

Streaming Model Based Volume Ray Casting Implementation for Cell Broadband Engine

Jusub Kim*

Joseph JaJa[†]

Institute for Advanced Computer Studies

Institute for Advanced Computer Studies

Department of Electrical and Computer Engineering

Department of Electrical and Computer Engineering

University of Maryland at College Park

University of Maryland at College Park

Abstract

Interactive high quality volume rendering is becoming increasingly more important as the amount of more complex volumetric data steadily grows. While a number of volumetric rendering techniques have been widely used, ray casting has been recognized as an effective approach for generating high quality visualization. However, for most users, the use of ray casting has been limited to datasets that are very small because of its high demands on computational power and memory bandwidth. However the recent introduction of the Cell Broadband Engine (Cell B.E.) processor, which consists of 9 heterogeneous cores designed to handle extremely demanding computations with large streams of data, provides an opportunity to put the ray casting into practical use. In this paper, we introduce an efficient parallel implementation of volume ray casting on the Cell B.E. The implementation is designed to take full advantage of the computational power and memory bandwidth of the Cell B.E. using an intricate orchestration of the ray casting computation on the available heterogeneous resources. Specifically, we introduce streaming model based schemes and techniques to efficiently implement acceleration techniques for ray casting on Cell B.E. In addition to ensuring effective SIMD utilization, our method provides two key benefits: there is no cost for empty space skipping and there is no memory bottleneck on moving volumetric data for processing. Our experimental results show that we can interactively render practical datasets on a single Cell B.E processor.

*e-mail: jusub@umd.edu

[†]e-mail: joseph@umiacs.umd.edu

1 INTRODUCTION

There is a consistent trend in almost all scientific, engineering and medical domains toward increasingly generating higher resolution volumetric data as computing power steadily increases and imaging instruments get more refined. For computational scientists, there is a constant desire to conduct more computationally demanding simulations in order to capture complex phenomena at finer scales. At the same time, biomedical researchers and practitioners are demanding higher quality visualization for medical data generated by increasingly more sophisticated imaging instruments such as CT, MRI, and 3-D confocal microscopy. Clearly, the ability to interactively visualize the volumetric data using high quality rendering techniques is critical to fully explore and understand the corresponding datasets.

Ray casting [8] has been recognized as a fundamental volume rendering technique that can produce very high quality images. However, for most of users, 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 [3, 7]. 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 [10]. 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.) [6], 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 paper, 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 [8] - 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 on 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 they 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.

Our experimental results show that we can interactively ray cast practical datasets of size 256^3 onto a

256² image at 9~26 frames/sec with one Cell B.E processor 3.2GHz, which is about an order of magnitude faster than the implementation at Intel Xeon 3GHz. Our scalability results with respect to volume size show that we can achieve interactive visualization for much larger data sets but we could not run larger experiments because the memory of our Cell B.E. was limited to 1GB.

In the following sections, we start by discussing related work and briefly introducing Cell B.E. architecture. We then explain our primary work decomposition and assignment scheme followed by a description of the techniques to deal with the speed gap between the heterogeneous cores. We end with a brief summary of the experimental results and a conclusion.

2 RELATED WORK

Levoy [8] proposed two optimization techniques for ray casting - empty space skipping and early ray termination, which are the most widely used optimization techniques for ray casting. He used a pyramid of binary volumes to make rays efficiently skip empty space and also made each ray terminate early if the opacity value accumulates to a level where the color stabilizes. Yagel and Shi [16] proposed another optimization technique using frame-to-frame coherency. Their method saves the coordinates of the first non-empty voxel encountered by each ray so that rays can start from these coordinates in the next frame. Their method was improved by Wan et al. [15] in several ways.

