(s)$ is the density at the location, which is used to account for the higher reflectance of particles with larger densities. The light is attenuated by the densities between the eye and the locations according to the exponential function. Practical volume rendering algorithms use discretized versions of the equation [MHB⁺00]. In the following, we review four main volume rendering techniques.

Ray Casting: Ray casting is generally known to produce the best quality images although it is also known to be the most time-consuming technique. The idea behind ray casting is to shoot rays from the view point (eye position), one per pixel on the screen into a volume, sample scalar field values along each ray, map each sampled value into a color and an opacity value based on a transfer function, and accumulate the mapped values along each ray to generate the final pixel value on the screen.

Levoy [Lev90] proposed two optimization techniques: empty space skipping and early ray termination. He used a complete binary octree to make rays efficiently skip empty space and also made each ray traversal terminate early if the opacity value accumulates to a level where the color stabilizes. Yagel and Shi [YS93] proposed another optimization technique using frame-to-frame coherency in volume animation, where the same volumetric data set is rendered, but changing some rendering parameters or object attributes such as transfer functions, light source position and color, or a viewer's position and viewing direction. Their method saves, at each pixel on the screen, the coordinates of the first non-empty voxel encountered by the ray emitted from that pixel, so that rays can start from the coordinates in the next frame avoiding the repeated traversal of the empty space. This method is especially useful when viewing parameters are fixed because a ray can jump to the first non-empty voxel without traversing empty space, but can also be used when the model rotates. In that case, the stored coordinates are first rotated according to the model rotation and reprojected to new screen pixels. Some new rays need to be cast from empty pixels after the reprojection. This space leaping method has been improved by Wan et al. [WSK02] in several ways.

For time-varying volume, Shen and Johnson [SJ94] propose using data coherence between consecutive time steps to compress data and also to accelerate sequential visualization of each time step given fixed viewing parameters. They generate a differential file which contains information about the voxels that changed their values from the previous time step and use it by casting rays only for the pixels on the screen corresponding to the changed voxels. They achieved a significant

speed up for the volumetric data where the number of changed values is small.

Splatting: While ray casting is a backward mapping technique (also called image-order algorithm) which maps the screen plane onto the data by casting rays from screen pixels to the data, splatting [Wes90] is a forward mapping technique (also called object-order algorithm) which maps the data onto the screen plane. In this approach, the renderer has to determine the screen space contribution (footprint functions) of each sample point to the final image, the sampling of the footprint function and the spreading of the sample’s contribution. Westover [Wes90] proposed the use of a pre-computed footprint function table to build a view-transformed footprint table for a particular view. In this method, the renderer performs the reconstruction for all samples along the plane, called a sheet, most parallel to the screen plane in the object space. When all the voxels in a sheet are processed, the sheet is composited into the intermediate image. When all sheets are processed, the working image becomes the final image. Forward mapping techniques have an advantage of being less computationally demanding than ray casting.

Bajaj et al. [BPRS98] used the splatting technique for visualizing volumetric data of any dimension by projecting n-Dimensional data to a 2-D plane. To achieve efficiency, they adopt a 2^n tree hierarchical representation for multiresolution display and use 2-D texture mapping hardware to efficiently render each splat as a 2-D textured polygon.

Neophytou and Mueller [NM02] used the splatting technique for time-varying data visualization based on more efficient sampling grids. Their approach first slices a 4-D scalar field by a 4-D hyperplane making a 3-D volume and render the volume

using the splatting technique. The speed-up in rendering time was achieved by using more efficient sampling grids called the Body Centered Cartesian (BCC) grid which is a generalization of the hexagonal grid. Theoretically, the BCC grids can save almost 30% of the samples in 3-D, which directly leads to the reduction of the rendering time.

Shear Warp: Shear Warp [LL94] is a hybrid technique that combines the advantages of the image-order (e.g. ray casting) and object-order algorithms (e.g. splatting). This method first shears the volume to an intermediate coordinate system in which all viewing rays are parallel to one of the coordinate axes. Then it projects each slice of the sheared volume onto a 2-D intermediate image in front-to-back order because interpolation coefficients are the same in all slices, and finally warps the 2-D image to produce the final image. Lacroute and Levoy [LL94] propose several optimization techniques for the shear warping rendering method. In their method, the volume is rendered by marching through run-length-encoded volume and image simultaneously in scanline order, where transparent voxels and opaque pixels are efficiently skipped during this traversal. Traversing the volume in storage order is the advantage of object-order algorithms, and early ray termination coming from skipping opaque pixels on the working image is a clear advantage of image-order algorithms. Although shear-warp method is known to be the fastest software rendering method, it potentially has a disadvantage of producing aliasing as the viewing angle gets close to 45 degrees relative to the slices of the data set because it uses bilinear interpolation to re-sample the volumetric data instead of trilinear interpolation in ray casting. Another drawback is that it requires three copies of

the compressed volume to allow for front-to-back traversal in all viewing directions. These drawbacks are addressed later by Sweeney and Mueller [SM02].

For time-varying volume, Anagnostou et al. [AAW00] propose several optimization techniques. In their work, they assume that a great percentage of the volume remains unchanged over time and seek to exploit the spatial and temporal coherence. For using the temporal coherence, they exploit a common property, called partial ray compositing, which guarantees that we can always safely divide a ray into two or more parts, separately composite each part of ray and generate the final image by combining them. They divide the volume into slabs (thick slice) along the viewing direction and re-render only slabs which have changed from the previous time step. They expand the previous RLE-encoded volumetric data [LL94] and make it easy to update the RLE-encoded data from the previous time step. However, it has a drawback of consuming a large amount of memory space.

Texture Mapping: As texture mapping capability has become a part of a standard graphics hardware, researchers began using the new capability for volume rendering [CCF94, CN94, WVV94]. Texture-mapped volume rendering makes an image by projecting a set of texture-mapped polygons that span the whole volume. In 2-D texture-mapped volume rendering, the polygons are orthographic slices along the most parallel axis to the viewing direction, and in 3-D texture-mapped volume rendering, the polygons are a set of slices that are perpendicular to the viewing directions. The difference between 2-D and 3-D texture mapping lies in the sampling accuracy (e.g. bilinear vs. trilinear). Texture mapping based volume rendering typically shows more artifacts than the software-based ray casting, which

is mainly due to the fact that we can not make the sampling distance equal on all rays in perspective projection using the texture mapping technique. Nevertheless, the texture-mapping based volume rendering has enabled interactive frame rates for a moderate-sized data, which was not possible with the software-based rendering methods. The performance of texture-mapped volume rendering is limited mostly by the transfer time between main memory and texture memory, and the input must be streamlined if the entire volumetric data is larger than the texture memory size.

For time-varying rendering, Ellsworth et al. [ECS00] used the Time-Space-Partitioning (TSP) tree [SCM99] to exploit spatial and temporal coherence with the 3-D texture mapping hardware for rendering. They render regions that have high spatial coherence using untextured, flat-shaded polygons freeing the associated texture memory. Regions with high temporal coherence are shared between two or more time steps saving also texture memory. They also introduce color-based error metrics to decide whether or not to use untextured polygons or share the volume region across certain time steps. Since the scalar values of the voxels are mapped to colors using a transfer function, the color-based error metrics improve the selection of texture volumes to be loaded into texture memory.

Lum et al. [LMC02] propose compressing time-varying volume using the Discrete Cosine Transform (DCT) and rendering by 2-D texture mapping hardware. They encode sequences of scalars along the time dimension into single scalar indices by transforming each sequence into a set of DCT coefficients, which are quantized and stored in 8-bit values. They are encoded such that single index value represents a sequence of temporally changing scalar values and therefore each index is a sample

in the space of possible time varying scalar values. This indexed texture is quickly decoded through the frame to frame manipulation of a palette on a graphics hardware instead of using inverse DCT. In order to reduce the discontinuity between window sequences, they interleave the starting times of the windows for each slice of the volume.

2.1.3 n-D TO 2-D Mapping

Rendering a 3-D volume on a screen can essentially be viewed as a 3-D-to-2-D mapping problem. In addition to the typical volume rendering techniques described earlier, slicing techniques in which the user displays a 2-D slice has been widely accepted in domains such as medical applications. Similarly, rendering a time-varying volume can be viewed as a 4-D-to-2-D mapping problem and the same techniques, projection and slicing, can be used.

Most of the visualization techniques for time-varying volumetric data have taken temporal-slicing first followed by the projection approach, in which a 3-D volume at each time step is rendered using a 3-D-to-2-D projection. However, there have also been several interesting different efforts that are worth mentioning. Woodring et al. [WWS03] and Neophytou and Mueller [NM02] use hyperslicing followed by projection approach, in which not only a temporal slice but also an arbitrary 3-D slice is projected. Kim and JaJa [KJ06] and Shi and JaJa [SJ06] also use hyperslicing first, but visualized the resulting 3-D volume using isosurface rendering. On the other hand, Bajaj et al. [BPRS98] use only the projection approach.

They perform the n-Dimensional volume rendering just like an X-ray projection, which makes the ordering issue with n-D to 2-D projection irrelevant. They propose a scalable graphical interface for an easy n-Dimensional data exploration based on generalization of the splatting technique to a higher dimension. Woodring and Shen [WS03] also use only projection approach proposing a technique called *Chrono-volume*, which integrates time-varying volumetric data along time and generates a single view that captures the essence of multiple time steps. The benefit of their method is in enabling users to view the surrounding context of adjacent time steps of a region of interest in a single view.

2.2 Representation

One of the most challenging issues associated with large volume visualization is how to manage the large size of the data to enable interactive exploration and discovery. The large amounts of data not only increase storage costs, but also increase I/O costs from disk to main memory or main memory to video memory. There are simple ways of reducing data size like cropping, dimension reduction (e.g. extracting a slice), and sub-sampling. However, more general ways are to use out-of-core algorithms, multiresolution, and compression techniques.

2.2.1 Out-of-Core Algorithms

Dealing with large data very often involves accessing data directly from hard disk drive during run time because whole data can not fit in main memory. However,

due to their electromechanical components, disks have several orders of magnitude longer access time than random access main memory. Out-of-core algorithms are concerned with how to efficiently perform certain operations when data is on external memory, typically hard disk drive. A single disk access reads or writes a block of contiguous data at once. The performance of an out-of-core algorithm [Vit00] is often dominated by the number of I/O operations, each involving the reading or writing of disk blocks. Hence designing an efficient out-of-core visualization algorithm requires a careful attention to data layout and the organization of disk accesses in such a way that necessary data blocks are moved in large contiguous chunks into main memory.

A number of out-of-core techniques to handle scientific visualization problems have appeared in the literature [HJ05], as larger and larger data sets are being generated. Cox and Ellsworth [CE97] show that application-controlled paging and data loading in a unit of subcube with the ability of controlling the page size can lead to better performance in out-of-core visualization. Out-of-core isosurface extraction algorithms are reported in [SH00, Chi03, SJ06]. Out-of-core volume rendering algorithms are reported in [SCM99, LM99, FS01]. Silva et al. [SCESL02] provide a good survey on out-of-core algorithms for scientific visualization and computer graphics.

For efficient out-of-core data accesses, it is important to lay out data in a way that algorithms retrieve data in contiguous chunks. Space filling curves [Sam90] have been used for mapping n-dimensional data to one-dimension while trying to preserve the spatial locality of the original n-dimensional data. The most popular ones are the Z-order [OM84] and the Peano-Hilbert order [Hil91]. While the Peano-Hilbert order has a slightly higher degree of locality, the Z-order has been more frequently

used because of the simplicity of the conversion process between the key and its corresponding element in the multidimensional space. Lawder [Law00] examines different kinds of space filling curves to develop indexing schemes for fast retrieval of data in multi-dimensional databases.

2.2.2 Multiresolution

The most common multidimensional structure for data sampled on regular grids is the Octree [Sam90]. Time-varying volumetric data can be treated as 4-D data considering time as the fourth dimension and a 4-D octree (a straightforward extension of 3-D octree) can be built to control rendering speed and image quality [WG94]. Shen et al. [SCM99] propose a Time-Space Partitioning Tree (TSP) which can capture both the temporal and spatial coherence more efficiently than the high dimensional octrees [WG94]. The TSP tree is basically an octree in which each node links to a binary time tree which hierarchically represents the corresponding subvolume through the entire time interval. In that way, they decouple the temporal and spatial coherence making it possible to capture coherence uniquely existing in spatial or temporal dimension. Each node of the binary time tree contains information such as the mean values of the subvolume, spatial error and temporal error in the given time span. Users can supply the desired error tolerances for the trade-off between image quality and rendering speed. In a traversal, the nodes are visited in a front-to-back visibility order according to a viewing direction and the subvolumes selected by users' error tolerance are rendered individually. The TSP

tree is extended for parallel environments by Wang et al. [WGLS05].

Gregorski et al. [GSDJ04] describe a scheme for adaptively extracting multiresolution isosurfaces. They generate a multiresolution representation using tetrahedral meshes defined by longest edge bisection. At run time, they refine the tetrahedra mesh given an error tolerance and view frustum and extract isosurfaces from the tetrahedra using the vertex programming capability of modern graphics cards. A data layout also follows the access pattern indicated by mesh refinement.

On the other hand, Linsen et al. [LPD⁺02] describe *4th root of 2* subdivision techniques, in which they make every subdivision step only double the number of vertices, which gives finer intermediate resolution than typical regular subdivision.

2.2.3 Compression

Neophytou and Mueller [NM02] propose a lossless compression strategy using sampling grids called Body-Centered-Cartesian (BCC) grids, which are a generalization of hexagonal grids. The BCC grids can reduce the number of samples by almost 30% in 3-D and 50% in 4-D without loss of fidelity under the condition that the sample signal has a spherically bandlimited frequency spectrum. Although lossless compression allows the exact reproduction of the compressed data and hence has the advantage that we can remove the original data, we need to use lossy compression to achieve greater compression rates.

For lossy compression, one has to consider both the image quality at a certain compression ratio and decompression time. Compression schemes based on wavelet

transform have been widely used in many applications and enable the generation of multiresolution features as well [Wes95, GWGS02]. For time-varying data, good compression schemes also have to exploit both spatial and temporal coherence. Temporal coherence has been exploited by either using simple difference files [SJ94] or more sophisticated techniques as described next.

Guthe and Straßer [GS01] describe a compression scheme which basically follows that of MPEG, where motion prediction and compensation are applied, processing a sequence of frames consisting of I (intra-coded), B (bidirectionally predicted) and P (predicted) frames. They compressed each I frame using wavelet transforms. The coefficients of wavelet transforms are quantized to optimize the visual perception and encoded using run length encoding followed by LZH or arithmetic encoding. At run time, decompression is completely done by CPU and the decompressed volumes are transferred to video memory and visualized by the texture mapping hardware. Lum et al. [LMC01] used a Discrete Cosine Transform (DCT) to encode sequences of scalars along the temporal dimension into single scalar indices by transforming each sequence into a set of DCT coefficients, which are quantized and stored in 8-bit values. They are encoded such that single index value represents a sequence of temporally changing scalar values and therefore each index is a sample in the space of possible time-varying scalar values. This index texture is quickly decoded through the frame to frame manipulation of a palette on a graphics hardware instead of using the inverse DCT.

On the other hand, as interactive visualization of static 3-D volumetric data has become easier due to 3-D texture mapping hardware, much effort in time-varying

volumetric data compression has been put into how to pack as many time steps as possible into the texture memory and how to decompress the compressed data using GPU-based hardware acceleration.

Binotto et al. [BCF03] describe a simple compression scheme to allow simple pixel shader to decompress the data set. Their compression scheme consists of index texture and refinement texture. In index texture, they store a coarser version of a dataset, in which each texel stores either a representative color value using a given transfer function in a homogeneous region or a reference to a corresponding grid in refinement texture in a non-homogeneous region. The same refinement patterns are shared among different time instances which are packed into the texture memory at the same time. However its use is limited to the data set that presents high spatial and temporal coherence. Shneider and Westermann [SW03] also address the limited texture memory problem in the texture mapping based volume rendering. They describe a hierarchical vector quantization scheme to encode each 4^3 subvolume into one single RGB index texel, in which the red stores the mean value of the subvolume and each of the other two stores an index to each corresponding codebook which represents difference values from the mean value at two different resolutions. Using this method, they achieve about 20:1 compression ratio while maintaining reasonable texture fidelity. They show decoding and rendering can be done using programmable graphics hardware although they report a factor of 2 to 3 performance loss due to the complex shader, which is partially overcome in the case of sparse data sets by skipping the execution of the shader program for empty space. For efficient quantization of time-varying data sets, they reuse the codebook in the

closest key frame as the initial codebook to the quantization algorithm. Akiba et al. [AMC05] employ two different data reduction techniques for visualizing time-varying volume on a low-cost commodity graphics card. They first convert the raw data into a hierarchical representation using wavelet transformations [Cly03] that permit the reconstruction of the sampled data at varying power of two resolutions. Then they additionally reduce the data size by a packing technique, in which they first partition the volume uniformly into a set of equal size subvolumes and only pack the subvolumes that contain values within the user-specified range of interest into a sequence of 3-D texture blocks. The wavelet decompression is done by the CPU; however unpacking and rendering are done by a GPU with the help of additional address texture.

2.3 Parallel Rendering

There are basically two models in parallel computing: shared-memory and distributed-memory. While the shared-memory model of traditional high-performance graphics systems provides a flexible programming environment, it did not offer the required graphics scalability or a good performance/cost rate. While there were research efforts in utilizing shared-memory multiprocessors for interactively visualizing massive data sets [PSL⁺98, CD99, MP01], more efforts have recently been placed into using low-cost commodity-based PC-clusters. Note that a commodity PC graphics card, which costs only a few hundred dollars now, has a greater rendering performance than that of much more (\sim \$100,000) expensive graphics machines

years ago [HJ05]. A PC-cluster offers more scalability at a low-cost. However, it has to be carefully employed to overcome its slow interprocess communication. In the following, we review the main parallel rendering techniques for volume rendering.

Sorting Scheme: In graphics, in which rendering primitives are typically polygons, a full 3-D rendering pipeline primarily consists of geometry processing and rasterization. Depending on at which stage primitives are sorted across multiple parallel graphics pipelines, there are three different ways: sort-first (sort before the geometry processing), sort-middle (sort between geometry processing and rasterization), and sort-last (sort after rasterization) [MCEF94]. In volume rendering, in which rendering primitives are voxels, there are essentially two ways: sort-first (sort before projection) and sort-last (sort after projection).

The sort-first approach partitions the image space into N regions such that each of N processors is responsible for completely rendering one of the N regions. In contrast, the sort-last approach has each node completely render its subvolume onto a full image size after which the N full images are composited to generate a final image. The sort-first scales well with the image size making it a good solution for tiled displays because the amount of communication is a function of data size rather than image size. However, it causes a scalability problem as the size of the data set increases although it can be mitigated in the case in which a frame-to-frame coherence can be used. Also a severe load imbalance can occur when primitives are concentrated in a region. On the other hand, the sort-last scales well with data set size because the amount of communication is a function of image size. However, the compositing step can be a performance bottleneck as the resolution of

the display increases. We can also expect handling of transparency not to be easy in this case. In addition, it will need additional efforts to enable rendering on tiled displays because there is a limit on image size that one commodity graphics card can render. While there were some research efforts in the sort-first parallel rendering system [BHPB03, AR05], more research efforts in large volume visualization have been put into the sort-last parallel rendering system [WPLM01] mostly because of its scalability in terms of data size and also its good load-balancing property.

2.3.1 Algorithms for Clusters of Processors

Compositing in Sort-last: In the sort-last distributed rendering, after each node renders its allocated data, all the images have to be composited to generate the final image. There are basically two compositing methods: software-based and hardware-based. A few specialized hardware architectures have been developed for real-time image compositing such as Sepia [MHS99], Lightning-2 [SHE⁺01], and Metabuffer [BBFZ00]. They mostly attempt to circumvent the compositing cost by reading the frame buffers directly from the DVI outputs instead of using the costly frame-read-back operation. However, the increased interest in constructing a *low-cost* GPU-cluster system requires us to build a fast software-based compositing scheme. Representative software compositing algorithms involve direct send [Neu94], binary swap [MPHK94], and SLIC [SML⁺03]. In the direct send method [Neu94], each processor send pixels directly to the processor responsible for compositing them. While it is easy to implement, it could cause traffic congestion on the network. In

the binary swap [MPHK94], the volume is partitioned using k-d trees and a pair of rendered images at the same tree level exchange half the image and composite them in a recursive fashion going up the tree. SLIC [SML⁺03] attempts to reduce the amount of transmitted pixels based on the observation that only overlapped pixels need to be transmitted for compositing. They send non-overlapped spans of pixels directly to a host node and assign only each overlapped span to a node in an interleaving fashion. They report that the compositing cost stays almost constant as more processors are used and outperforms the previous two methods when high resolution image (over 500x500) is desired.

Time-varying Data: Given a parallel machine with N processors, there are basically two different strategies to render time-varying volumetric data: intra-volume and inter-volume parallelism [MC00, HJ05]. Intra-volume parallelism is the most widely used approach, in which a volume at each time step is partitioned and distributed among the N processors. Time-varying data is sequentially rendered in a batch mode fashion. In contrast, inter-volume parallelism suggests that each processor processes subvolumes from different time steps simultaneously. In a hybrid approach, the N processors can be grouped into multiple groups and each group is responsible for a single time step exploiting the intra-volume parallelism. Each group forms a pipeline and the concurrent multiple pipelines exploit the inter-volume parallelism. In this way, the expensive I/O time between time steps in a batch mode can be hidden leading to improved performance.

Other few research efforts using only intra-volume parallelism for time-varying volumetric data rendering are worth mentioning. Lum et al. [LMC02] employ a

cluster to make their Discrete Cosine Transform (DCT) based compression scheme [LMC01] scalable with large data sets. They divide each volume into N equal slabs, where N is the number of rendering nodes. At run time, each subvolume loaded into a texture memory is decompressed using simple palette manipulation and rendered using the texture mapping hardware. Wang et al. [WGLS05] extend the Time-Space Partitioning tree (TSP) [SCM99] to a cluster environment with software ray casting for time-varying volume rendering.