As volume rendering is computationally quite demanding especially for large datasets or high resolution screens, there have been many efforts to accelerate this method using the latest hardware and/or a cluster of computers. Recently, the most popular method seems to be the one that is based on using the graphics cards' texture mapping capability [3], which can also be extended to multiple cards and a cluster system. Kniss et al. [7] distribute a data set into multiple graphics cards in a shared-memory system, and make each card render a subvolume by using the texture mapping technique. Lum et al. [9] and Strengert et al. [14] 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. [13] show that ray casting can be implemented on the programmable graphics processor and Müller et al. [11] extend the hardware accelerated ray casting technique to a cluster system. On the other hand, it is worth mentioning the recent hardware-based ray tracing techniques for geometric rendering since the

principle of ray shooting is the same. Horn et al. [4] develop k-D tree based GPU ray tracing methods and Benthin et al. [2] introduce ray tracing techniques for Cell B.E. For isosurface rendering on the Cell B.E., we refer the reader to O’Conor et al. [12].

3 CELL BROADBAND ENGINE OVERVIEW

The Cell Broadband Engine (Cell B.E.) [6], as shown in Figure 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.

The 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 the SPEs. It is also possible for the 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.

4 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 2.

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

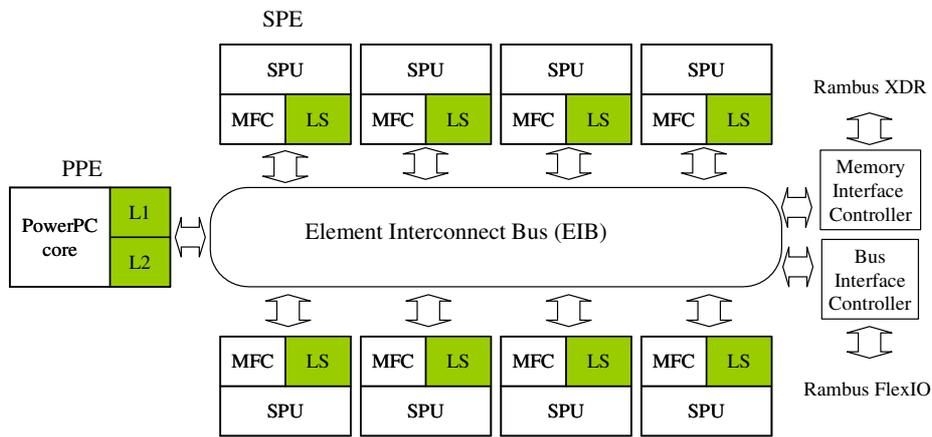


Figure 1: Cell Broadband Engine Overview [5].

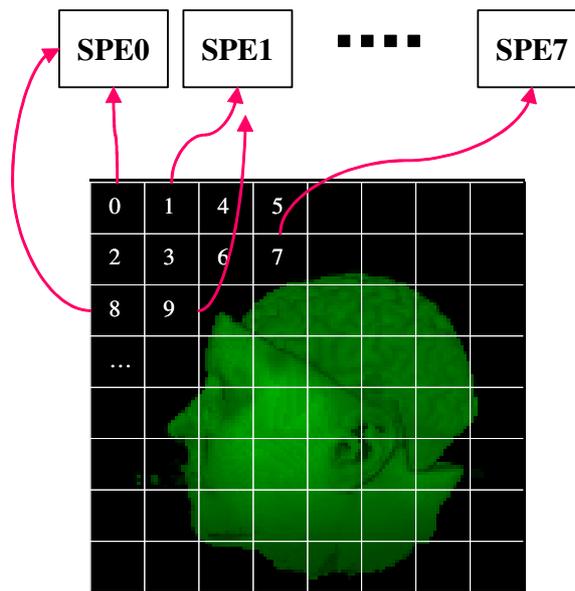


Figure 2: Work decomposition and assignment to the SPEs.

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 perfor-

mance 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 parallelism 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 2, which shows a 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 IMPLEMENTATION OF ACCELERATION TECHNIQUES

There are two most widely used acceleration techniques for ray casting [8]: 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.

5.1 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 3.

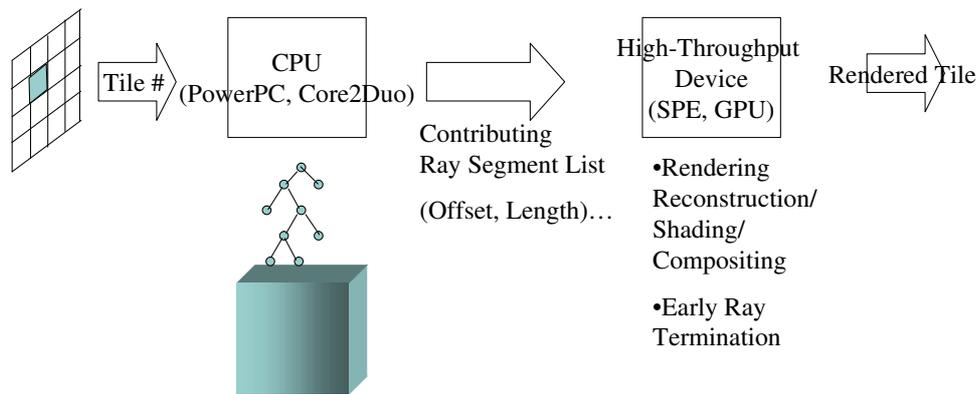


Figure 3: Our streaming model for acceleration techniques.

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.