Multiresolution: As multiresolution techniques become common in large data visualization, several research efforts have tried to port the existing multiresolution schemes into a cluster environment. Wang et al. [WGS04, WGLS05] extends the Time-Space-Partitioning tree (TSP) [SCM99] for a sort-last cluster environment. They compress each node of the TSP tree using a wavelet transform and describe an algorithm to partition the tree and distribute the data among processors. They distribute the data in such a way that data blocks at similar resolution levels and error ranges are shared among different processors, using the hilbert space-filling curve and error-guided bucketization respectively. Although they achieve a good load balancing, the rendering time suffers from the expensive reconstruction time. Strengert et al. [SMW⁺04, SMW⁺05] attempt to employ the hierarchical wavelet compression technique [GWGS02] also in a sort-last cluster environment. They address the discontinuity issues that happen when subvolumes at different quality levels are individually rendered by different processors. They report that the performance in time-varying data is limited by the decompression time because their caching strategy for decompressed blocks only works for static data, suggesting to

exploit the coherence between time steps.

2.3.2 Hardware Acceleration

On the other hand, innovative use of recent programmable advanced graphics cards is even further improving the performance of parallel rendering systems. Kniss et al. [KMM⁺01] distribute a data set into multiple graphics cards, having each node render a subvolume using the texture mapping volume rendering technique and finally compositing them later, in a shared-memory system. Lum et al. [LMC02] and Strengert et al. [SMW⁺04] use the same technique in a cluster environment. As graphics processors become more programmable, there have been efforts to implement ray casting on the graphics cards. Stegmaier and et al. [SSKE05] show that ray casting can be implemented on the programmable graphics processor, and Müller et al. [MSE06] extend the hardware accelerated ray casting technique to a cluster system. Their efforts focus on achieving a good load balance and optimization in a sort-first cluster environment. They use a kd-tree to assign bricks to processors with adaptive load balancing technique and use a seamless bricking technique to ensure no artifacts will occur between bricks. On the other hand, it is worth mentioning some recent hardware-based ray tracing techniques for geometric rendering since the principle of ray shooting is the same. Horn et al. [HSHH07] show k-D tree based GPU raytracing methods and Benthin et al. [BWSF06] introduce ray tracing techniques for Cell Broadband Engine, based on software cache and hyper-threading.

Chapter 3

Component Based Layout for Large

Volumetric Data

Efficient visualization of large 3-D/4-D data sets on current PC architectures requires efficient strategies to deal with the storage hierarchy. First, efficient data transfer from disk to main memory is important because the data set often can not fit into main memory. Second, efficient data transfer from main memory to processing units such as multi-core CPUs or GPUs is essential to achieving interactivity. In this chapter and the next chapter, we address the initial stage, efficient out-of-core data access, while we deal in later chapters with the second stage. We address efficient data layout for out-of-core axis-aligned slicing problems in this chapter.

3.1 Efficient Data Layout for Slicing Queries

In n-D volumetric data, we define axis-aligned slicing as the process of obtaining a (n-1)-D slice by taking the sample points on the n-D plane $I_j = \alpha$, where (I_1, I_2, \dots, I_n) are the dimensions and α is one of the grid values. For example, in the case of the time-varying 3-D volumetric data, typical sequential rendering of each time step is a slicing along the temporal dimension. In addition to the popular

temporal dimension slicing, we consider all the axis-aligned hyper-slicings. In fact, the non-temporal slicing can enable more effectively to observe patterns and trends along the temporal dimension.

The widely used space filling curves such as Z-order and Peano-Hilbert order [Sam90] store neighboring multidimensional data as close as possible in storage, and hence they have been widely used because they provide good cache performance for accessing n-dimensional data. While they are effective in the situation where data access occurs across all n dimensions, more efficient data layouts are needed for slicing queries because data access occurs across only (n-1) dimensions for each such query.

For illustration purposes, Figure 3.1 shows how three different layouts affect the disk I/O efficiency for a slicing query in a 2-D case. Disk I/O efficiency can be expressed by how many contiguous disk pages are accessed for a given query. As shown in the figure, the particular lexicographical-order sequentially stores the one-dimensional slice corresponding to $y = \beta$ and thus achieves the highest contiguity in disk access for this particular slicing query while the other space filling curve schemes achieve very little contiguity.

However, employing the straightforward lexicographical-order results in worse performance for the least priority slicing query ($x=\alpha$ in the example) although it achieves better performance for the other (n-1) types of slicing queries for the n-dimensional data. The most naive approach of eliminating the single worst case would be to create an additional copy of the data and store that copy in the lexicographical-order in favor of the least priority slicing. However, it is not practical

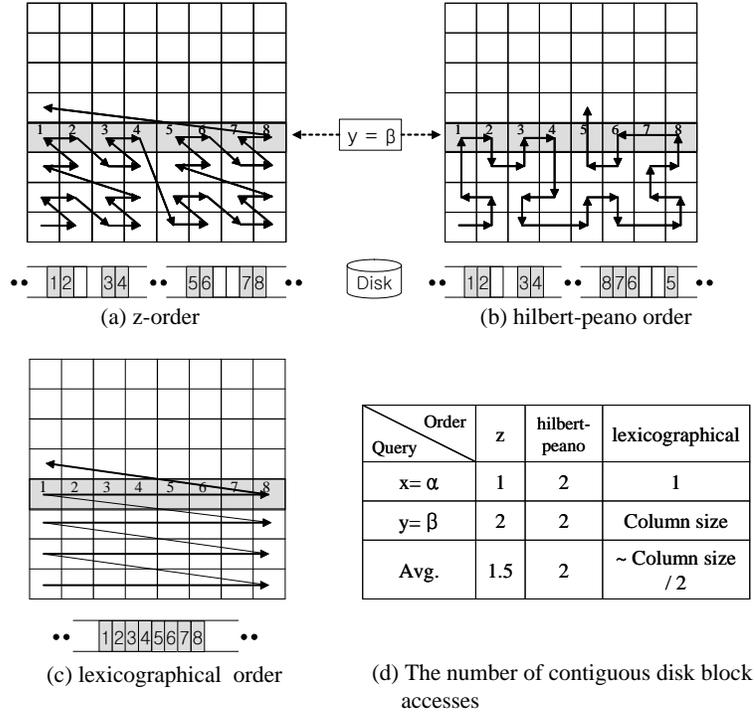


Figure 3.1: Data access patterns for a slicing query in a 2-D case for three different data layouts. Grey blocks correspond to the disk blocks satisfying the slicing query $y = \beta$.

to duplicate the already very large data set.

In the rest of this chapter, we introduce our *component-based layout* scheme to address this problem in a specialized problem domain, in which it is only required to provide fast slicing along an arbitrary dimension at every k -th value, $k > 1$. The idea is to divide the n -dimensional data set into non-overlapping components and to systematically provide a different layout scheme tailored to each component in such a way that isolated disk accesses are eliminated.

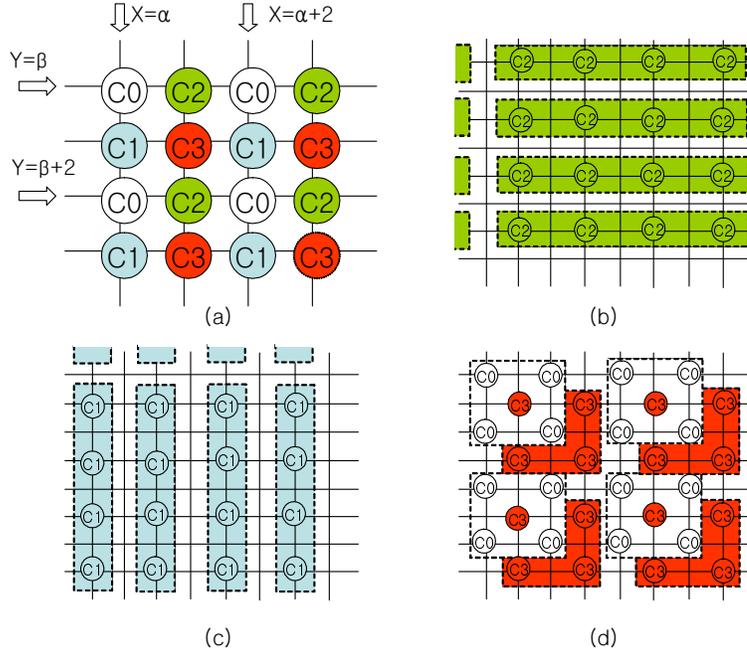


Figure 3.2: A 2-D example of the component-based layout for fast slicing at every other value. Components C1 and C2 are grouped into 1-D supercells and stored in the required lexicographical orders while C0 and C3 are grouped into 2-D supercells and stored according to the Z-order. Note that a dotted box indicates each supercell.

3.1.1 Case I: $k=2$

We first explain our component-based layout scheme for $k=2$, meaning that it is required to provide fast out-of-core slicing only at every other value. Figure 3.2 shows the layout scheme for two-dimensional data for illustration purposes. We divide the 2-D grid into four non-overlapping components, and store each component in the following way. First, we group the elements in component C1 into 1-D supercells, which are blocks of disk page size, and store the corresponding supercells in a lexicographical order that favors the X-slicing since C1 is required only by the

X-slicing. Similarly, we group the elements in component C2 into 1-D supercells and store them in a lexicographical order that favors the Y-slicing. Second, we group the elements in components C0 and C3 into 2-D supercells and store the supercells according to the Z-order since they are either required by both slicing types or not required by any type of slicing. Now given a slicing query at every other value as shown in the figure, half of the data that we access are always stored in the lexicographical order in favor of that particular slicing, providing maximum contiguous data access, while half of them are stored in Z-order.

Now, we generalize the idea to n-dimensional data. Given a n-dimensional regular grid, let (i_1, i_2, \dots, i_n) denote the index of a grid point. Then we define Component-Code (C-CODE) of the index, $C\text{-CODE}(i_1, i_2, \dots, i_n)$, as a concatenation of $(i_j \bmod 2)$, $j=1,2,\dots,n$. Then, we define each component C_i of the n-dimensional regular grid as follows.

$$C_i \equiv \{(i_1, i_2, \dots, i_n) \mid C\text{-CODE}(i_1, i_2, \dots, i_n)=i\}$$

$$Grid_n \equiv \{C_i \mid i = 0, 1, \dots, 2^n - 1\}$$

For example, the C-CODE of the 3-D grid point at (3,2,5) is 5 (=101₂) and thus belongs to component C_5 .

Now we define a slicing query as the query to generate the sample points residing on the hyperplane $I_j = \alpha$, where $\alpha \bmod 2 = 0$ (because $k = 2$). Then, a set of necessary components, A_j , for answering the slicing query $I_j = \alpha$ is

$$A_j = \{C_i \mid \text{the } j\text{-th most significant bit of } i=0\}$$

because only the components for which $i_j \bmod 2 = 0$ can be sliced by the plane and there are a total of 2^{n-1} such components. For example, given a component C_i whose C-CODE is 010_2 in a 3-D grid, we know that it is required for both X- and Z-slicing.

The 2^n components comprising an n-dimensional grid consists of 4 types of components. Let p denote the number of slicing types that a component may be subjected to (note that it is the same as the number of '0's in the C-CODE of the component). Then each component belongs to one of the following types.

1. Type I ($p=0$): A component that is not required by any type of slicing. There is only one such component such that all the bit values of its C-CODE are equal to '1'.
2. Type II ($p=1$): A component that is exclusively required by a particular slicing. There are n such components, each of which has a C-CODE having only one bit equal to '0'.
3. Type III ($2 \leq p \leq n - 1$): A component that is commonly required by p different types of slicing. Given p , there are ${}_n C_p$ such components.
4. Type IV ($p=n$): A component that is required by all slicing types. There is only one such component such that all the bit values of its C-CODE are equal to '0'.

For out-of-core access, we store each type of components in the following way.

- Type I and IV ($p=0$ or n): The elements of each component are grouped into n -dimensional supercells, which are then stored according to the Z-order.
- Type II ($p=1$): The elements of each component are grouped into $(n-1)$ -dimensional supercells, which are then stored in a lexicographical order in a way that the exclusive slicing type gets the highest priority.
- Type III ($2 \leq p \leq n - 1$): The elements of each component are grouped into n -dimensional supercells, which are then stored in a lexicographical order in a way that all the p types of slicing get higher priority than the remaining $(n-p)$ types.

Note that for type II, we group elements into $(n-1)$ -dimensional supercells because it is exclusively sliced by a particular slicing type. For type III, we always avoid the worst case, in which any of the p types of slicing gets the least priority in the lexicographical order, since there is always at least one slicing axis that does not require the component and the least priority can be assigned to that dimension.

3.1.2 Case II: $k > 2$

Now, we consider the general case where it is required to provide fast slicing at every k -th value, $k > 2$. Figure 3.3 shows a 2-D example in the case of $k=3$. Note that the only change is the element size of each component, where the element of each component is the maximal group of contiguous grid points which belong to the same component.

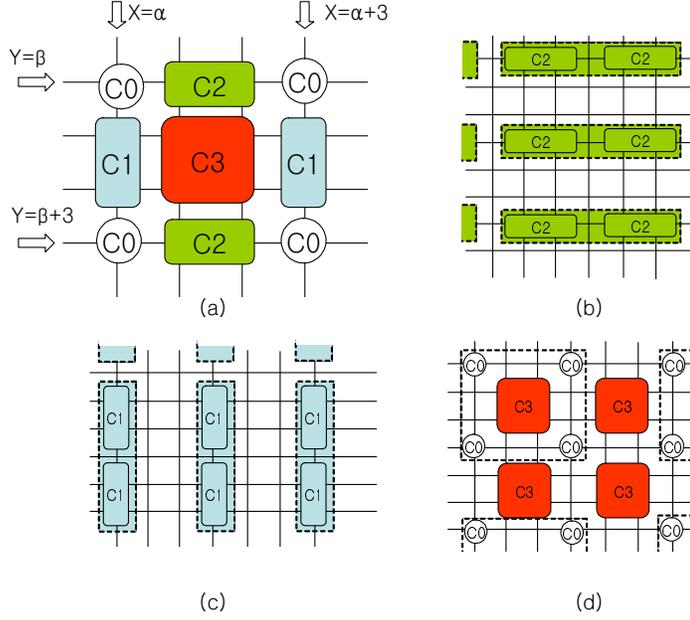


Figure 3.3: A 2-D example of the component-based layout for fast slicing at every third value. Note that the only change is the element size of each component.

We redefine a slicing query as retrieving the sample points on the plane $I_j = \alpha$, where $\alpha \bmod k = 0$. Then, we only need to generalize the previous Component-Code (C-CODE) definition as follows.

$$\text{C-CODE}(i_1, i_2, \dots, i_n) \equiv \text{a concatenation of } b_j, j=1,2,\dots,n.$$

$$b_j = \begin{cases} 0, & \text{if } (i_j \bmod k) = 0. \\ 1, & \text{if } (i_j \bmod k) \neq 0. \end{cases}$$

Now, the element size of each component increases by a factor of $(k-1)$ per a bit value '1' of the C-CODE because $(k-1)$ -times more grid points get included due to $(i_j \bmod k) \neq 0$. Hence, the size increases by a factor of $(k-1)^{n-p}$, where p is the number of '0's and thus $n-p$ is the number of '1's in the C-CODE.

Since the schemes in the case of $k=2$ for n -dimensional data depend only on

the C-CODE of a component, they apply to the general case of $k > 2$ in the same way.

3.2 Analysis

To analyze and compare the performance of our scheme to other schemes, we define *Contiguity* as the ratio of the average number of disk blocks that can be accessed sequentially to the total number of disk blocks needed for a particular slicing, and *Effectiveness* as the ratio of the amount of data needed to the amount of data transferred. Both indices are equally important in terms of disk I/O cost. In fact, disk access time can be approximated by the the time to read the necessary data at maximum transfer rate $\times \frac{1}{\textit{Effectiveness}}$, plus the time for disk head movement $\times \frac{1}{\textit{Contiguity}}$. Hence, using the two indices, we can compare the component-based layout scheme with the typical Z-order scheme in which the data is first decomposed into n-dimensional supercells each of whose size is equal to the disk page and then stored by Z-order.

Let CO_z and EF_z denote the contiguity and the effectiveness of the Z-order scheme while CO_c and EF_c correspond to the component-based data layout scheme. Assuming that an n-D supercell is of size $\underbrace{L \times L \times \dots \times L}_n$ (and hence the size of a disk block is L^n) and the n-D volume consists of $\underbrace{M \times M \times \dots \times M}_n$ supercells ($M \gg 2^{n-1}$), a slice of the supercell is of size L^{n-1} and thus the effectiveness of the n-D supercell is always $\frac{1}{L}$ ($= \frac{L^{n-1}}{L^n}$).

The number of sequentially accessed blocks in Z-order is 1,2,4,...,or 2^{n-1} ac-

ording to the slicing axis. Table 3.1 shows the contiguity and the effectiveness of the Z-order combined with the n-D supercell scheme.

	Contiguity (CO_z)	Effectiveness (EF_z)
Z-order	$\mathcal{O}\left(\frac{2^{n-1}}{M^{n-1}}\right)$	$\frac{1}{L}$

Table 3.1: Contiguity and effectiveness of Z-order + n-D supercell scheme.

On the other hand, in a lexicographical order in favor of a certain slicing priority, the contiguity becomes $1, \frac{1}{M}, \frac{1}{M^2}, \dots$ in a decreasing order of the priority of the slicing. Table 3.2 shows the contiguity and the effectiveness in each type of a component in the component-based layout, assuming that each component is of full volume size. Note that type I components are never required and that type II components have no discontinuous disk accesses and do not load any redundant data.

	Contiguity (CO_c)	Effectiveness (EF_c)
Type II	1	1
Type III	$\Omega\left(\frac{1}{M^{n-2}}\right)$	EF_z
Type IV	CO_z	EF_z

Table 3.2: Contiguity and effectiveness in each type of a component.

Since CO_z is upper bounded by $\left(\frac{2}{M}\right)^{n-1}$ and CO_c of type III is lower bounded by $\frac{1}{M^{n-2}}$, the value of CO_c for type III components is at least $\frac{M}{2^{n-1}}$ times as high as the CO_z . Thus, type III components always have better contiguity than type IV components of Z-order and the benefit increases as the volume size gets larger. For

example, in the case of $n=3$, i.e., a 3-dimensional volume, type III components have at least $\frac{M}{4}$ times as high contiguity as type IV components of Z-order, i.e., $\frac{4}{M}$ less discontinuous disk head movements than Z-order.

Now given a slicing query, there are a total of 2^{n-1} components needed to answer the query and among them there is only one component of type II or IV and the other $2^{n-1}-2$ components are of type III.

3.2.1 Case I ($k = 2$)

Since the number of elements of all the 2^{n-1} components comprising a slice is the same, we have

$$\frac{1}{CO_c} = \frac{1}{2^{n-1}} \cdot \frac{1}{CO_z} + \left(1 - \frac{1}{2^{n-1}}\right) \cdot \frac{1}{\overline{CO_c}}$$

$$\frac{1}{EF_c} = \left(1 - \frac{1}{2^{n-1}}\right) \cdot \frac{1}{EF_z} + \frac{1}{2^{n-1}} \cdot 1$$

(Note that we use harmonic mean for more correct averaging of the two indices. $\overline{CO_c}$ is an average of CO_c for type II and III.)

There is always contiguity improvement over the Z-order scheme, which is upper bounded by 2^{n-1} times as high contiguity. In addition, there is always effectiveness improvement upper bounded by a factor of $\frac{2^{n-1}}{2^{n-1}-1}$. Note that the higher effectiveness also means less cache memory size required for the same slicing query.

3.2.2 Case II ($k > 2$)

Since the element size of each component of which the C-CODE bit values have p '0's increases by a factor of $(k-1)^{n-p}$, 2^{n-1} is replaced by $R_n(k)$ ($=\sum_{p=1}^n \binom{n-1}{p-1} \cdot (k-1)^{n-p}$), which is lower bounded by 2^{n-1} for $k > 2$, then,

$$\frac{1}{CO_c} = \frac{1}{R_n(k)} \cdot \frac{1}{CO_z} + \left(1 - \frac{1}{R_n(k)}\right) \cdot \frac{1}{\overline{CO_c}}$$

$$\frac{1}{EF_c} = \left(1 - \frac{(k-1)^{n-1}}{R_n(k)}\right) \cdot \frac{1}{EF_z} + \frac{(k-1)^{n-1}}{R_n(k)} \cdot 1$$

Note that the portion that the type IV component of Z-order contributes to the slice decreases while the contribution of type II increases more than any other types since the element size of type II ($p=1$) increases by the largest factor $(k-1)^{n-1}$ while that of type IV ($p=n$) does not increase. As a result, we achieve better contiguity and effectiveness as k increases.

3.3 Experimental Results

We evaluated the performance of our scheme for $k=2$, in which it is required to process slicing queries at every other value. For the evaluation, we used a subset of the visible human male anatomical image data set [Nat]. The test volumetric data consists of $2048 \times 1216 \times 800$ grid with 1-byte values, resulting in total size of 2 GB.

We ran all the tests on a single Linux machine which has dual 3.0 GHz Xeon processors with approximately 50 MB/s maximum disk I/O transfer rate. In all our experiments, we made use of only one of the two processors. Also, we used a simple

buffer management system in order to control disk I/O. The blocking factor for the data was selected arbitrarily to be $8 \times 8 \times 8$.

Figure 3.4 shows the contribution of each type of components to the total time in performing each of the slicing queries. Given a slicing query, a total of 4 components are required, among which there are only one type II and IV component and two type III components. The type IV component, which is stored in Z-order, takes the largest 48% of the total time while the type II component, which is stored in a lexicographical order in a way that the exclusive slicing type gets the highest priority, takes only 4% of the total time. Each of the two type III components takes 40% and 8% of the total time respectively. The Z-slicing takes the longest time because the slice size is the largest.

We compare the performance of the component-based data layout scheme with the Z-order combined with the n-D supercell scheme for three different types of axis-aligned slicing queries.

Figures 3.5, 3.6, and 3.7 compare the total disk I/O time for reading X, Y, or $Z=\alpha$ slices at full resolution. The component-based layout scheme always achieves better performance, by a factor of 3.2 on average. In addition, it requires 16 MB cache memory, which is 22% less than what the Z-order scheme requires. These experimental results are close to the analytical upper bound in our analysis, by which we expect the performance improvement and the cache size reduction to be upper bounded respectively by a factor of 4 and 25%. Note that the same improvements can be expected on any data of the same size given the same blocking factor, since the content of the data is not considered in any of the above schemes.

For $k=2$, an additional benefit of the component-based data layout scheme is to be able to perform all types of the half-resolution slicing queries at the maximum disk transfer rate because for every axis-aligned slicing type there is always one half-resolution type II component which is stored in an optimal way for the slicing type. Figures 3.8, 3.9, and 3.10 compare the total disk I/O time for reading X, Y, or $Z=\alpha$ slices at half resolution. The component-based layout scheme is an order of magnitude faster on average without any performance fluctuation as seen in the figures. In addition, it requires only $\frac{1}{8}$ of the cache memory size for the Z-order scheme, given the particular blocking factor. Note that we compare with the Z-order scheme at half-resolution data (i.e., type IV component).

Figure 3.11 shows a sample output slice image in each type of the slicing queries from our test volumetric data.

3.4 Discussion

The component-based data layout scheme shows two times larger intervals in performance fluctuation in the timing results. The performance fluctuation is related to the blocking factor. The blocking essentially prefetches the data under the assumption that the slicing queries are given incrementally. While the Z-order and n-D supercells scheme prefetches the data not needed by every k -th value slicing ($k > 1$) as well as the necessary data, the component-based scheme does not prefetch the unnecessary data. Thus it can effectively prefetch larger intervals given the same blocking factor.

Being able to perform the half-resolution queries at maximum disk I/O transfer rate in every slicing type (when $k=2$) becomes more beneficial when we deal with larger dimensions. For a time-series of the test volumetric data, one 3-D slice could easily be of size in the order of hundreds of megabytes to gigabytes. Unless we replicate the already large data, it will be very difficult to achieve the maximum disk I/O transfer rate for all the slicing types by using previous methods. In addition, our scheme requires only the cache memory size equal to the slice size for half-resolution queries.

While the contiguity of type III components is at least $\frac{M}{2^{n-1}}$ times as high as that of type IV components stored in Z-order as shown in the analysis, the performance result in Figure 3.4 shows that type III components take almost equivalent time (only 8% less as that of type IV components of Z-order at the worst case). We believe that this is because the disk head movement time is different between Z-order and lexicographical order. Although the Z-order has more discontinuous disk block accesses, the distance between two discontinuous disk blocks is smaller than lexicographical order. In order to investigate this further, we ran all the tests with a $16 \times 16 \times 16$ blocking factor, which is 8 times bigger in disk page size. We observed 60% less time in type III components compared to type IV components in the worst case. Overall, the change in blocking factor results in 23% less time in component-based scheme and 15% less time in the Z-order scheme due to the cache memory being twice as large, but with poorer peak processing time. Performance improvement was slightly bigger (~ 3.5) at full resolution.

In comparison to a lexicographical order, from the results in Figure 3.4, we

can expect that our scheme will always produce better results except one particular slicing type of the lexicographical order’s highest priority. Also, we can expect that the performance of retrieving slices at grid values that are not every k -th will at least be better than the worst case in a lexicographical order.

3.5 Conclusions

We have presented a new data layout scheme to efficiently handle out-of-core axis-aligned slicing queries of very large multidimensional rectilinear grids. Our component-based data layout scheme features much faster processing time for any types of axis-aligned slicing queries at every k -th value, $k > 1$, with no data duplication, and can be generalized to any high dimension. We have analytically shown that our scheme provides faster processing time and requires less cache memory than the typical Z-order scheme for any type of axis-aligned out-of-core slicing queries at every k -th value ($k > 1$), without any data replication. Through experimental results, we have also demonstrated that it can achieve 3-fold and 10-fold performance improvements requiring only 78% and 12% of the cache memory size for the Z-order scheme at full and half resolution respectively.

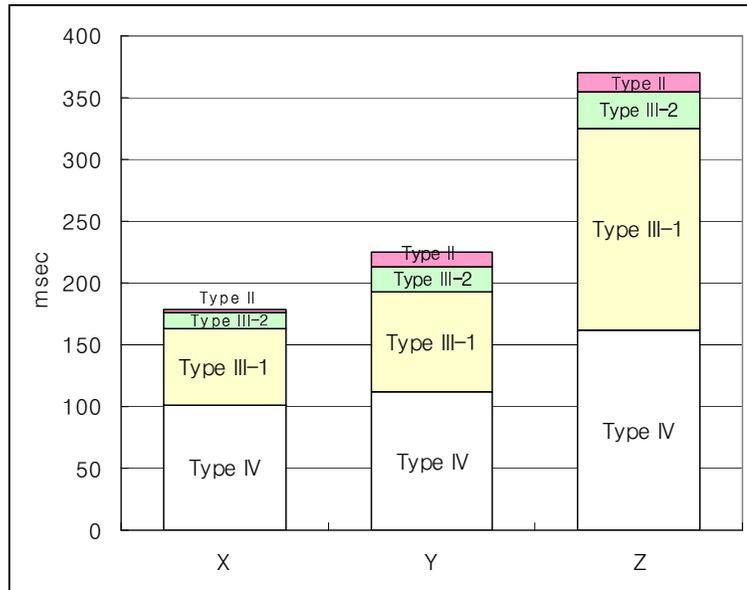


Figure 3.4: Contribution of each type of components to total time for X, Y, and $Z=\alpha$ queries.

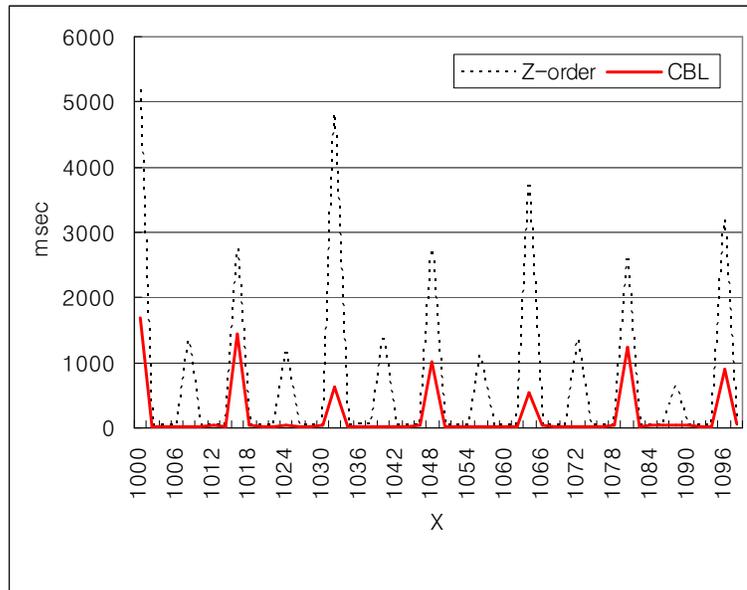


Figure 3.5: Performance comparison for loading $X=\alpha$ slices (1216×800).

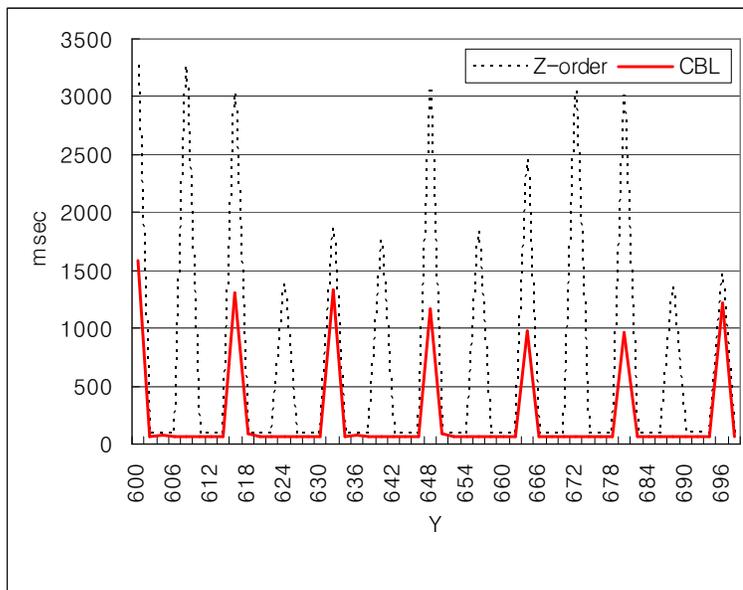


Figure 3.6: Performance comparison for loading $Y=\alpha$ slices (2048×800).

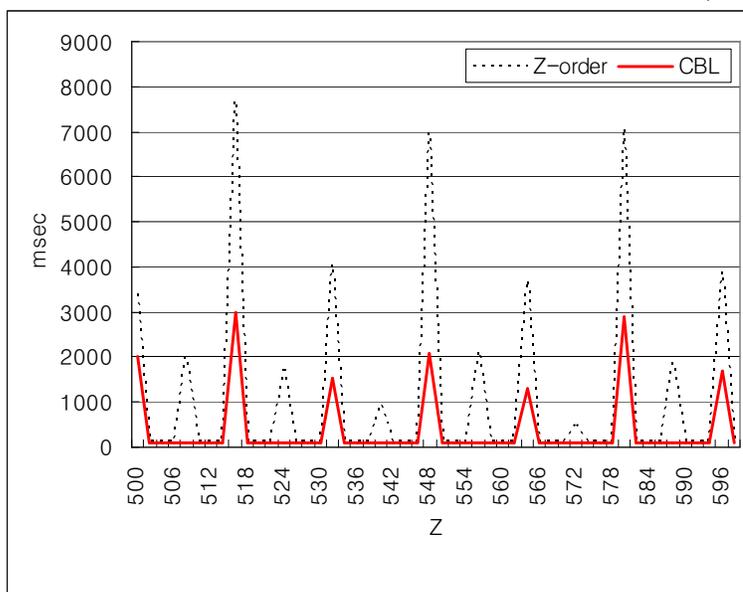


Figure 3.7: Performance comparison for loading $Z=\alpha$ slices (2048×1216).

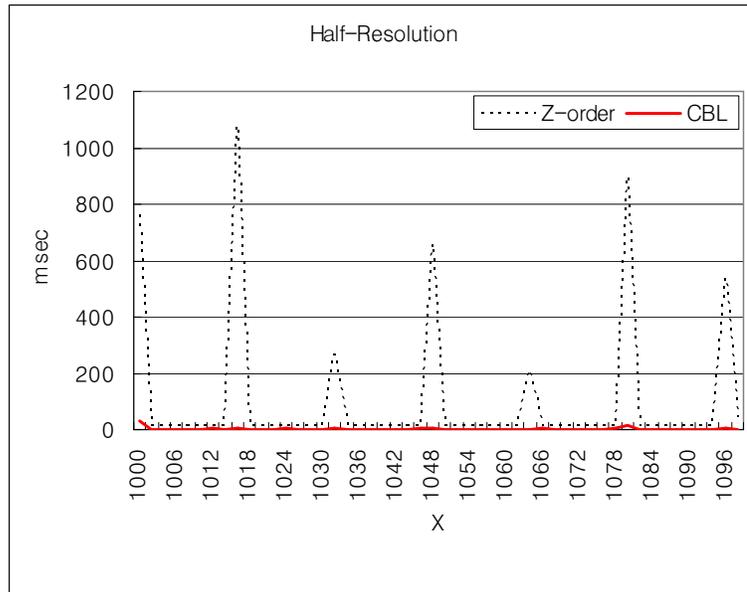


Figure 3.8: Performance comparison for loading $X=\alpha$ slices at half resolution (608×400).

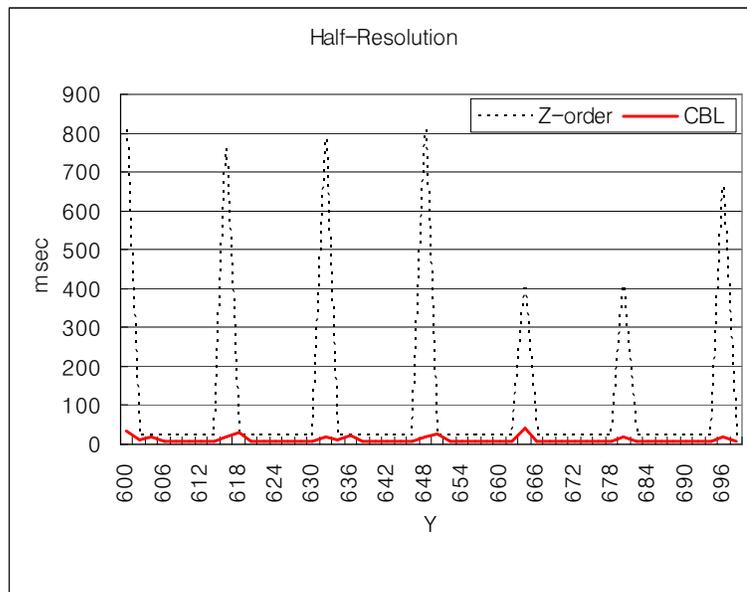


Figure 3.9: Performance comparison for loading $Y=\alpha$ slices at half resolution (1024×400).

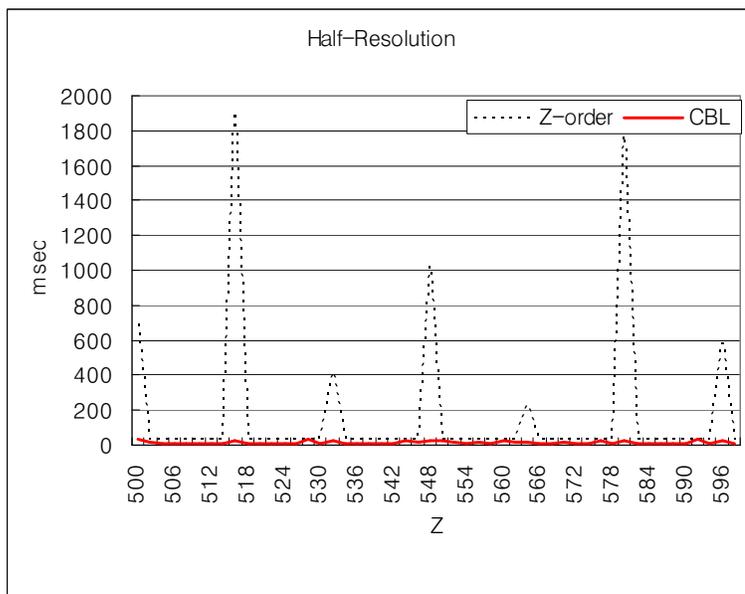


Figure 3.10: Performance comparison for loading $Z=\alpha$ slices at half resolution (1024×608).

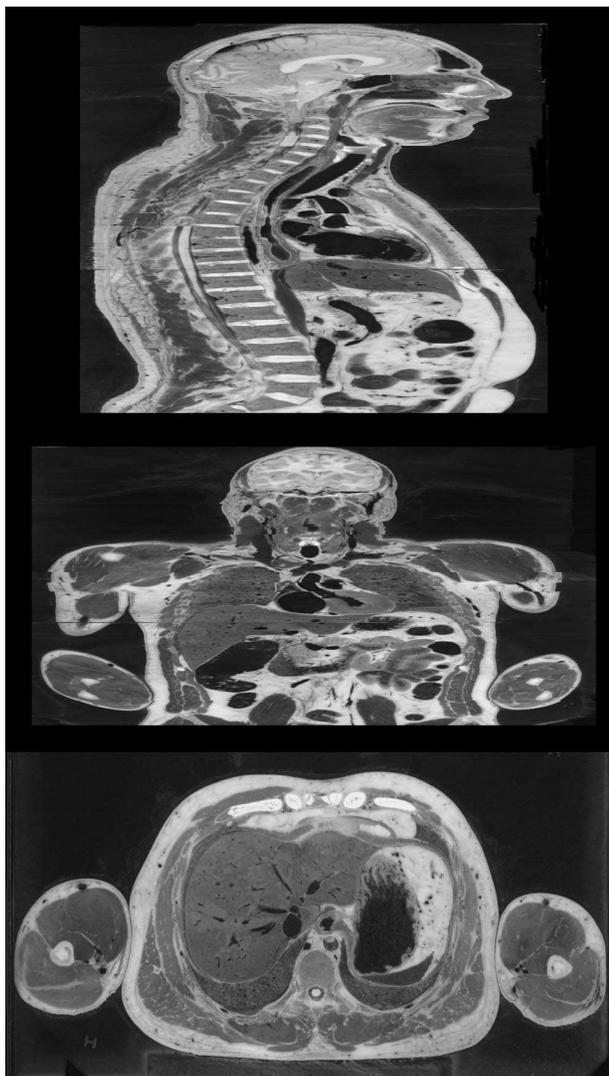


Figure 3.11: Sample slice images of the test volumetric data at X , Y , and $Z=\alpha$.

(1216×800 , 2048×800 , and 2048×1216 from top to bottom.)

Chapter 4

Efficient Out-of-Core Indexing Structure

Isosurface rendering is an important visualization technique that enables the visual exploration of volumetric data using surfaces. Since it basically renders surfaces, it is often preferable to volume rendering due to faster rendering time, especially for large volumetric data. Since the introduction of the Marching Cubes algorithm [LC87] that performs a complete scan of all the data cells, a variety of more efficient algorithms have appeared in the literature. These algorithms include a preprocessing step that constructs an indexing structure to speed up the identification of active cells (cells cut by the isosurface). Such techniques use either spatial data structures such as the octree [WG92], or the span space [LSJ96], or the interval tree [CMM⁺97], or the seed propagation technique [BPS96]. The case when the data is too large to fit in main memory has recently received significant attention. There are two main directions that have been pursued to deal with such large datasets. The first focuses on the development of out-of-core efficient implementations of internal memory algorithms (e.g. [CS97, CSS98]), while the second direction makes use of parallel processing to achieve good performance on multiprocessor systems (e.g. [BPTZ99, CFSW01, ZBR02]).

For time-varying volumetric data, a 4-D isosurface can be defined as the set of

points that satisfy $f(x, y, z, t) = c$, for a given isovalue c . The problem of extracting linear approximations to 4-D isosurfaces from sampling over a structured grid was addressed by Weigle and Banks [WB98] and Bhaniramka et al. [BWC04] but these techniques are computationally quite expensive and are only practical for very small datasets. Instead of dealing with the extraction of 4-D isosurfaces, most of the other work on time-varying data has focused on the problem of generating isosurfaces for each of the given time steps separately. For example, Chiang [Chi03] develops an out-of-core version of the hierarchical temporal tree described in [She98] by incorporating an out-of-core version of the interval tree indexing structure. Another example is the temporal branch-on-need (T-BON) octree described in [SH00] based on the branch-on-need octree structure [WG92].

In this chapter, we address the problem of exploring 4-D isosurfaces of time varying volumetric data by rendering the 3-D isosurfaces obtained through an axis-orthogonal hyperplane cut. Such a cut is defined by a constraint of the form $x[y, z, \text{ or } t]=\alpha$, for a user-specified α . This is an axis-orthogonal hyperplane, where the axes are x, y, z , or t . Note in particular that a temporal cut $t = \alpha$ may fall in between two consecutive time steps in which case we will use interpolation on the fly to generate the corresponding isosurface. Our approach provides a rich environment that enables users to more effectively observe patterns and trends along the temporal dimension through the use of arbitrary spatial cuts $x[y, \text{ or } z]=\alpha$, and to render isosurfaces for any intermediate time step, without explicitly constructing the simplices needed for extracting the 4-D isosurface or decomposing hypercubes into simplices, as in [WB98, BWC04]. Carrying out such a plan on a large scale

dataset requires a space efficient and scalable indexing structure that allows the fast retrieval of *active data blocks* from disk, that is, blocks that are necessary to build the 3-D isosurface corresponding to the parameters specified by the user, an *isovalue* and an *axis-orthogonal hyperplane*.

None of the existing indexing structures can be easily extended to effectively solve such a problem, except possibly for the T-BON structure [SH00] but in this case we can only use temporal cuts along the given time steps. Moreover, The T-BON structure consumes a large amount of space that grows linearly with the number of time steps because it does not exploit any type of possible coherence across the temporal dimension. This lack of scalability becomes more problematic as we generate higher and higher resolution data in every dimension including the time dimension.

In the chapter, we present a new efficient out-of-core multidimensional indexing structure, Information-Aware 2^n tree (IA- 2^n tree). Our strategy is to basically build an n-dimensional indexing structure on n-dimensional data because it can exploit the coherence across all n dimensions and thus lead to compact size, addressing the scalability problem. However, we need to also consider indexing effectiveness as well as the scalability. The effectiveness of out-of-core indexing can be measured by how much data is actually needed from the loaded data because the finest indexed object is not an individual voxel, but a group of data of disk page size. Thus, using the new structure, we seek to increase the ratio of indexing effectiveness to indexing structure size, which we define as *indexing structure efficiency*.

4.1 Information-Aware 2^n tree

Information-Aware 2^n trees (IA- 2^n trees) are basically 2^n trees (e.g. quadtrees for 2-D and octrees for 3-D [Sam90]) for n -dimensional data. However, it is different in terms of the extent ratios of a subvolume determined when multi-dimensions are integrated into one hierarchical indexing structure. The coherence information along each dimension is extracted and used for the decision so that each subvolume contains as much coherence as possible along each dimension.

4.1.1 Dimension Integration

We present an entropy-based dimension integration technique. Entropy [CT91] is a numerical measure of the uncertainty of the outcome for an event x , given by $H(x) = -\sum_{i=1}^n p_i \log_2 p_i$, where x is a random variable, n is the number of possible states of x , and p_i is the probability of x being in state i . This measure indicates how much information is contained in observing x . The more the variability of x , the more unpredictable x is, and the higher the entropy. For example, consider the series of scalar field values for a voxel v over the time dimension. The temporal entropy of v indicates the degree of variability in the series. Therefore, high entropy implies high information content, and thus more resources are required to store or communicate the series. Note that the entropy is maximized when all the probabilities p_i are equal.

We use the entropy notion to determine the relative sizes of the extents of the supercell (that is, a leaf node in the trees). Higher entropy of a dimension relative

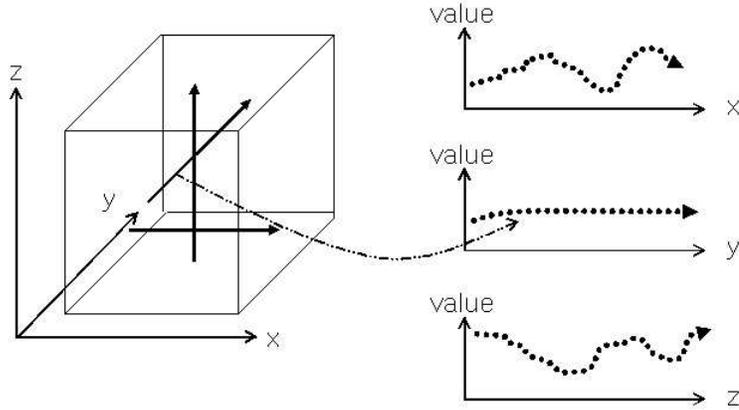


Figure 4.1: Entropy estimation in each dimension. Note that the y dimension has almost zero entropy in this example.

to the other dimensions implies that this dimension needs to be split at finer scales than the other dimensions. For example, if a temporal entropy is twice as much as the spatial entropy, we design the supercell to be of size $s \times s \times s \times \frac{s}{2}$ ($x \times y \times z \times t$), where s is the size of the spatial dimension of the supercell.