The 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

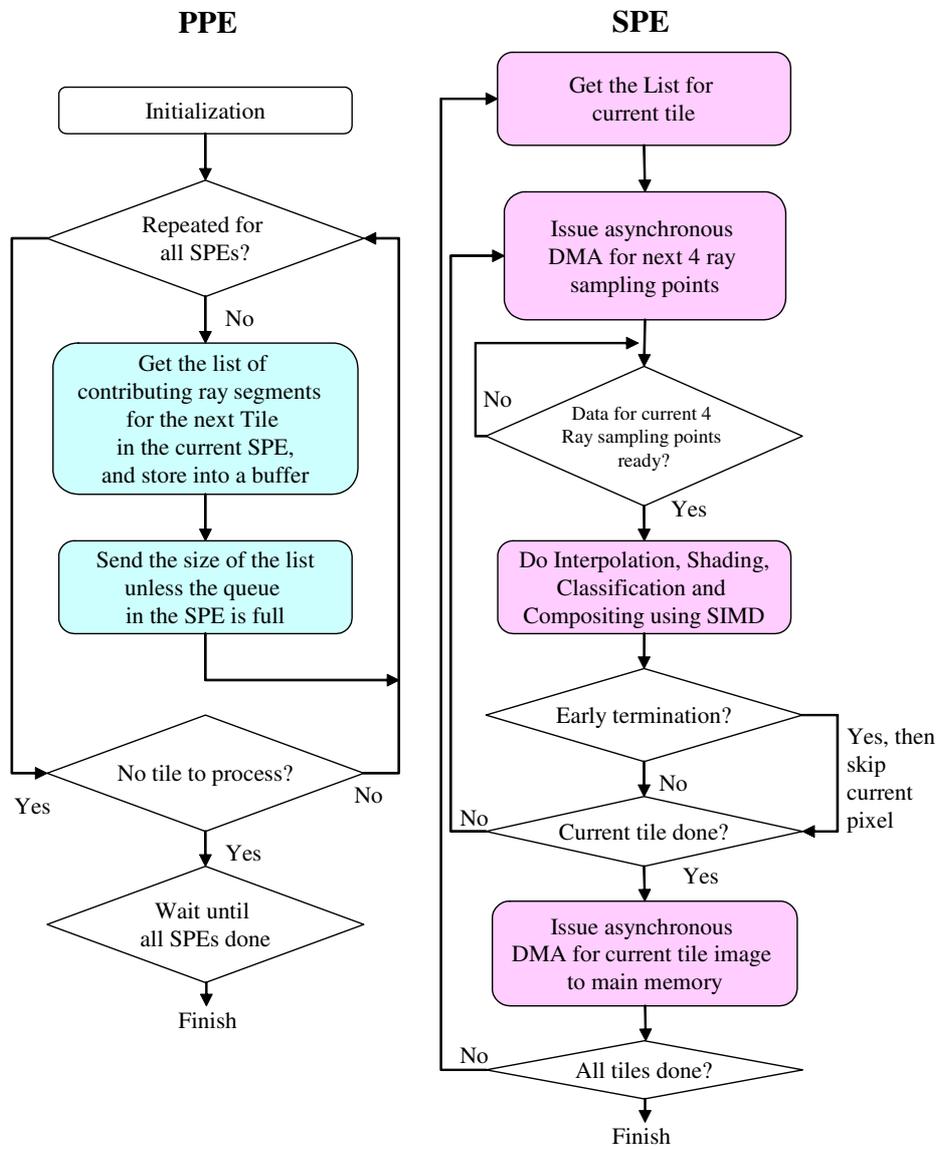


Figure 4: Main algorithms in PPE and SPE.

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 [5]. 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.

5.2 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. As seen in Figure 5, 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

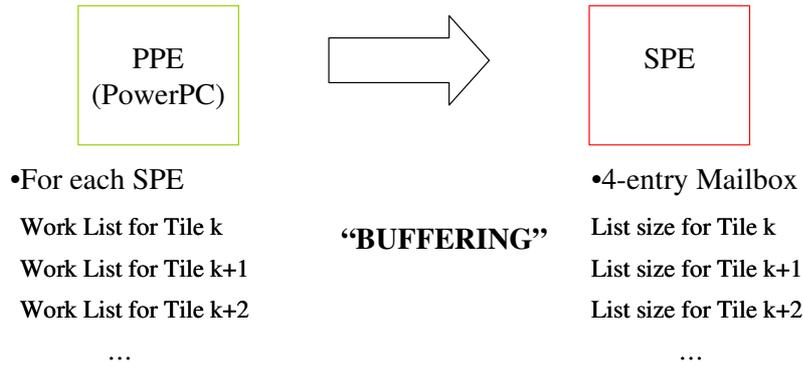


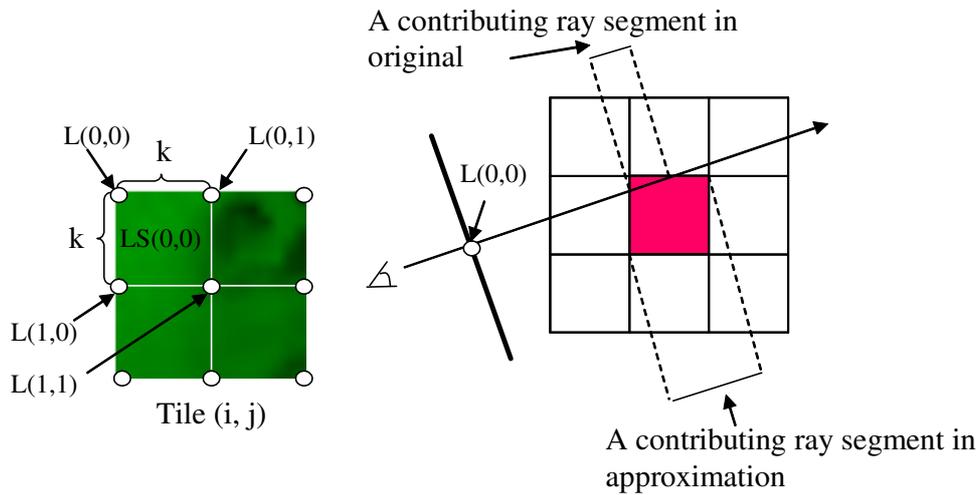
Figure 5: Our buffering scheme in order to handle the case when the PPE executes its tasks faster than the SPEs.