Figures 4.1 and 4.2 show how this entropy-based dimension integration leads to an indexing structure for the 3-D case. Figure 4.1 shows an extreme case in which the values along the y dimension remain almost constant over all possible (x, z) values (that is, the entropy of y is almost zero) while each of the x and z dimensions has some degree of variability. The supercell size and the corresponding hierarchical indexing structure will be designed as shown in Figure 4.2 (b), that is, it has a quadtree structure unlike the standard octree of Figure 4.2 (a) in which the supercell has the same size in each dimension.

To estimate the entropy, we select a set of time steps, referred to as *reference time steps*, and compute spatial entropies for the corresponding 3-D volumes. These

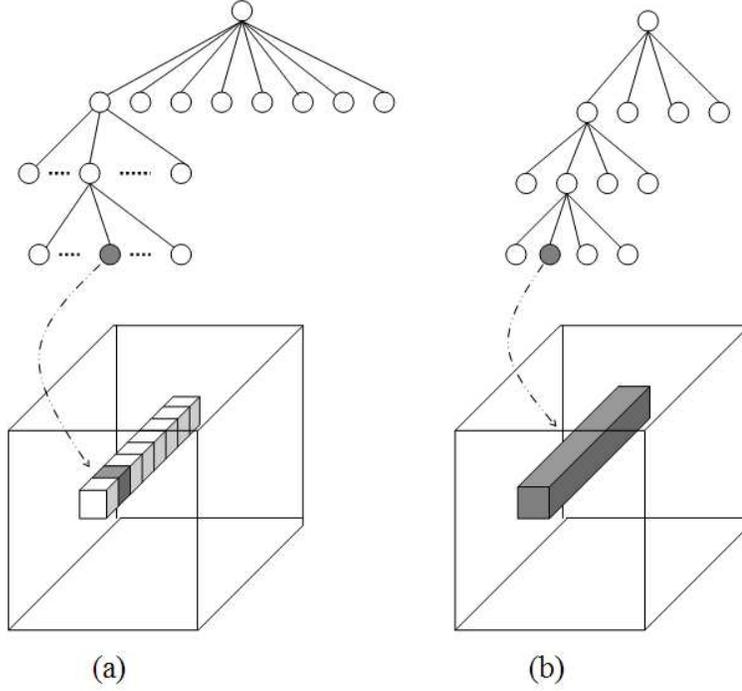


Figure 4.2: Different supercell sizes and corresponding hierarchical indexing structures for the data of Figure 4.1: (a) standard supercell; (b) information-aware supercell.

time steps are selected uniformly from the overall time series (or can be adaptively sampled at a higher rate in the time domain of high temporal entropy for more correct estimation). For each corresponding 3-D volume, we compute spatial entropies for a random subset of subvolumes. Consider for example one sampled subvolume shown in Figure 4.3. Assuming the scalar field values $v \in [1, 2, \dots, n]$, the entropy E_x is defined as follows (direction l_{yz} is parallel to the x -axis and is anchored at the point (y, z)) :

$$P_v^{yz} = \frac{\# \text{ of occurrences of the scalar field value } v}{\text{Total } \# \text{ of voxels on the direction } l_{yz}} \quad (4.1)$$

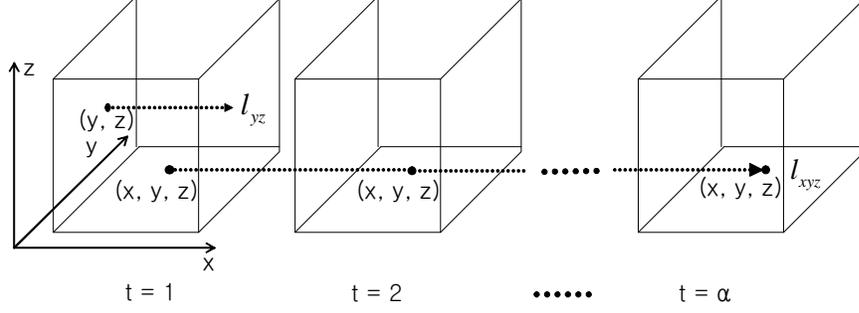


Figure 4.3: A sampled subvolume along time steps. Entropy computation of x dimension is performed on the line l_{yz} and averaged over all (y,z) values, while the one of t (time) dimension on l_{xyz} and averaged over all (x,y,z) values.

$$E_x^{yz} = - \sum_{v=1}^n P_v^{yz} \log_2 P_v^{yz} \quad (4.2)$$

$$E_x = \frac{\sum_{y,z} E_x^{y,z}}{\sum_{y,z}} \quad (4.3)$$

where $\sum_{y,z}$ is over the grid points in the (y, z) plane.

On the other hand, the entropy of the temporal dimension E_t is computed by taking values at the same voxel along the time-axis as follows.

$$P_v^{xyz} = \frac{\# \text{ of occurrences of the scalar field value } v}{\text{Total } \# \text{ of voxels along the temporal direction } l_{xyz}} \quad (4.4)$$

$$E_t^{xyz} = - \sum_{v=1}^n P_v^{xyz} \log_2 P_v^{xyz} \quad (4.5)$$

$$E_t = \frac{\sum_{x,y,z} E_t^{x,y,z}}{\sum_{x,y,z}} \quad (4.6)$$

where $\sum_{x,y,z}$ is the number of voxels at a time step.

If the number of the possible scalar field values is large (as for example would be the case for floating point scalar field values), we first *quantize* the original values into n values using a non-uniform quantizer such as the Lloyd-max quantizer [Jai89].

Note that this quantization is only used for the purpose of computing the entropies.

Even though it can apply to general cases, we are primarily concerned about establishing the relationship between spatial and temporal dimensions because there is usually constant difference in the coherence of data values between the two different types of dimensions. Thus we compute the *spatio-temporal entropy ratio* defined as the ratio of the average spatial entropy to the temporal entropy.

We note that in general a time series will consist of a number of temporal domains during which the spatio-temporal entropy ratio can be different. Our general strategy is to decompose the time series into a set of temporal regions, each of which will be characterized by its spatio-temporal entropy ratio. Hence we build a separate IA-Octree for each temporal region.

4.1.2 Indexing Structure

The starting point of our IA-Octree is the 4-D octree structure that usually divides the 4-dimensional space into 2^4 subspaces. We make use of the spatio-temporal entropy ratio to determine the branching factor and the size of the 4-D volumes at the leaves, and follow the branch-on-need strategy [WG92] as well. We delay the branching until it is absolutely necessary as in [WG92]; however the temporal branching is further delayed by a factor of the spatio-temporal entropy ratio if the ratio is more than 1 or expedited by that factor if it is less than 1. Each tree node contains the minimum and maximum values of the scalar fields in the region represented by the node, a pointer to the first child, and a branching factor.

4.2 Tree Traversal and Controllable Delayed Fetching

To optimize disk accesses and make rendering scalable, we organize the access of active data blocks in large contiguous chunks, each of which corresponds to a subspace of the original data as shown in Figure 4.4 for the 3-D case. We essentially delay the retrieval of a supercell until a subtree, whose root is at a preset level, is completely traversed. We traverse the IA-Octree in a depth-first order by checking whether the node's minimum and maximum values span the isovalue and the user-specified hyperplane intersects the node. Once we reach the active supercells at the lowest level, we insert each supercell into a *priority queue* using the priority key (t,z,y,x) in the ordering that complies with the underlying data layout.

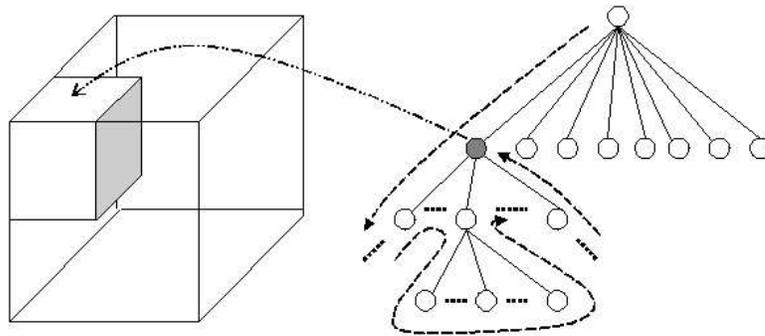


Figure 4.4: Controllable delayed data fetching. Disk accesses for the active supercells in the subvolume corresponding to the grey node are delayed until the traversal revisits the grey node.

The actual data fetching of the active supercells is delayed until the depth-first order tree traversal revisits the nodes at a preset level. For example, disk accesses

for the active supercells in the subvolume in Figure 4.4 are delayed until the tree traversal revisits the grey node and then actual data fetching starts by popping the top entry of the priority-queue and issuing the corresponding disk access. This process continues until the priority-queue is empty after which the tree traversal proceeds in depth-first order. Note that the example shows a 3-D case for illustration purposes.

The controllable delayed fetching enables us to optimize the disk accesses and is also quite effective in enabling scalable rendering which we explain in the next section. We adjust the preset level at which delayed fetching is carried out depending on the sizes of the data and main memory, as well as rendering speed. Setting the prefetching level is a trade-off between efficient access (the higher the level the more efficient the disk access is) and scalability.

4.3 Scalable Rendering

Each time we fetch the data at a preset level, isosurface extraction for the corresponding subspace is performed. Scalar field values corresponding to all the active supercells are loaded into a main memory. Consider a spatial-cut query type specified by an isovalue and a hyperplane $x[y \text{ or } z] = \alpha$. If α is along a grid line, we extract the corresponding slice from each of the loaded supercells, and organize all the slices together to constitute a 3-D volume. For example, for the $x = \alpha$ hyperplane query, we create a 3-D volume whose dimensions are equal to the subvolume's y , z , and t dimensions. Then we use the standard marching cubes

algorithm [LC87] to extract triangles in the 3-D volume. However the volume is selectively traversed because we know which portions of the volume are filled with the data from the active supercells. This partial assembly of slices and triangle extraction performed after every delayed fetching enables us to avoid forming a large 3-D volume in main memory for triangle extraction in the spatial-cut query. If α is not along a grid line, we extract the two consecutive slices containing α , and perform a linear interpolation pointwise, to generate the slice corresponding to α . The process is now completed as before. For temporal-cut queries, triangle extraction is straightforward for cuts along one of the given time steps since each loaded disk page corresponds to a 3-D supercell that satisfies the query. Otherwise, we load the two supercells corresponding to consecutive time steps containing α , and interpolate to generate the appropriate supercell. Now we can proceed as in the spatial cut case using the loaded supercells.

Isosurface rendering is incrementally performed by rendering the extracted triangles whenever the number of triangles reaches some preset threshold value, concurrently with tree traversal, data movement, and triangle extraction.

4.4 Experimental Results

We compared the indexing structure efficiency of our IA 2^n -tree with a typical 2^n -tree and also the T-BON scheme [SH00], which is one of the two most popular schemes for time-varying isosurface rendering and can also handle slicing queries. For evaluation, we consider two large time-varying volumetric data sets:

the Richtmyer-Meshkov data set for time steps 100 – 139, each down-sampled by two along each spatial dimension, and the Five Jets data set [JET] consisting of 2000 time steps. Each time step of the Richtmyer-Meshkov data set involves a $1024 \times 1024 \times 960$ grid with one-byte scalar values resulting in total 40 GB data set. The Five Jets data set consists of $128 \times 128 \times 128$ grid with 4-bytes floating point values resulting in total 16 GB.

We ran all the tests on a single Linux machine which has dual 3.0 GHz Xeon processors with approximately 50 MB/s maximum disk I/O transfer rate. In all our experiments, we made use of only one of the two processors. Also, we used a simple buffer management system in order to control disk I/O.

	Ratio (IA- 2^n / 2^n)
Loaded Data	1.02
Effectiveness	0.97
Disk Access Time	1.01
Tree Traversal	0.93
Total Time	1.00

Table 4.1: Query performance comparison between IA 2^n -tree and 2^n -tree for the Richtmyer-Meshkov data set. The results are the average values over various types of slicing and different isovalues.

We first compare the IA 2^n -tree with a 2^n -tree for the Richtmyer-Meshkov data set. Using the entropy measure, we obtained a spatio-temporal entropy ratio equal to 1.5 over the time steps 100 – 139 for the data set, resulting in 30% less indexing structure size than the 2^n -tree. However, Table 4.1 shows that it experiences only 3% indexing effectiveness reduction. Note that the tree traversal time decreases

because the number of nodes that the tree has to visit decreases. Overall, it results in 1.4 times better indexing structure efficiency.

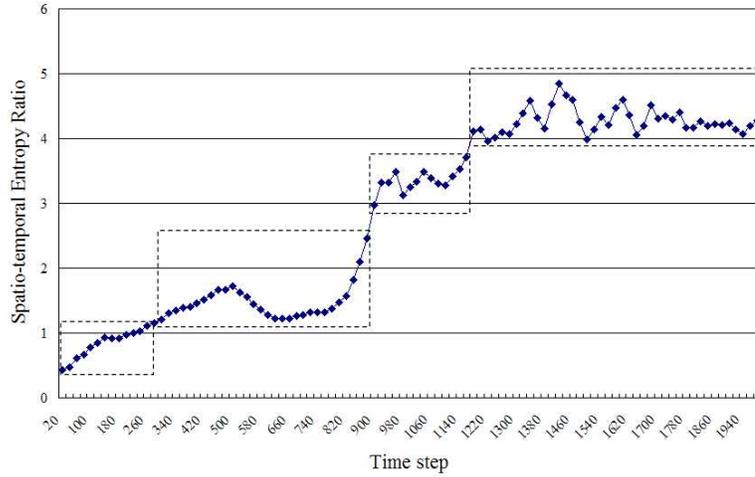


Figure 4.5: Spatio-temporal entropy ratios computed at uniformly selected 100 reference time steps among the 2000 time steps in the Five Jets data. Each dashed box corresponds to a time region.

Now we also compare the IA 2^n -tree with the T-BON scheme for both the Richtmyer-Meshkov and the Jet data set. The size of IA 2^n -tree is only about 1/9 of the T-BON structure for the Richtmyer-Meshkov data set. For the Jet data set, we arbitrarily divided the temporal domain of the Five Jets data set into four time regions (see Figure 4.5) having respectively the spatio-temporal entropy ratios of 0.5, 1, 3, and 4. We separately built our tree on each time region. The size of IA 2^n -tree is only about 1/8 of the T-BON structure for the Jet data. However, Table 4.2 shows that the indexing effectiveness reduction is only 9% and 6% respectively for each of the two data sets. It results in about 8 times better indexing structure efficiency.

	Richtmyer-Meshkov	Jet
	Ratio (IA-2 ⁿ / T-BON)	Ratio (IA-2 ⁿ / T-BON)
Loaded Data	1.10	1.05
Effectiveness	0.91	0.94
Disk Access Time	1.15	1.09
Tree Traversal	0.71	0.70
Total Time	0.98	0.98

Table 4.2: Query performance comparison between IA 2^n -tree and T-BON for the Richtmyer-Meshkov and the Jet data set. The results are the average values over various types of slicing and different isovalues.

The experimental results show that we can even obtain slightly better timing results. This is because the effect of the increased data transfer due to the reduced effectiveness can be mitigated by memory cache effect, but there is no way that the longer tree traversal time of the larger T-BON structure and 2^n -tree can be mitigated in the course of successive queries.

4.5 Conclusions

We introduced a new indexing structure called Information-Aware 2^n -trees. Building a series of (n-1)-dimensional indexing structures causes a scalability problem in the current situation of continually growing resolution in every dimension. However, building a single n-dimensional indexing structure can cause an indexing effectiveness problem compared to the former case. The Information-Aware 2^n -tree is an effort to maximize the indexing structure efficiency by measuring the entropy of the data and use it to ensure that the subdivision of space has as much coherence

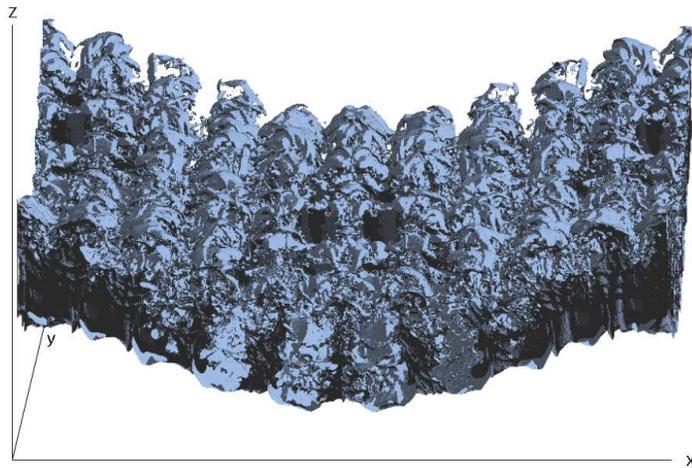


Figure 4.6: Isosurface of the Richtmyer-Meshkov instability data set ($1024 \times 1024 \times 960$) rendered at isovalue=70 and $T=139$.

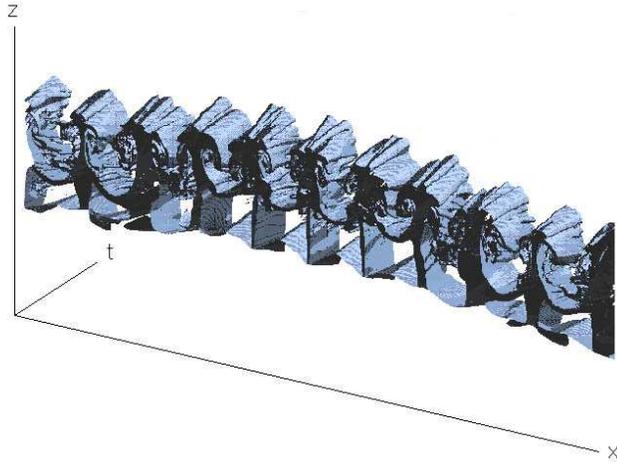


Figure 4.7: Isosurface of the Richtmyer-Meshkov instability data set ($1024 \times 1024 \times 960$, $T=100-139$) cut by isovalue=70 and $Y=300$.

as possible along each dimension. As a result, it provides higher indexing structure efficiency than the previous 2^n tree or $n \times 2^{n-1}$ trees.

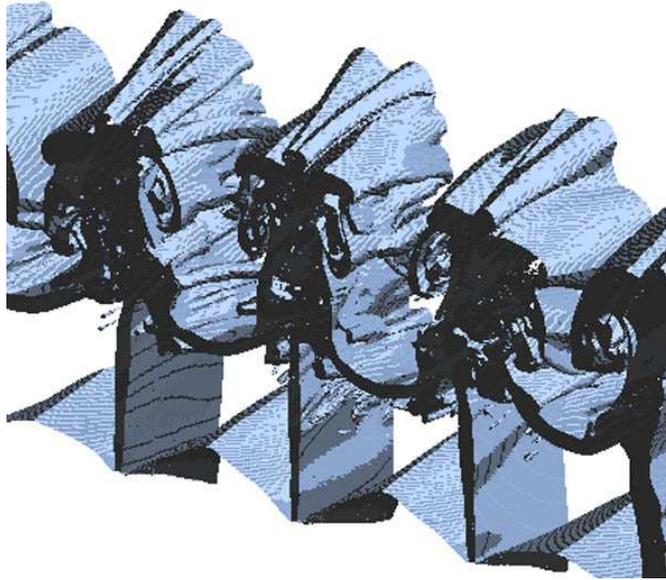


Figure 4.8: Zoomed image of Fig. 4.7.

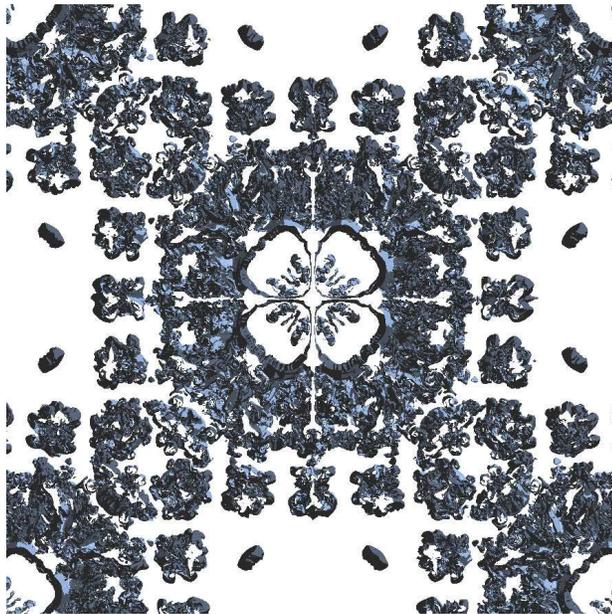


Figure 4.9: Isosurface of the Richtmyer-Meshkov instability data set ($1024 \times 1024 \times 960$, $T=100-139$) cut by $\text{isovalue}=70$ and $Z=500$. (Time axis is orthogonal to the paper.)

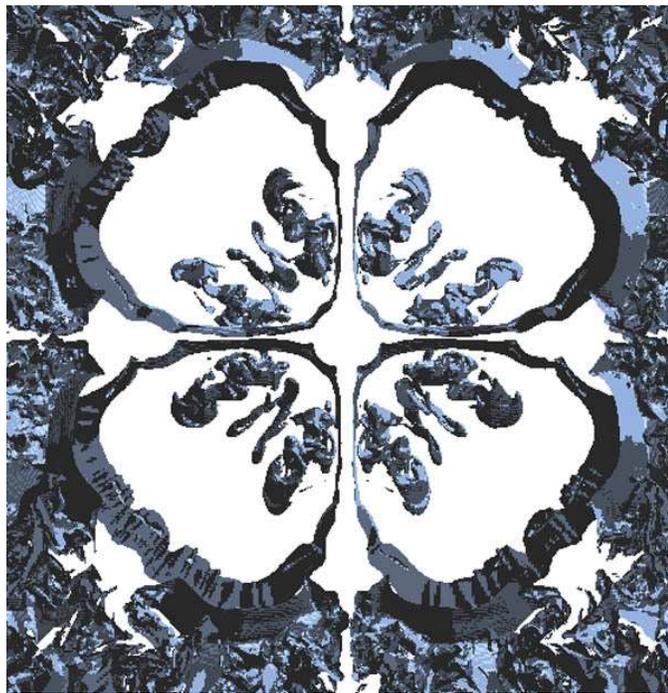


Figure 4.10: Zoomed image of Fig. 4.9.

Chapter 5

Volume Ray Casting on Cell Broadband

Engine

Ray casting [Lev90] has been recognized as a fundamental volume rendering technique that can produce very high quality images. However, its application has been limited only to datasets of very small sizes because of its high computational requirements and its irregular data accesses. In particular, the amount of data to be processed and the generally irregular access patterns required make it very hard to exploit caches, which in general result in high memory latencies. Thus, it is very difficult for current general purpose desktop computers to deliver the targeted level of interactivity for most practical volumetric datasets.

Significant research efforts have attempted to accelerate volume rendering using graphics hardware. A representative technique is based on the exploitation of the texture-mapping capabilities of the graphics hardware [CCF94, KMM⁺01]. The texture-mapping based volume rendering has enabled a single PC with a commodity graphics card to achieve interactive frame rates for moderate-sized data. However, the rendering quality is generally not satisfactory [MHB⁺00]. Also, the size of the data that can interactively be rendered is limited by the graphics memory size, which is typically substantially smaller than system memory. When the data set

does not fit in the graphics memory, which is often the case in time-series data, interactivity becomes very hard to achieve because data has to be transferred from system memory to graphics memory, a process that usually takes at least an order of magnitude more time than the graphics memory bandwidth.

On the other hand, in order to address the increasing demands on interactive, higher-quality video rendering, Sony, Toshiba and IBM (STI) teamed together to develop the Cell Broadband Engine (Cell B.E.) [JAKS05], which is the first implementation of a chip multiprocessor with a significant number of general purpose programmable cores. The Cell B.E. is a heterogeneous multicore chip capable of massive floating point processing optimized for computation-intensive workloads and rich broadband media applications, and thus opening up the opportunity to put the ray casting algorithm into widespread, practical use.

In this chapter, we introduce a carefully tailored, efficient parallel implementation of volume ray casting on the Cell B.E. In general, achieving high performance for demanding computations with highly irregular data movements is extremely difficult on the Cell B.E. as it was primarily designed for large scale SIMD operations on media data streaming through the core processors. In our work, we aim to take full advantage of the unique capabilities of the Cell B.E while overcoming its unique challenges. In particular, we achieve an optimized implementation of two main acceleration techniques for volume ray casting [Lev90] - empty space skipping and early ray termination - on the Cell B.E.

We present a streaming model based scheme to efficiently employ both acceleration techniques. This scheme makes an effective use of the heterogeneous cores and

asynchronous DMA features of the Cell B.E. In our scheme, a PPE (the PowerPC processor in the Cell B.E.) is responsible for traversing a hierarchical data structure and generating the lists of intersecting voxels along the rays over non-empty regions, as well as it is responsible for feeding the SPEs (Synergistic Processing Elements - SIMD type cores with very high peak floating point performance) with the corresponding lists. The SPEs are responsible for actual rendering of the data received from the PPE, and naturally implement the early ray termination acceleration technique. To deal with the speed gap between the heterogeneous cores (PPE versus SPEs), we introduce a couple of important techniques.

Our streaming model based scheme provides the following two key benefits. First, we essentially remove the overhead caused by traversing the hierarchical data structure by overlapping the empty space skipping process with the actual rendering process. Second, using prefetching, we essentially remove memory access latency, which has been the main performance degradation factor that is due to the irregular data access patterns. In addition to these two key benefits, we can also achieve better SIMD utilization in the SPEs because the SPEs know the sampling voxels to process in advance and thus they can pack them into SIMD operations.

5.1 Cell Broadband Engine Overview

The Cell Broadband Engine (Cell B.E.) [JAKS05], as shown in Figure 5.1, consists of one 64-bit PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs), all connected together by a high-bandwidth Element

Interconnect Bus (EIB).

Each SPE contains a Synergistic Processor Unit (SPU), a Memory Flow Controller (MFC) and 256K bytes of local storage (LS). The MFC has DMA engines that can asynchronously transfer data across the EIB between the LS and main memory. Each SPU contains a 128-bit-wide SIMD engine enabling 4-way 32-bit floating point operations. The SPU can not access main memory directly. It obtains data and instruction from its 256 Kbytes local storage and it has to issue DMA commands to the MFC to bring data into the local store or write results back to main memory.

Cell allows using the same virtual addresses to specify system memory locations regardless of processor element type and thus, it enables seamless data sharing between threads on both the PPE and SPE. It is also possible for SPEs to reference different virtual memory spaces associated with respective applications executing concurrently in the system.

With a clock speed of 3.2 GHz, the Cell B.E. has a theoretical peak performance of 204.8 GFlops/s. The EIB supports a peak bandwidth of 204.8 GBytes/s for on-chip data transfers. The memory interface controller provides 25.6 GBytes/s bandwidth to main memory at peak performance.

5.2 Primary Work Decomposition and Allocation

In this section, we describe our primary work decomposition and assignment scheme for volume ray casting on the Cell B.E. Our scheme is illustrated in Figure 5.2.

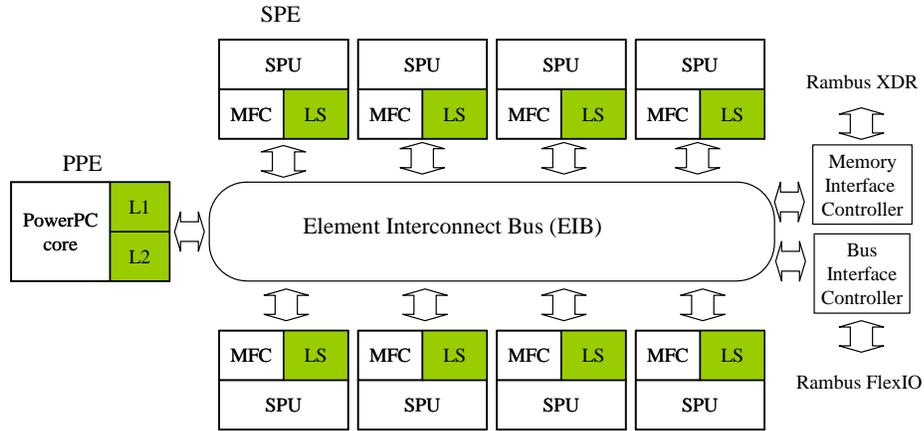


Figure 5.1: Cell Broadband Engine Overview [IBM06].

Our work decomposition scheme is based on fine-grain task parallelism that achieves load balancing among the SPEs as well as matching workload between the PPE and the SPEs. In ray casting, the overall concurrency is obvious since we can compute each pixel value on the screen independently of all the other pixels. To take advantage of this fact, we divide the screen into a grid of small tiles. Each tile will be independently rendered by a certain SPE. The size of the tile should be small enough to balance loads between the SPEs. Also, an SPE should be able to store in its very limited local memory the task list generated by the PPE as well as the tile image itself. Note that the size of the task list from the PPE increases as the tile size does. On the other hand, the tile size should be large enough to ensure enough work between synchronizations.

The high communication bandwidth of the Cell B.E. makes it possible to achieve excellent performance using image-based fine-grain decomposition despite the fact that the Cell B.E. is essentially a distributed memory system, in which object-based coarse-grain decomposition is usually chosen. This fine-grain task par-

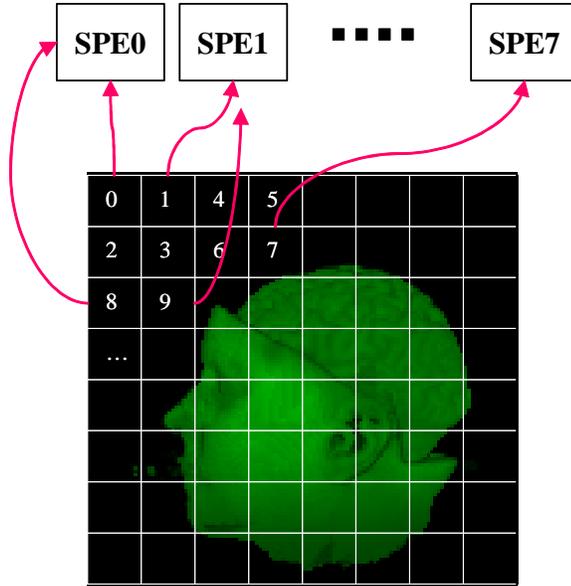


Figure 5.2: Work decomposition and assignment to the SPEs.

allelism enables us to achieve near-optimal load balancing and also to overcome the limited local memory size.

Our work assignment scheme is static. We assign each tile to each SPE in some order as shown in Figure 5.2, which shows Z-order based scheme. Such an ordering tries to exploit spatial locality as much as possible. Even though the assignment is static, the time it takes to render all the assigned tiles in each SPE is almost identical for the different SPEs because of the fine-grain work decomposition.

5.3 Implementation of Acceleration Techniques

There are two most widely used acceleration techniques for ray casting [Lev90]: empty space skipping and early ray termination. To skip empty space, one usually constructs a hierarchical data structure that stores the information about which

subvolume is empty and skips the subvolume during traversal. This acceleration technique is very useful in most volumetric datasets since they usually have significant portions that are empty space. On the other hand, early ray termination can also save significant time by stopping a ray traversal after its opacity value reaches some threshold since its final pixel value will hardly change by further ray traversal. This acceleration technique is particularly useful when the objects embedded in the volume are mostly opaque. Efficiently implementing these two acceleration techniques is very important since it significantly affects the ray casting performance.

5.3.1 Streaming model for acceleration

Our basic idea for implementing the acceleration techniques on the Cell B.E. is to assign empty space skipping to the PPE and early ray termination to the SPEs. The PPE is a full-fledged 64-bit PowerPC with L1 and L2 caches, and hence can handle branch prediction much better than the SPE. Clearly the PPE is a better candidate for efficiently traversing a hierarchical data structure. Furthermore, the SPE would have substantial overhead in handling empty space skipping due to the limited local memory size as the size of the hierarchical data structure increases. On the other hand, the SPE is ideal for the rendering work since it was designed for compute-intensive workloads using SIMD style operations. Thus, we naturally implement early ray termination on the SPE.

We streamline the empty space skipping process and the actual rendering process. Given a ray, the PPE traverses the hierarchical data structure along the

ray direction and collects ray segments (defining the corresponding sampled voxels) which are only in non-empty subvolumes. Each ray segment is characterized by two parameters R and L such that R is the ray offset from the viewpoint and L is the length of the corresponding segment. The collected ray segments for all the pixels of a tile are concatenated and transferred to the SPE in charge of the corresponding tile, which then renders the tile with early ray termination option. This streaming model is illustrated in Figure 5.3.

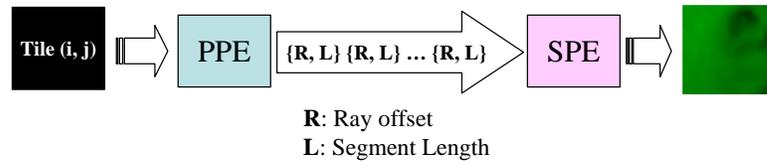


Figure 5.3: Our streaming model for acceleration techniques.

In this streaming model, the PPE side is responsible for generating and sending only the contributing (non-empty) ray segments to the SPEs. For that, we use a simple 3-D octree data structure, in which each node has 8 children and stores a maximum value of any voxel in the subvolume rooted at the node. However we should carefully set the leaf node size. The smaller the size of the leaf node, the more traversal time and the more amount of data needs to be transferred to the SPEs. However, the larger the leaf size, the more empty space will need to be handled by the SPEs, eventually leading to significant increase in rendering time. Empty space can be determined by either opacity values after classification or raw voxel values. If opacity values are used, the octree would have to be updated every time the classification table is changed.

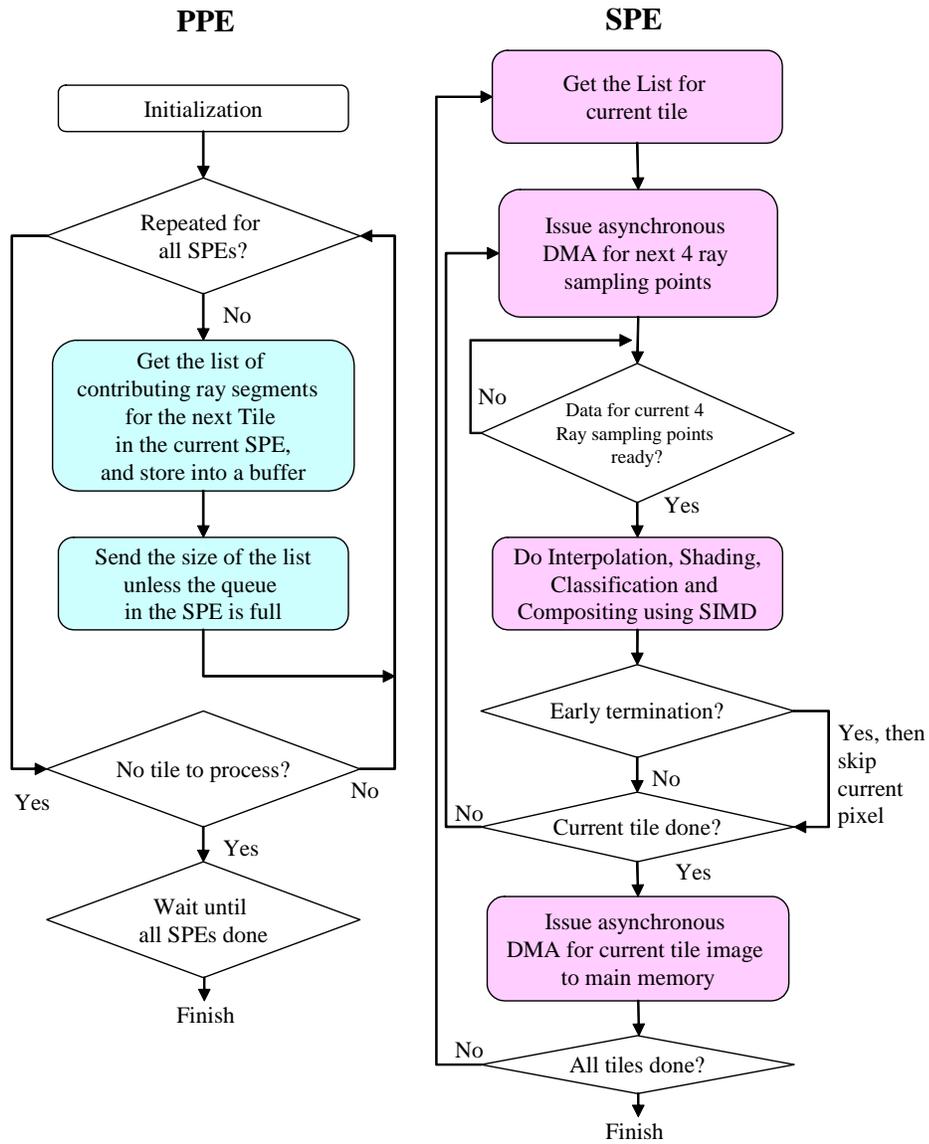


Figure 5.4: Main algorithms in PPE and SPE.

SPEs are responsible for the actual rendering process. An SPE waits until it gets a signal from the PPE that it has collected all the contributing ray segments corresponding to all the pixels in the tile under consideration. Once it receives the signal from the PPE, it starts the rendering process for the corresponding tile. The rendering process consists of four main steps: prefetching, interpolation, shading/classification, and compositing. During the rendering, four ray sampling points are processed together in a loop to exploit the SIMD capabilities of the SPE. First, for prefetching, we take advantage of the asynchronous DMA feature of the Cell B.E. and use double buffering [IBM06]. We prefetch the next 4 subvolumes required for rendering the next 4 ray sampling points into a buffer. To achieve peak performance, we arrange the volume residing in main memory into a 3-D grid of small subvolumes. If the 4 subvolumes necessary for rendering the current 4 ray sampling points are ready, we concurrently perform 4 tri-linear interpolations using 4-way SIMD instructions to reconstruct the signals. Reconstructed values are mapped to a color and opacity value using shading and classification tables. Finally, we composite the 4 values sequentially since compositing can not be concurrently done. However, we concurrently composite the R, G, B values, and hence we utilize 3/4 of the SIMD capability of the SPE. The final opacity value is then tested for early ray termination, and, if so, we proceed to the next ray. After getting all the pixel values for the tile, we send the tile image back to the main memory using asynchronous DMA, and proceed to the next tile.

5.3.2 Techniques for filling performance gap between heterogeneous cores

The successful implementation of our streaming model critically depends on how much we can match the executions of the two stages of the model. If the PPE performs its tasks faster than the SPE, the outputs generated by the PPE should be stored somewhere so that it can proceed to execute the next task. However, much more difficult is the situation when the PPE performs its tasks slower than the SPE. In that case, the SPEs will be idle waiting for the inputs from the PPE, which can substantially degrade the overall performance and negatively impact scalability since the more the number of SPEs, the more work has to be performed by the PPE and hence the more time the SPE will have to wait. In the following, we introduce a couple of techniques for taking care of the possible performance gap between the heterogeneous cores.

We first describe a simple way to handle the case when the PPE executes its tasks faster than the SPEs handling of their corresponding tasks. We keep a small buffer for each SPE in main memory, where each entry stores a complete list of contributing ray segments for a tile. When the PPE finishes the task of creating a list of ray segments for a tile, it stores the list in the buffer and sends a message, which is actually the size of the list, to the mailbox of the SPE assigned for the tile. Then, the SPE initiates the transfer from the buffer to its local memory. The SPE keeps track of the entry from which it has to transfer data for the current tile. Since the mailbox in the SPE has 4 entries, we essentially use it as a 4-entry queue

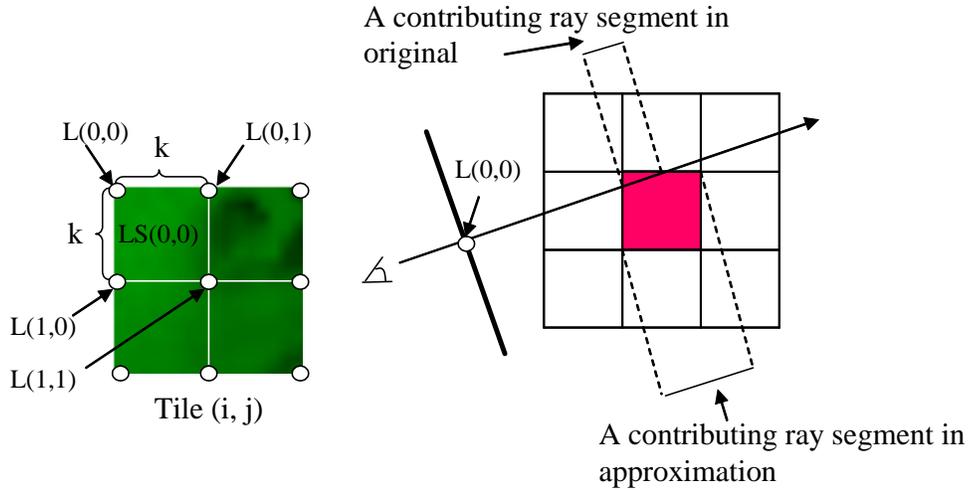
so that the messages from the PPE can be buffered and used immediately when the SPE is ready to proceed to the next tile. This scheme of using buffers on both PPE and SPE enables us to efficiently deal with the situation of overflowing inputs from PPE.

In the following subsections, we introduce our "approximation+refining" scheme to deal with the other case, in which the PPE is not fast enough to feed the SPEs. This is unfortunately the case for the current Cell B.E.

Approximation

In order to reduce the workload of the PPE, we only generate the list of contributing ray segments for every $k \times k$ -th pixel, rather than for every pixel. Each segment is now computed by projecting the boundary of an intersected octree leaf (corresponding to non-empty subvolume) onto the ray direction as shown in Figure 5.5. We estimate the contributing ray segments for each subtile by taking the union of the ray segments lists at the surrounding 4 corners. Then, the SPE assigned to the tile uses the resulting list to render to all the pixels in the subtile of size $k \times k$. This method significantly reduces the processing time in the PPE by a factor of k^2 . However, it increases the processing time in the SPE because the SPE ends up with processing much more empty voxels due to the approximate nature of the contributing ray segments used for each pixel.

In this approximation technique, we might miss some intersecting subvolumes for some pixels as shown in Figure 5.6 even though we use the projected ray segments since we selectively shoot rays to get the contributing ray segments. Missed subvolumes may lead to incorrect rendering since it can end up with reporting no



$L(a,b)$: a contributing ray segment list at (a,b) inside the tile.

List for a subtile, $LS(0,0) = L(0,0) \cup L(0,1) \cup L(1,0) \cup L(1,1)$

List for the tile, $LT(i,j) =$

a concatenation of $LS(0,0) LS(0,1) LS(1,0) LS(1,1)$

Figure 5.5: Approximation technique.

contributing ray segments for a particular subtile even though there is a non-empty subvolume, which is not traversed by any of the four rays.

Thus, we need to make sure that we never miss any subvolume for correct rendering. In orthographic ray casting, where all rays are cast in parallel, we only need to make sure the interval value k is smaller than the minimum distance between any two grid points of the leaf subvolume. In perspective ray casting, we can easily prove the following.

Proposition 1 Two rays that are k apart on the image plane, originating from the same viewpoint, never diverge more than $2k$ inside the volume as long as the distance from the viewpoint to the image plane, $dist_e$, is larger than the distance

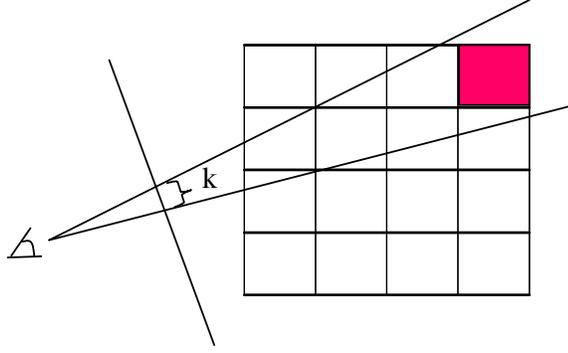


Figure 5.6: The case of missing non-empty subvolumes. In the figure, the shaded region is not checked by any of rays using the approximation technique.

from the image plane to the far end of the volume, $dist_v$. See Figure 5.7.

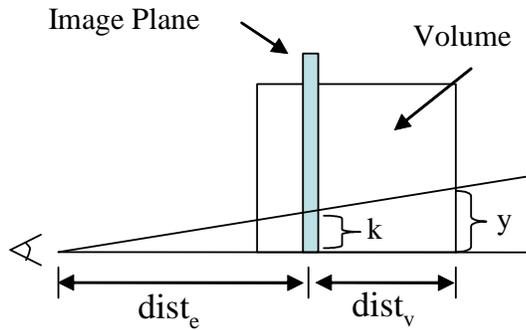


Figure 5.7: Proof of proposition 1.

Proof. First, for the case when the image plane is beyond the far end of the volume, rays are always less than k apart inside the volume. For the other case, we have a simple relationship $dist_e : k = (dist_e + dist_v) : y$, where k is the distance between the two rays on the image plane and y is the one on the far end of the volume. In order to have y less than $2k$, $dist_e$ must be larger than $dist_v$. \square

Thus, we use the approximation technique by setting the k value to half of the minimum distance between any two grid points of the leaf subvolume. Then,

we can guarantee that we can safely zoom out up to 1/2 and zoom in to infinity since the eye distance to the image plane is always larger than the distance from the image plane to the far end of the volume. This guarantee is acceptable in volume rendering since we are not interested in investigating objects in smaller size.

Refining

In order to reduce the amount of additional work performed by the SPE due to the approximation technique, we send to SPEs additional information about which subvolume is empty so that SPEs can skip the processing of the sampling points that belong to empty subvolumes. We use hashing to capture this additional information as follows.

Given a tile, we keep a hash table for every $k \times k$ -th ray and record which subvolume is not empty using the following universal hashing function.

$$\text{key} = (\text{ranX} \cdot x + \text{ranY} \cdot y + \text{ranZ} \cdot z) \text{ modulo PR}$$

$$\text{hash-table}[\text{key}] = 1$$

$$\left\{ \begin{array}{l} \text{PR: prime number equal to the hash table size} \\ 0 < \text{random number ranX, ranY, ranZ} < \text{PR} \\ \text{x,y,z: the smallest coordinates of the subvolume} \end{array} \right.$$

Then, we approximate the hash table for a subtile by taking the union of the hash tables at the 4 surrounding corners and send it to a corresponding SPE. The SPE skips a sampling point if it belongs to an empty subvolume by checking the hash table. Note that by using the hash table, we might have the case where an empty subvolume is recognized as non-empty, but will never have the opposite case. Also, by setting the hash table size large enough, we can significantly reduce the

false alarm rate.

5.4 Experimental Results

Dataset	Size	Characteristic
Foot	256^3 (16MB)	small empty space + moderate opaque interior
Aneurism	256^3 (16MB)	moderate empty space + moderate opaque interior
Engine	$256^2 \times 128$ (8MB)	small empty space + small opaque interior
Fuel	256^3 (16MB)	large empty space + small opaque interior

Table 5.1: Test Datasets. (Fuel dataset size is originally 64^3 . We enlarged it for better comparison.)

To evaluate the performance of our streaming model based methods, we selected four volumetric datasets that are widely used in the literature: two from the medical domain (foot and aneurism) and two from the science/engineering domain (fuel and engine) [Dat]. Table 5.1 summarizes the characteristics of the corresponding datasets.

All default rendering modes are semi-transparent and default rendering image size is 256^2 . All experimental results were obtained by averaging the results from 24 randomly selected view points. We chose 16×16 for tile size and 8 for the k-value by experiments. We used one Cell B.E. 3.2GHz throughout the evaluation. Figure 5.8 shows rendered images obtained using our method.

We first demonstrate that our streaming model with the "approximation+refining"

scheme removes the overhead of traversing the octree structure for empty space skipping by almost fully overlapping it with the actual rendering process. Figure 5.9 shows the processing time for the PPE and the SPEs for three different combinations of the techniques using the four datasets. Processing time on the PPE is the time it takes to traverse the octree data structure and to generate the contributing ray segments. The SPE time is the time it takes to perform the actual rendering. When none of the techniques is used, we end up starving the SPEs due to the long processing time on the PPE. When only the approximation technique is used, we significantly reduce the processing time on the PPE, but end up with increased SPE time. Finally, when the approximation technique is used in combination with the refining technique, we achieve the best results. Figure 5.9 also shows that the current implementation can scale up to the double number of SPEs since the processing time on the PPE is allowed to double for the balance of performance between the PPE and the SPE.

Another important benefit of our streaming model is that it essentially removes the latency due to the access of volume data by making it possible to almost always prefetch the data. The first two rows of Table 5.2 compares the rendering time with and without prefetching and shows that prefetching reduces rendering time by about one half.

However, it does not show that there is no memory access latency. The SPE program is blocked until the subvolumes required for rendering the current sampling points are moved to the local memory. If prefetching hides memory latency, our rendering time should be approximately the same as the time it takes for rendering

	foot	aneurism	engine	fuel
w/o prefetch	224	210	133	76
w/ prefetch	113	99	72	38
local volume, w/o early ter- mination	161	103	85	39
w/ prefetch, w/o early ter- mination	179	112	88	41

Table 5.2: Effects of prefetching (in milliseconds).

any volumetric data stored in the local memory. The third and fourth row of Table 5.2 compares the rendering time on local volume with one with our prefetching scheme. Note that since early ray termination makes the rendering time depend on the data contents, we disabled early ray termination in those experiments. We believe that the $\sim 7\%$ increase in the results is from prefetch I/O overhead because we achieved only less than 1% better results in the same experiments with only difference in the size of data transfer, which was set to zero.

Our fine-grain task decomposition allows us to achieve very good load balance. Figure 5.10 shows that our scheme achieves near-optimal load balance with average percentage standard deviation 1.7% among the 8 SPEs of the Cell B.E.

Finally, we compare the rendering performance on the Cell B.E. 3.2GHz with that of the Intel Xeon dual processor 3GHz with SSE2. We implemented the same acceleration techniques with the same ray casting algorithm. SSE2 vector instruc-

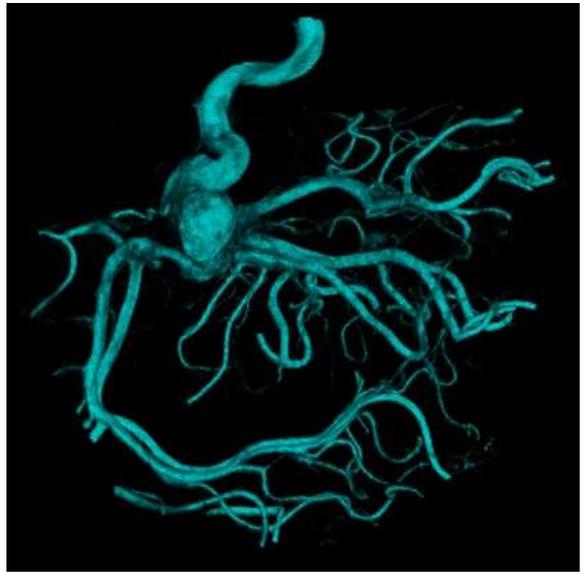
tions are used for interpolation and compositing in the same way as in the SPEs at Cell B.E. We created two threads on Intel Xeon while creating two threads on the PPE and 8 threads on the SPEs. Two threads on each of Xeon and the PPE reduce tree traversal time by dividing the traversal work. Figure 5.14 shows that our scheme for Cell B.E. consistently achieves an order of magnitude better performance.

5.5 Conclusions

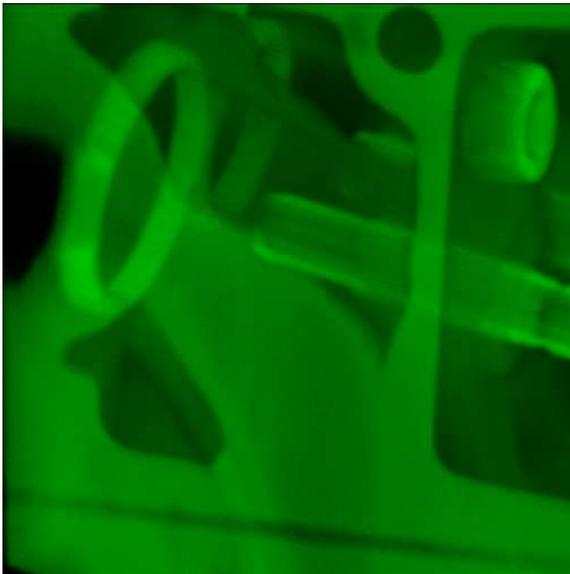
We presented a streaming model based volume ray casting, which is a new strategy for performing ray casting. This strategy enables the full utilization of empty space skipping and early ray termination, in addition to removing memory latency overheads typically encountered in ray casting due to irregular data accesses. Moreover, to successfully implement this strategy on the Cell B.E., we introduced a few additional techniques including the "approximation+ refining" technique to balance the performance gap between the two streaming stages. We have presented experimental results that illustrate the effectiveness of our new techniques.



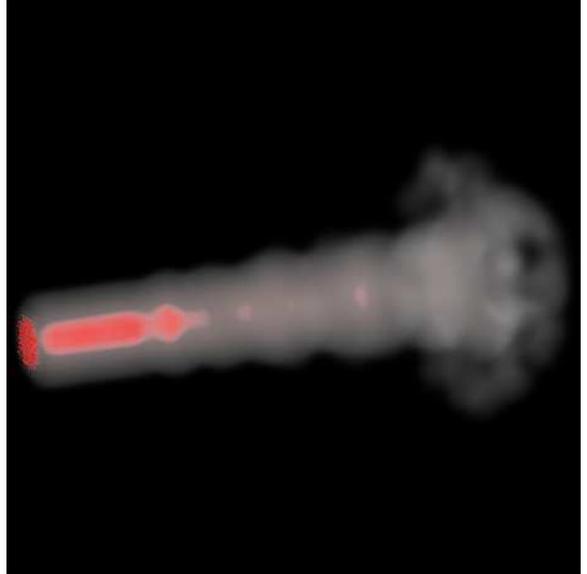
Foot



Aneurism

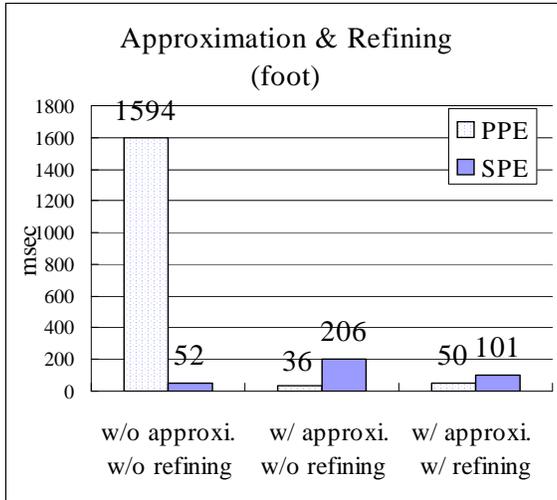


Engine

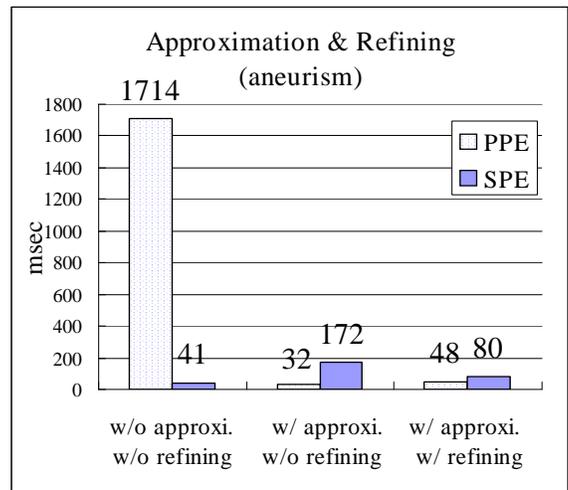


Fuel

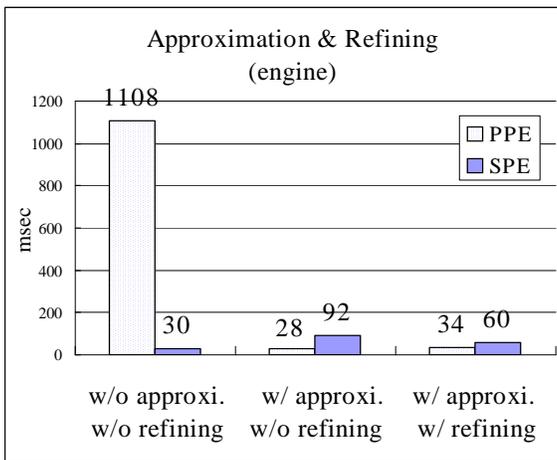
Figure 5.8: Rendered images from four datasets throughout the tests.



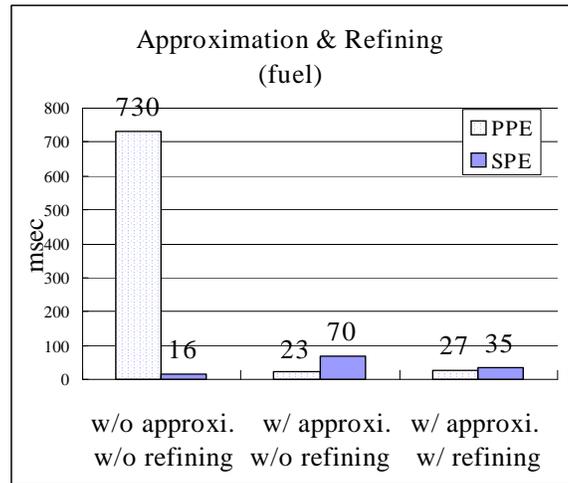
Foot



Aneurism



Engine



Fuel

Figure 5.9: Processing time in PPE and SPE for three different combinations of approximation and refining techniques.

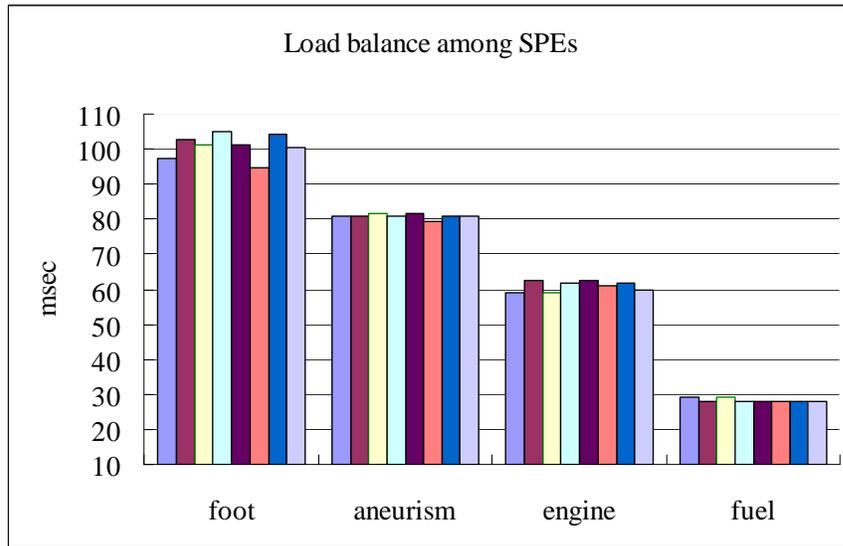


Figure 5.10: Load balance among eight SPEs.

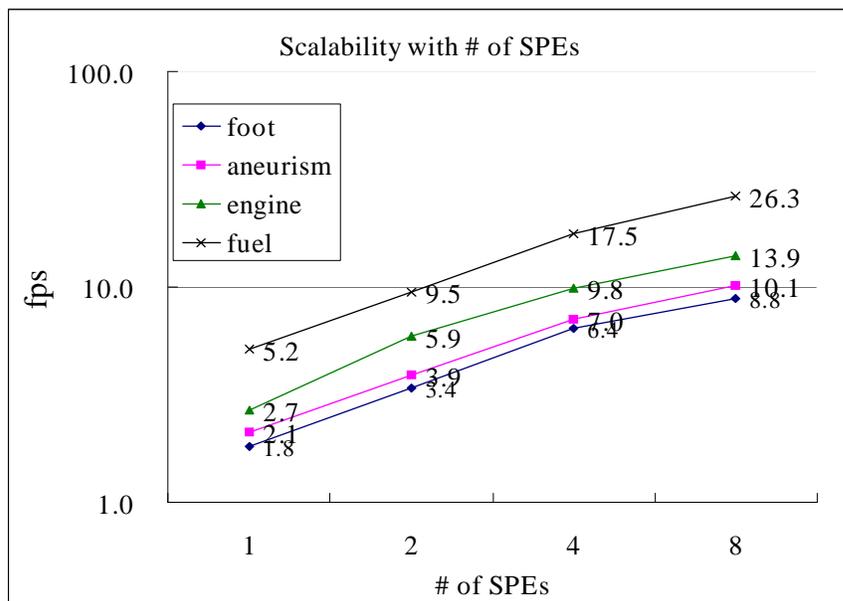


Figure 5.11: Speed up with respect to the number of SPEs.

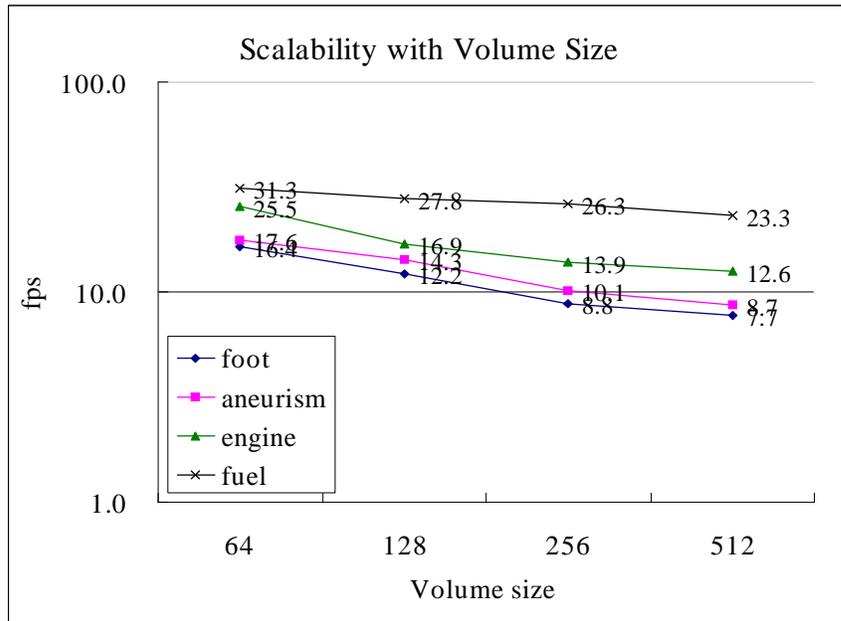


Figure 5.12: Performance with respect to the volume size.

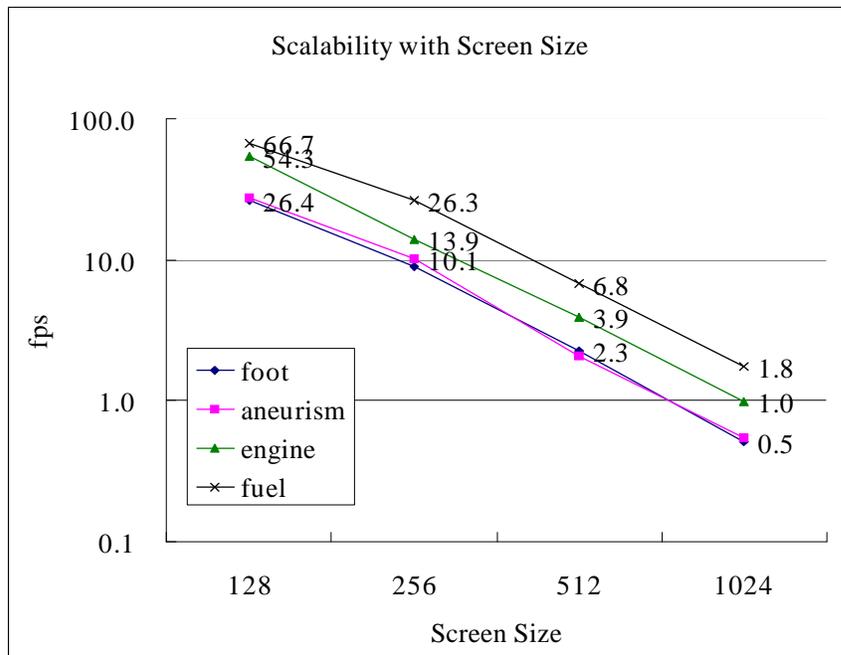


Figure 5.13: Performance with respect to the screen size.

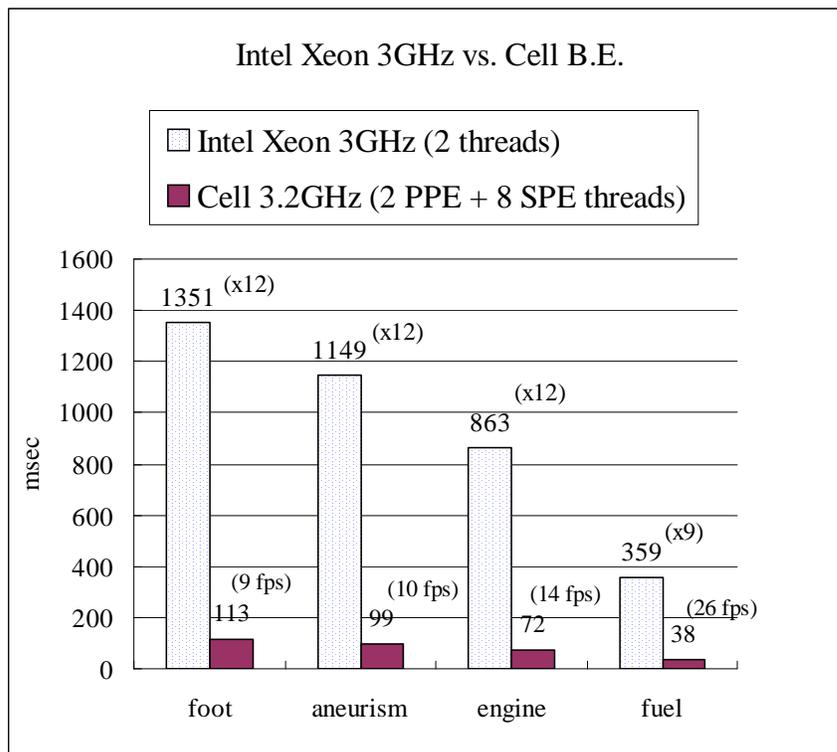


Figure 5.14: Performance comparison with Intel Xeon dual processor 3GHz with SSE2.

Chapter 6

Volume Ray Casting on CUDA

The performance of graphics processors (GPUs) is improving at a rapid rate, almost doubling every year. Such an evolution has been made possible because the GPU is specialized for highly parallel compute-intensive applications, primarily graphics rendering, and thus designed such that more transistors are devoted to computation rather than caching and branch prediction units. Due to compute-intensive applications' high arithmetic intensity (the ratio of arithmetic operations to memory operations), the memory latency can be hidden with computations instead of using caches on GPUs. In addition, since the same instructions are executed on many data elements in parallel, sophisticated flow control units such as branch prediction units in CPUs are not required on GPUs as much.

Although the performance of 3D graphics rendering achieved by dedicating graphics hardware to it far exceeds the performance achievable from just using CPU, graphics programmers had up to now to give up programmability in exchange for speed. They were limited to using a fixed set of graphics operations. On the other hand, instead of using GPUs, images for films and videos are rendered using an off-line rendering system that uses general purpose CPUs to render a frame in hours because the general purpose CPUs give graphics programmers a lot of flexibility to

create rich effects. The generality and flexibility of CPUs are what the GPU has been missing until very recently.

In order to reduce the gap, graphics hardware designers have continuously introduced more programmability through several generations of GPUs. Up until 2000, no programmability was supported in GPUs. However, in 2001, vertex-level programmability started to appear, and in 2002, pixel-level programmability also started being provided on GPUs such as NVIDIA's GeForce FX family and ATI's Radeon 9700 series. This level of programmability allows programmers to have considerably more configurability by making it possible to specify a sequence of instructions for processing both vertex and fragment processors.

However, accessing the computational power of GPUs for non-graphics applications or global illumination rendering such as ray tracing often requires ingenious efforts. One reason is that GPUs could only be programmed using a graphics API such as OpenGL, which imposes a significant overhead to the non-graphics applications. Programmers had to express their algorithms in terms of the inadequate APIs, which required sometimes heroic efforts to make an efficient use of the GPU. Another reason is the limited writing capability of the GPU. The GPU program could gather data element from any part of memory, but could not scatter data to arbitrary locations, which removes lots of the programming flexibility available on the CPU.

In order to overcome the above limitation, NVIDIA has developed a new hardware and software architecture, called CUDA (Compute Unified Device Architecture), for issuing and managing computations on the GPU as a data-parallel com-

puting device that does not require mapping instructions to a graphics API [NVI07]. CUDA provides the general memory access feature, and thus, the GPU program is now allowed to read from and write to any location in memory on CUDA.

In order to harness the power of the CUDA architecture, we need new design strategies and techniques that fully utilize the new features of the architecture. CUDA is basically tailored for data-parallel computations and thus is not well suited for other types of computations. Moreover, the current version of CUDA requires programmers to understand the specific architecture details in order to achieve the desired performance gains. Programs written without the careful attention to the architecture details are very likely to perform poorly.

In this chapter, we explore the application of our streaming model, which was introduced in the previous chapter for the Cell processor, for the CUDA architecture. Since the model is designed for heterogeneous compute resource environment, it is also well suited for the CPU and CUDA combined environment. Our basic strategy in the streaming model is the same as in the case of Cell processor. We assign the work list generation to the first stage (CPU) and actual rendering work to the second stage (CUDA) with data movement streamlined through the two stages. The key is that we carefully match the performances of the two stages so that two processes are completely overlapped and no stage has to wait for the input from the other stage.

Our scheme features the following. First, we essentially remove the overhead caused by traversing the hierarchical data structure by overlapping the empty space skipping process with the actual rendering process. Second, our algorithms are

carefully tailored to take into account the CUDA architecture's unique details such as the concept of warp and local shared memory to achieve high performance. Last, the ray casting performance is 1.5 times better than that of the Cell processor with only a third lines of codes of the Cell processor and 15 times better than that of Intel Xeon processor.

6.1 The CUDA Architecture Overview

The CUDA (Compute Unified Device Architecture) hardware model has a set of SIMD multiprocessors as shown in Figure 6.1. Each multiprocessor has a small local shared memory, constant cache, texture cache and a set of processors. At any given clock, every processor in the multiprocessor executes the same instruction. For example, NVIDIA Geforce 8800GTX architecture is comprised of 16 multiprocessors. Each multiprocessor has 8 streaming processors for a total of 128 processors.

Figure 6.2 shows the CUDA programming model. CUDA allows programmers to use C-language to program it instead of graphics APIs such as OpenGL and Direct3D. In CUDA, the GPU is a compute device that can execute a very high number of threads concurrently. The batch of threads is organized as a grid of thread blocks as shown in the Figure 6.2. A thread block is a group of threads that can synchronize and efficiently share data through the local shared memory. One or more thread blocks are dispatched to each multiprocessor and executed using time sharing. Blocks are further organized into a grid. However, threads in different blocks can not communicate and synchronize with each other. In fact,

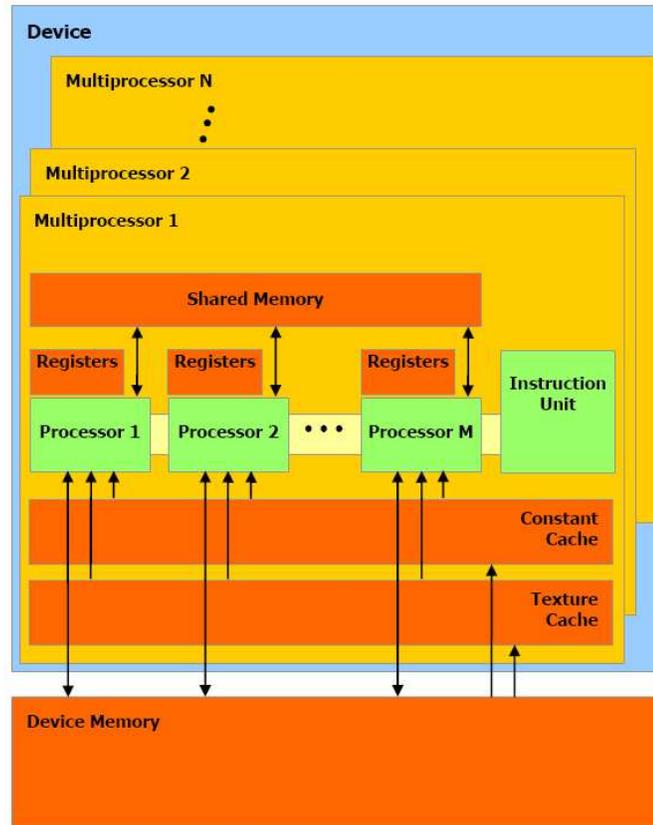


Figure 6.1: CUDA Hardware Architecture [NVI07].

synchronization mechanism is provided only to the threads in the same block, and thus, the correctness of any other communication attempts is not guaranteed because there is no mechanism that can determine the order of the threads executions in the case. This block independence makes CUDA scalable architecture because we can process more blocks in parallel as we add more processing units although it reduces programming flexibility.

As the memory and register file in a multiprocessor are shared by one or more blocks of a large number of threads, there is a limit in how many threads and blocks can be launched, depending on how much resources each thread and block requires.

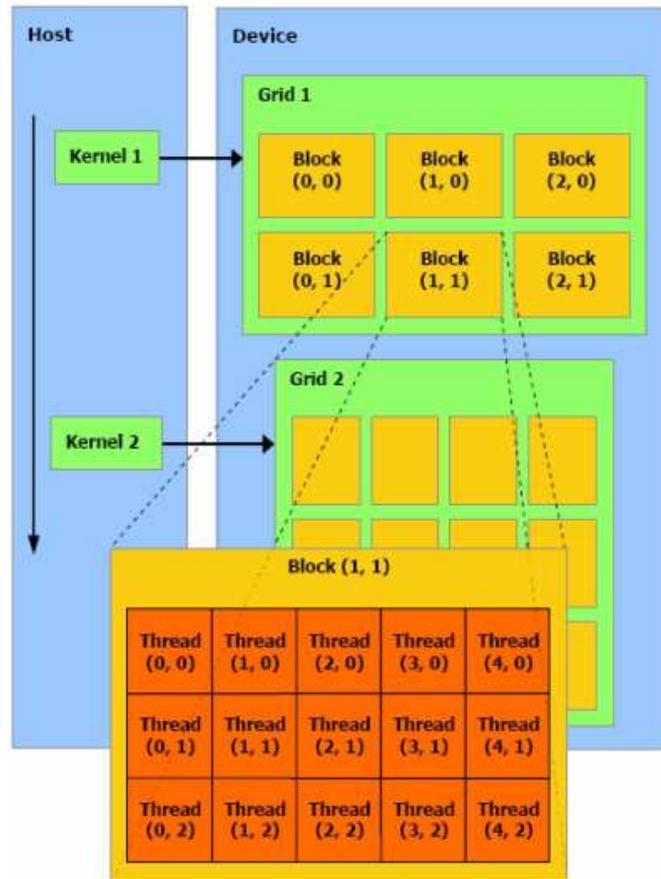


Figure 6.2: CUDA Programming Model [NVI07].

It is important to optimize the resource usage per thread so that more threads and blocks can be launched because as more threads get available there is a better chance that memory latency can be hidden.

It is also important to efficiently use memory hierarchy of CUDA to achieve high performance. The shared memory in each multiprocessor provides more than two orders of magnitude faster access to data than what the device memory does, therefore it is important to utilize the shared memory. The best way is to pre-load data that is frequently accessed in the program onto the shared memory before it is used.

Another important aspect of the current version of CUDA is the concept of *warps*. A warp is a SIMD group of threads, which constitute the unit of threads that a thread scheduler for a multiprocessor periodically switches to maximize the computational unit usage. If a warp of threads can not progress any more for some reason, then the scheduler replaces the current one with another warp that was waiting in the threads pool. A block of threads is typically comprised of a few warps. If the threads in a warp execute different instructions or their memory accesses cause bank conflicts, then the execution of the threads in the warp will be serialized, which will cause significant performance degradation. Thus, it is very important to take the concept of warp into consideration when programming for CDUA so that we can fully take advantage of the simultaneous computations of the multiprocessor.

6.2 CELL v.s. CUDA

Table 6.1 compares the two different architectures, Cell processor and CUDA, with a traditional CPU architecture in several categories. The main feature of Cell processor is that it provides more general parallel programming models than CUDA, making it a better choice for more general applications. For example, CUDA can not implement a streaming model on the chip, where a group of threads produce data and another group of threads consume the data for a certain processing at one kernel launch, while Cell can support that streaming programming model. However, CUDA provides much easier parallel programming model than Cell. For example, our

ported code of the core volume rendering function for the Cell processor has more than 3 times as many lines as that of CUDA.

	Cell B.E.	CUDA	CPU
Programming model	SPMD, MPMD	SPMD	SPSD
Simultaneous Threads	tens	Thousands	1
Programmability	Difficult	Medium	Easy
Handling Memory Latency	Pre-fetching and Double Buffering	Multithreading	Cache
Feature	Various Parallel Programming Model	Easier Parallel Programming	General Purpose
Limitation	Explicit data movement by programmers	Limited Programming Model	Low Performance

Table 6.1: Comparison of three different architectures.

In the context of volume rendering, besides the programmability and performance difference, another main difference of the two parallel architectures is that Cell processor provides more scalable support to large volume rendering because it uses main memory as a primary data storage. On the other hand, CUDA has to move data from main memory to graphics device memory which is usually smaller than main memory and because the data communication bandwidth is usually an order of magnitude slower than graphics memory bandwidth, it loses significant

amount of performance once it begins communicating with main memory during run time.

6.3 Primary Work Decomposition and Allocation

In this section, we describe our primary work decomposition and allocation scheme for volume ray casting on CUDA.

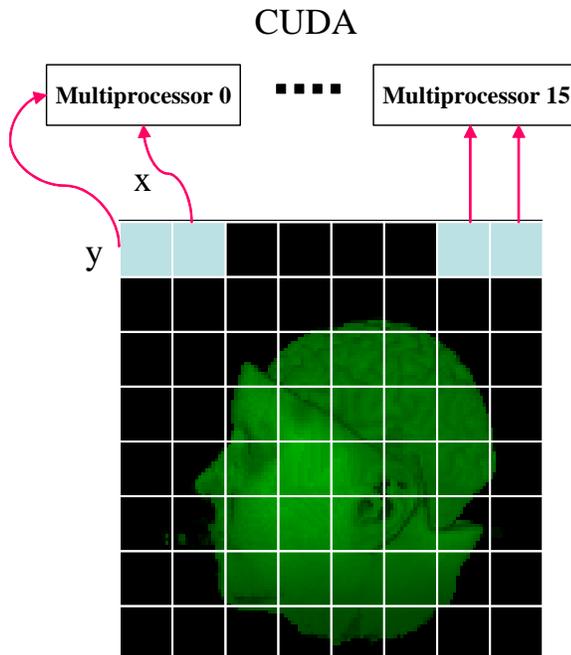


Figure 6.3: Work decomposition and assignment on CUDA. A tile consists of x by y block of threads and is dispatched into one of the multiprocessors.

Our work decomposition scheme is based on fine-grain task parallelism that achieves load balancing among the multiprocessors. In ray casting, the overall concurrency is obvious since we can compute each pixel value on the screen independently of all the other pixels. To take advantage of this fact, we divide the screen

into a grid of small tiles as we did in the case of Cell processor. A block of threads equal to the number of pixels on each tile will be allocated for the tile and the block of threads will be executed by a multiprocessor, independently of other blocks of threads.

However, there are several significant details that are different than the case of Cell processor. First, the maximum size of the tile is determined by how much resource each thread requires. Since the register file and shared memory are shared by one or more blocks of threads, we can only launch as many threads as the resource allows. Second, the dimensions of the tile are carefully selected considering the concept of warp. Since the threads in a warp should share the work list to achieve high performance, we design the dimensions of the tile such that a warp of threads occupy a rectangular region with as equal dimensions as possible. In our implementation, we use a tile of 4x32 dimension with a 4x4 subtile sharing the work list. Last, the assignment of each tile to a multiprocessor is done by the CUDA scheduler while we had to assign the tasks to the cores of the Cell processor.

6.4 Implementation of the Streaming Model

In this section, we describe the implementation of our streaming model from the previous chapter on CUDA architecture. As in the case of Cell processor, we assign two optimization techniques, empty space skipping and early ray termination, to an appropriate hardware, and streamline the data movement between the stages in the model. Efficiently implementing these two acceleration techniques is very

important since it significantly affects the ray casting performance.

6.4.1 Stage 1: Work List Generation

A general purpose processor is clearly a better candidate for efficiently traversing a hierarchical data structure. Furthermore, CUDA would have a substantial overhead in handling empty space skipping due to the concept of warp, in which a group threads, 32 in the current version of CUDA, have to execute the same instruction at any given clock cycle for high performance.

The procedure for generating work lists is the same as in the case of Cell processor. Given a ray, a CPU traverses the hierarchical data structure along the ray direction and collects contributing ray segments traversing non-empty subvolumes. Each ray segment is characterized by the ray offset from the viewpoint and the length of the corresponding segment. The collected ray segments for all the pixels of a tile are concatenated and transferred to CUDA.

We also employ the approximation technique used for Cell processor. However, for CUDA, there is another reason for using this technique. Due to the concept of warp, it is better for a group of threads to share the work lists than each thread in the same warp to run independently. Therefore, we only generate the list of contributing ray segments for every $k \times k$ -th pixel, rather than for every pixel. For example, our tile (thread block) dimensions are 4×32 and we choose every 4×4 -th pixel for the work list generation. The region (16 threads) surrounded by the 4 chosen pixels is half warp size, and we estimate the contributing ray segments for

the region by taking the union of the ray segments lists at the surrounding 4 corners. Then, CUDA uses the resulting list to render to all the pixels in the region of size $k \times k$. Note that the current version of CUDA has a shared memory organized into 16 banks and thus it is recommended that at least a half warp of threads executes the same instruction.

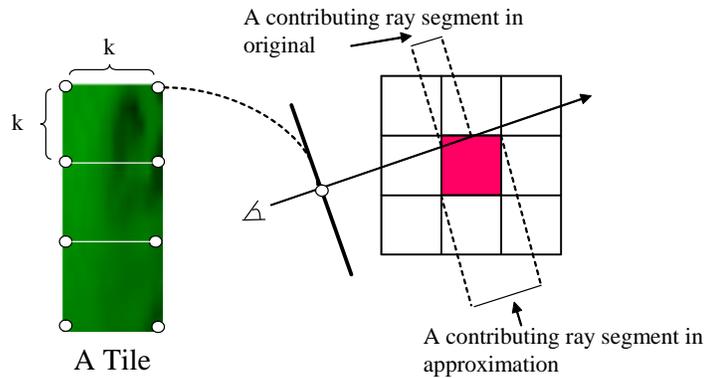


Figure 6.4: Approximation technique on CUDA.

The main difference of implementing the streaming model from the case of the Cell processor is the method used to stream the data. While we have multiple channels from the first stage (a PPE) to the second stage (SPEs) on the Cell processor because each SPE runs independently, we have only one channel to CUDA because CUDA does not allow the independent access to each multiprocessor. Therefore, we need a large streaming unit to move to all the multiprocessors at one kernel launch as illustrated in Figure 6.5. In our implementation, our streaming unit is 32 tiles (blocks), which will allocate 2 blocks of threads to each multiprocessor on the Geforce 8800GTX with 16 multiprocessors. After launching 32 tiles of work on CUDA, the CPU starts getting the contributing lists for the next 32 tiles and wait

until the previous launch is completed.

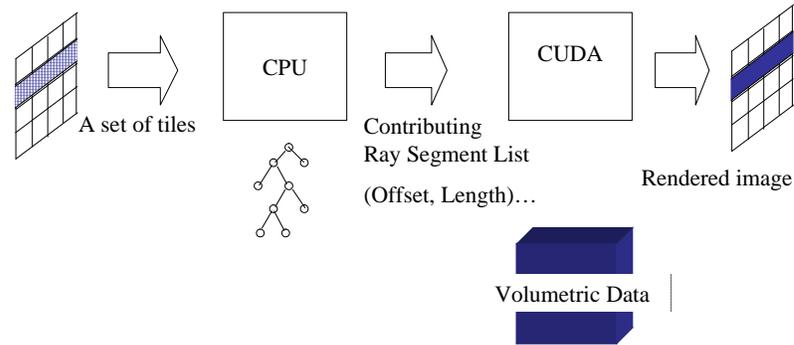


Figure 6.5: The streaming model for CUDA. Note that the streaming unit is a set of tiles compared to one tile in the case of Cell processor.

6.4.2 Stage 2: Rendering

CUDA is ideal for the actual rendering work since it was designed for compute-intensive parallel workloads. Thus, we naturally implement rendering and early ray termination on CUDA. Before the rendering starts, we pre-load all the work lists for the current tile into the shared memory since the shared memory provides data with the latency of L-1 cache (1~2 cycles). The other procedures are the same as before. We perform reconstruction, shading, classification, and finally compositing on the sample points along all the contributing ray segments. The final image is transferred back into main memory after a final kernel launch is finished.

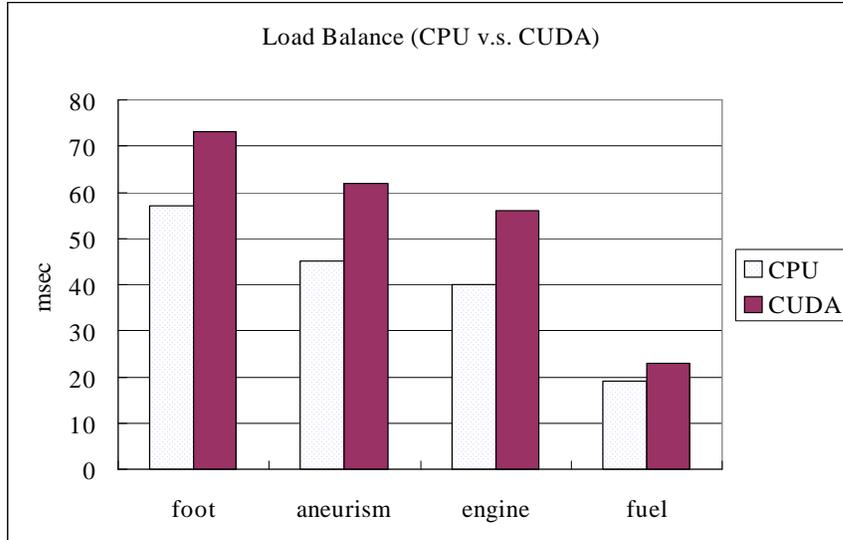


Figure 6.6: Load balance between CPU and CUDA.

6.5 Experimental Results

To evaluate the performance of our streaming model based implementation, we used the same four volumetric datasets and rendering mode used in the case of Cell processor. Please refer to Table 5.1 for the characteristics of the datasets. We used Geforce 8800GTX with a ver 1.0 CUDA drivers with Intel core 2 duo processor throughout the evaluation.

We first demonstrate that our streaming model implemented on the CUDA environment removes the overhead of traversing the octree structure for empty space skipping by fully overlapping it with the actual rendering process. Figure 6.6 shows the processing time for CPU and CUDA on the four datasets. Processing time on the CPU is the time it takes to traverse the octree data structure to generate the contributing ray segments. The CUDA time is the time it takes to perform the actual rendering. This figure shows that the empty space skipping time is completely

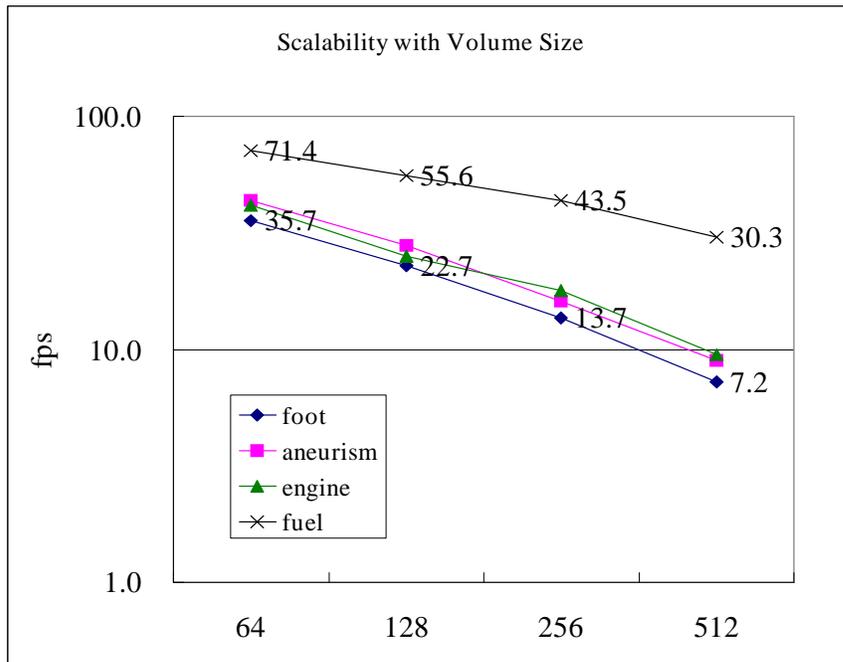


Figure 6.7: Performance with respect to the volume size.

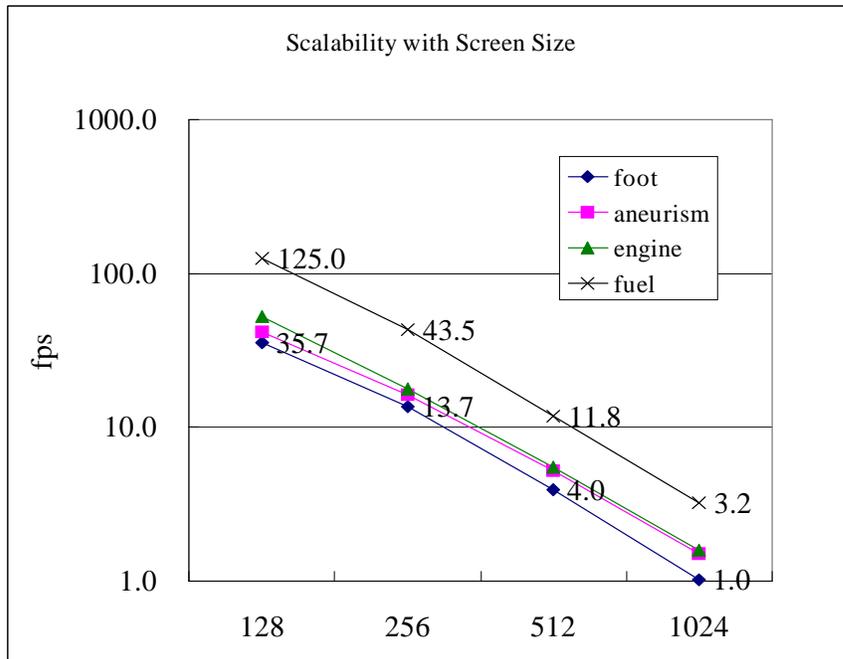


Figure 6.8: Performance with respect to the screen size.

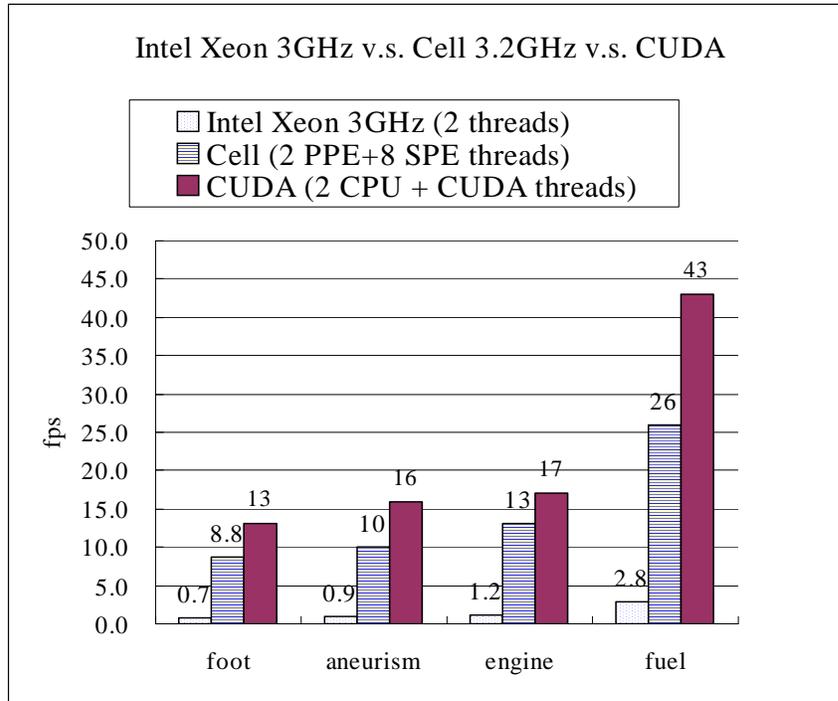


Figure 6.9: Performance comparison (CPU v.s. Cell v.s. CUDA).

hidden.

Figure 6.7 and 6.8 show that the performance of our implementation with respect to input volumetric size and output screen size.

We compare the rendering performance on CUDA with the Cell B.E. 3.2GHz and also Intel Xeon dual processor 3GHz with SSE2. Figure 6.9 shows that the performance on CUDA consistently achieves 15 times better performance than that of Intel and 1.5 times better than that of Cell processor. Also, it is very likely that it will produce even better performance once its 3-D texture unit is exposed in the later version of CUDA because we can utilize the texture cache unit in each multiprocessor.

This results show that the new multi-core/many-core architectures can handle

compute and communication intensive applications such as volume ray casting in much more efficient way since in particular, the Xeon processor and the Cell processor that we have used for the experiments do not have much difference in the number of transistors (286 million and 234 million, respectively) and operate at about the same frequency (3GHz and 3.2GHz, respectively).

6.6 Conclusions

In this chapter, we explored the application of our streaming model, which was introduced in the previous chapter for Cell processor, for the CUDA architecture. Our scheme fully utilizes the heterogeneous compute resource environment by using both task parallelism (simultaneous processing of the optimization techniques on different types of cores) and data parallelism (rendering by thousands of threads).

CUDA provides about 1.5 times better performance than Cell processor while the CUDA program has only a third of parallel code lines of that of Cell processor in our implementation. However, aside from other factors such as the number of transistors and price, CUDA has a scalability problem with data set size because it can only efficiently render a data set which can fit in graphics memory, while the Cell processor can handle as large data as main memory allows.

The improvements in GPU performance and flexibility are likely to continue in the future and will allow programmers to write increasingly diverse and sophisticated programs that take advantage of the capabilities of the GPUs. There are emerging efforts that combine CPU and GPU into a single chip. However, we will need

efficient algorithmic strategies to make use of the available heterogeneous compute resources.

Chapter 7

Conclusion

In this dissertation, we have explored how to efficiently render large volumetric data sets. In [Dub05], Intel recognized that recognition, mining, and synthesis applications will be future workloads, which will push computers into the era of Teraflops. They are all compute intensive, highly parallel applications. Volume rendering, as a compute intensive and highly parallel synthesis application, represents a future workload on computers. The methods presented here are efforts to maximize what the current computer system can offer for this future workload. Based on our two most critical observations on our current computer systems (wider gap between computation and communication performance, and trend toward heterogeneous parallel compute resources), this dissertation has made the following contributions to the field.

- We have shown how to layout data on a disk to efficiently perform an out-of-core axis-aligned slicing of large multidimensional scalar fields. We have analytically and through experimental results shown that our scheme provides faster processing time and requires less cache memory than the typical Z-order scheme for any type of axis-aligned out-of-core slicing queries at every k -th value ($k > 1$), without any data replication. Our scheme is currently the

best known scheme for this application.

- We have shown how to efficiently build an out-of-core indexing structure for an arbitrary n-dimensional volumetric data. Our scheme uses the entropy theoretic notion to maximize the indexing structure efficiency and is particularly useful when dealing with time-series volumetric data. We have shown that our scheme achieves an order of magnitude higher indexing structure efficiency than the best previous known indexing structures.
- We have presented a streaming model to efficiently implement volume rendering on a heterogeneous compute resource environment. We have shown how to efficiently implement two main optimization techniques on a heterogeneous multi-core chip. We have particularly shown how to implement the model on Cell Broadband Engine. Through the experimental results, we have shown that our model essentially removes the optimization overhead occurred in previous sequential models and moreover it enables us to essentially remove memory access latency through prefetching.
- We have shown that our streaming model can be applied to other heterogeneous compute resource environment. We have extended the streaming model for volume rendering used on Cell processor to the PC environment where CPU and GPU (particularly CUDA) are combined into a single system. We have shown that our model gives the same benefits as in the Cell processor. Also, we have compared the strengths and weaknesses of three processor architectures (Intel Xeon processor, Sony/Toshiba/IBM Cell Broadband Engine,

and NVIDIA CUDA architecture) and reported with the performance results on each architecture for a number of widely used benchmarks.

Parallel processors such as multi-cores and GPUs have certainly become a mainstream computing environment. Currently, what is missing is efficient programming models for those heterogeneous, parallel processors. Developing efficient parallel programming models is very important to make those parallel processors successfully accepted. An efficient programming model makes it easier to harness the strength of a particular hardware without having to write low level code tightly coupled to the architecture. It lightens the burden of programmers throughout the whole software development process including coding, debugging, and maintenance. It should abstract the hardware so that programmers do not have to consider low architecture details which may change in later versions of the hardware and possibly support cross-platform portability. However, unfortunately, programmers currently must know low architecture details to get any good performance on those parallel processors and portability is almost never achievable. Our work on the Cell and CUDA has aimed at providing a high-level programming model that can facilitate the exposure of the strengths of those parallel architectures for a specific application. We believe that our work provides insights for developing better programming models for wider applications.

BIBLIOGRAPHY

- [AAW00] K. Anagnostou, T.J. Atherton, and A.E. Waterfall. 4D volume rendering with the Shear Warp factorisation. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 129–137. ACM Press New York, NY, USA, 2000.
- [AMC05] H. Akiba, K.L. Ma, and J. Clyne. End-to-end data reduction and hardware accelerated rendering techniques for visualizing time-varying non-uniform grid volume data. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 31–225, 2005.
- [AR05] J. Allard and B. Raffin. A Shader-Based Parallel Rendering Framework. In *Visualization, IEEE 2005*, pages 17–17, 2005.
- [BBFZ00] W. Blanke, C. Bajaj, D. Fussell, and X. Zhang. The metabuffer: A scalable multiresolution multidisplay 3-d graphics system using commodity rendering engines. *Tr2000-16, University of Texas at Austin, February, 2000*.
- [BCF03] A.P.D. Binotto, J.L.D. Comba, and C.M.D. Freitas. Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 10, 2003.
- [BHPB03] E.W. Bethel, G. Humphreys, B. Paul, and J.D. Brederson. Sort-First, Distributed Memory Parallel Visualization and Rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. IEEE Computer Society Washington, DC, USA, 2003.
- [BPRS98] C. L. Bajaj, V. Pascucci, G. Rabbio, and D.R. Schikore. Hypervolume visualization: a challenge in simplicity. In *Volume Visualization, IEEE Symposium on*, pages 95–102, 1998.
- [BPS96] C.L. Bajaj, V. Pascucci, and D.R. Schikore. Fast isocontouring for improved interactivity. In *Proceedings of the IEEE symposium on Volume visualization*, pages 39–46, 1996.
- [BPTZ99] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the IEEE symposium on Parallel visualization and graphics*, pages 97–104, 1999.
- [BSS02] C. Bajaj, A. Shamir, and B.S. Sohn. Progressive Tracking of Isosurfaces in Time-Varying Scalar Fields. *CS & TICAM Technical Report, University of Texas at Austin, 2002*.

- [BWC04] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):130–141, Mar 2004.
- [BWSF06] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the Cell Processor. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, pages 15–23, 2006.
- [CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, pages 91–98. ACM Press, 1994.
- [CD99] J. Clyne and J. Dennis. Interactive direct volume rendering of time-varying data. *Data Visualization*, 99:109–120, 1999.
- [CE97] M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proceedings of the 8th conference on Visualization*, 1997.
- [CFSW01] Y.J. Chiang, R. Farias, C.T. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proceedings of the IEEE symposium on parallel and large-data visualization and graphics*, pages 59–66, 2001.
- [Chi03] Y.J. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proceedings of IEEE Visualization*, pages 29–36, 2003.
- [Cly03] J. Clyne. The Multiresolution Toolkit: Progressive Access for Regular Gridded Data. In *Proceedings of Visualization, Imaging, and Image Processing*, pages 152–157, 2003.
- [CMM⁺97] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, Apr 1997.
- [CN94] T.J. Cullip and U. Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. 1994.
- [CS97] Y.J. Chiang and C.T. Silva. I/o optimal isosurface extraction. In *Proceedings of IEEE Visualization*, pages 293–300, 1997.
- [CSS98] Y.J. Chiang, C.T. Silva, and W.J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of IEEE Visualization*, pages 167–174, 1998.
- [CT91] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. John Wiley, 1991.

- [Dat] Volume datasets repository, <http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/datasets.html>.
- [Dub05] P. Dubey. A platform 2015 workload model: Recognition, mining and synthesis moves computers to the era of tera. *Intel White Paper*, 2005.
- [ECS00] D. Ellsworth, L.J. Chiang, and H.W. Shen. Accelerating time-varying hardware volume rendering using TSP trees and color-based error metrics. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 119–128. ACM Press New York, NY, USA, 2000.
- [FS01] R. Farias and C.T. Silva. Out-of-core rendering of large unstructured grids. *IEEE Computer Graphics and Applications*, 21(4):42–50, 2001.
- [GS01] S. Guthe and W. Straßer. Real-time decompression and visualization of animated volume data. In *Proceedings of the conference on Visualization'01*, pages 349–356. IEEE Computer Society Washington, DC, USA, 2001.
- [GSDJ04] B. Gregorski, J. Senecal, M.A. Duchaineau, and K.I. Joy. Adaptive extraction of time-varying isosurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):683–694, 2004.
- [GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *Proceedings of the conference on Visualization'02*, pages 53–60. IEEE Computer Society Washington, DC, USA, 2002.
- [Hil91] D. Hilbert. Ueber stetige abbildung einer linie auf ein flachenstuck. *Mathematische Annalen*, 38:459–460, 1891.
- [HJ05] C.D. Hansen and C.R. Johnson. *The visualization handbook*. Elsevier Butterworth-Heinemann, 2005.
- [HSHH07] D.R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive kd tree GPU raytracing. pages 167–174. ACM Press, 2007.
- [IBM06] IBM. *Cell Broadband Engine Programming Tutorial verion 2.0*, 2006.
- [Jai89] A.K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, 1989.
- [JAKS05] H. P. Hofstee C. R. Johns T. R. Maeurer J. A. Kahle, M. N. Day and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, 2005.
- [JET] Time-Varying Volume Data Repository, The Five Jets dataset. <http://www.cs.ucdavis.edu/~ma/ITR/tvdr.html>.

- [KJ06] J. Kim and J. JaJa. A novel information-aware octree for the visualization of large scale time-varying data. Technical Report CS-TR-4778 and UMIACS-TR-2006-03, University of Maryland at College Park, 2006.
- [KMM⁺01] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen. Interactive texture-based volume rendering for large data sets. *Computer Graphics and Applications, IEEE*, 21(4):52–61, 2001.
- [Kru90] W. Krueger. The application of transport theory to visualization of 3D scalar data fields. In *Proceedings of the First IEEE Conference on Visualization*, pages 273–280, 1990.
- [Law00] J.K. Lawder. *The Application of Space-filling Curves to the Storage and Retrieval of Multi-Dimensional Data*. PhD thesis, University of London, 2000.
- [LC87] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, 1987.
- [Lev90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics (TOG)*, 9(3):245–261, 1990.
- [LL94] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM Press.
- [LM99] P. Leutenegger and K.L. Ma. Fast retrieval of disk resident unstructured volume data for visualization. *External Memory Algorithms and Visualization*, 50, 1999.
- [LMC01] E.B. Lum, K.L. Ma, and J. Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of the conference on Visualization'01*, pages 263–270. IEEE Computer Society Washington, DC, USA, 2001.
- [LMC02] E.B. Lum, K.L. Ma, and J. Clyne. A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):286–301, 2002.
- [LPD⁺02] L. Linsen, V. Pascucci, M.A. Duchaineau, B. Hamann, and K.I. Joy. Hierarchical Representation of Time-Varying Volume Data with "4th-root-of-2" Subdivision and Quadrilinear B-Spline Wavelets. In *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*. IEEE Computer Society Washington, DC, USA, 2002.

- [LSJ96] Y. Livnat, H.W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, Mar 1996.
- [MC00] K.L. Ma and D. Camp. High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of Supercomputing 2000 Conference*, 2000.
- [MCEF94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE*, 14(4):23–32, 1994.
- [MHB⁺00] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proceedings of the IEEE symposium on Volume visualization*, pages 81–90. ACM Press New York, NY, USA, 2000.
- [MHS99] L. Moll, A. Heirich, and M. Shand. Sepia: scalable 3D compositing using PCI Pamette. In *Field-Programmable Custom Computing Machines, Seventh Annual IEEE Symposium on*, pages 146–155, 1999.
- [MP01] K.L. Ma and S. Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Computer Graphics and Applications*, 21(4):72–83, 2001.
- [MPHK94] K.L. Ma, JS Painter, C.D. Hansen, and M.F. Krogh. Parallel volume rendering using binary-swap compositing. *Computer Graphics and Applications, IEEE*, 14(4):59–68, 1994.
- [MSE06] C. Müller, M. Strengert, and T. Ertl. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2006.
- [Nat] National Library of Medicine, National Institutes of Health, <http://www.nlm.nih.gov/research/visible/visiblehuman.html>. *The Visible Human project*.
- [Neu94] U. Neumann. Communication costs for parallel volume-rendering algorithms. *Computer Graphics and Applications, IEEE*, 14(4):49–58, 1994.
- [NM02] N. Neophytou and K. Mueller. Space-time points: 4d splatting on efficient grids. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 97–106. IEEE Press Piscataway, NJ, USA, 2002.
- [NVI07] NVIDIA. *CUDA Programming Guide 1.0*, 2007.

- [OM84] J.A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, April 1984.
- [PSL⁺98] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.P. Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization 1998*, volume 2, 1998.
- [PVH⁺03] F.H. Post, B. Vrolijk, H. Hauser, R.S. Laramée, and H. Doleisch. State of the Art Reviews The State of the Art in Flow Visualisation: Feature Extraction and Tracking. *Computer Graphics Forum*, 22(4):775, 2003.
- [Sam90] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.
- [SCESL02] C. Silva, Y.J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization Course Notes*, 2002.
- [SCM99] H.W. Shen, L.J. Chiang, and K.L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In *Proceedings of the conference on Visualization'99*, pages 371–377. IEEE Computer Society Press Los Alamitos, CA, USA, 1999.
- [SH00] P.M. Sutton and C.D. Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, Apr 2000.
- [She98] H.W. Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proceedings of IEEE Visualization*, pages 159–166, 1998.
- [SHE⁺01] G. Stoll, P. Hanrahan, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, and S. Hunt. Lightning-2: a high-performance display subsystem for PC clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 141–148. ACM Press New York, NY, USA, 2001.
- [Shi05] P. Shirley. *Fundamentals of computer graphics*. AK Peters, 2005.
- [SHLJ96] H.W. Shen, C.D. Hansen, Y. Livnat, and C.R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proceedings of the 7th conference on Visualization '96*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
- [SJ94] H.W. Shen and C.R. Johnson. Differential volume rendering: a fast volume visualization technique for flow animation. pages 180–187, 1994.

- [SJ06] Q. Shi and J. JaJa. Isosurface extraction and spatial filtering using Persistent OcTree (POT). *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 12(5):1283, 2006.
- [SM02] J. Sweeney and K. Mueller. Shear-Warp deluxe: the Shear-Warp algorithm revisited. In *Proceedings of the symposium on Data Visualisation*. Eurographics Association, 2002.
- [SML⁺03] A. Stompel, K.L. Ma, E.B. Lum, J. Ahrens, and J. Patchett. SLIC: scheduled linear image compositing for parallel volume rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 33–40, 2003.
- [SMW⁺04] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Parallel Graphics and Visualization*, 2004.
- [SMW⁺05] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, and T. Ertl. Large Volume Visualization of Compressed Time-Dependent Datasets on GPU Clusters. *Parallel Computing*, 31(2):205–219, 2005.
- [SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Proceedings of Volume Graphics*, pages 187–195, 2005.
- [SW03] J. Schneider and R. Westermann. Compression domain volume rendering. In *IEEE Visualization 2003*, pages 293–300, 2003.
- [Vit00] J.S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, March 2000.
- [WB98] C. Weigle and D.C. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *Proceedings of the IEEE symposium on Volume visualization*, pages 103–110, 1998.
- [WE98] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH*, volume 98, pages 169–178, 1998.
- [Wes90] L. Westover. Footprint evaluation for volume rendering. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376. ACM Press New York, NY, USA, 1990.
- [Wes95] R. Westermann. Compression Domain Rendering of Time-Resolved Volume Data. In *IEEE Visualization 1995 Conference Proceedings*, pages 51–58, 1995.

- [WFM⁺05] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.P. Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–573, 2005.
- [WG92] J. Wilhelms and A.V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, Jul 1992.
- [WG94] J. Wilhelms and A.V. Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 27–34, New York, NY, USA, 1994. ACM Press.
- [WGLS05] C. Wang, J. Gao, L. Li, and H.W. Shen. A Multiresolution Volume Rendering Framework for Large-Scale Time-Varying Data Visualization. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 11–223, 2005.
- [WGS04] C. Wang, J. Gao, and H.W. Shen. Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 23–30, 2004.
- [WPLM01] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on PC clusters. *Computer Graphics and Applications, IEEE*, 21(4):62–69, 2001.
- [WS03] J. Woodring and H.W. Shen. Chronovolumes: a direct rendering technique for visualizing time-varying data. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 27–34. ACM Press New York, NY, USA, 2003.
- [WSK02] M. Wan, A. Sadiq, and A. Kaufman. Fast and reliable space leaping for interactive volume rendering. In *Proceedings of the conference on Visualization '02*, pages 195–202. IEEE Computer Society, 2002.
- [WVW94] O. Wilson, A. VanGelder, and J. Wilhelms. Direct volume rendering via 3d textures. Technical report, Santa Cruz, CA, USA, 1994.
- [WWS03] J. Woodring, C. Wang, and H.W. Shen. High dimensional direct rendering of time-varying volumetric data. In *Visualization, IEEE 2003*, pages 417–424, 2003.
- [YS93] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *IEEE Visualization 1993*, pages 62–69, 1993.
- [ZBR02] X. Zhang, C. Bajaj, and V. Ramachandran. Parallel and out-of-core view-dependent isocontour visualization using random data distribution. In *Proceedings of the IEEE symposium on Parallel visualization and graphics*, pages 9–17, 2002.