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 6. 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.



$L(a,b)$: 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 6: Approximation technique.

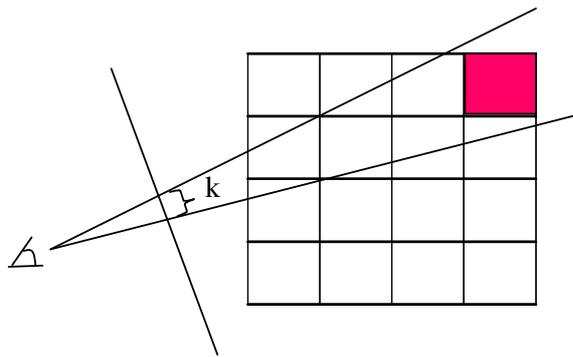


Figure 7: The case of missing non-empty subvolumes. In the figure, the shaded region is not checked by any of rays using the approximation technique.

In this approximation technique, we might miss some intersecting subvolumes for some pixels as shown in Figure 7 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 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 from the image plane to the far end of the volume, $dist_v$. See Figure 8.

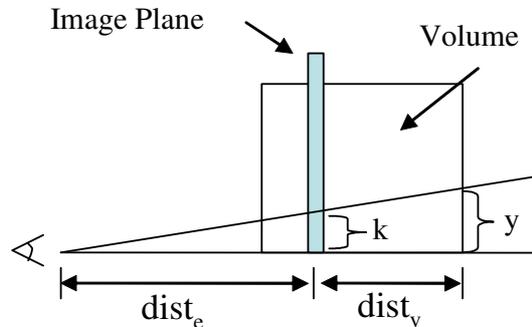


Figure 8: Proof of proposition 1.

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 the SPEs additional information about which subvolume is empty so that the 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 as seen in Figure 9.

•Every $k \times k$ -th ray

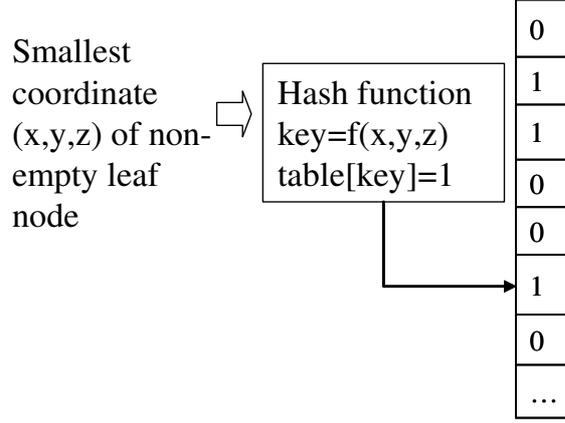


Figure 9: Our hashing scheme in order to handle the case when the PPE executes its tasks slower than the SPEs. The hash table is sent to the corresponding SPE. A '1' indicates a non-empty subvolume.

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

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

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

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.

6 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 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) [1]. Table 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 used by experiments. We used one Cell B.E. 3.2GHz throughout the evaluation. Figure 10 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 11 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 11 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 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 any volumetric data stored in the local memory. The third and fourth rows of Table 2

	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 termination	179	112	88	41

Table 2: Effects of prefetching (in milliseconds).

compare the rendering time on local volume with or without 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 12 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. The SSE2 vector instructions are used for interpolation and compositing in the same way as in the SPEs of the 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 16 shows that our scheme for Cell B.E. consistently achieves an order of magnitude better performance.

The results show that the new multi-core architecture can handle compute and communication intensive applications such as volume ray casting in much more efficient way since the particular 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).

7 CONCLUSION

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. In addition to successfully implementing 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.

8 ACKNOWLEDGEMENT

We would like to thank Professor David Bader for generously allowing us to use Cell Processors at the STI Cell Center of Competence at Georgia Tech for this research.

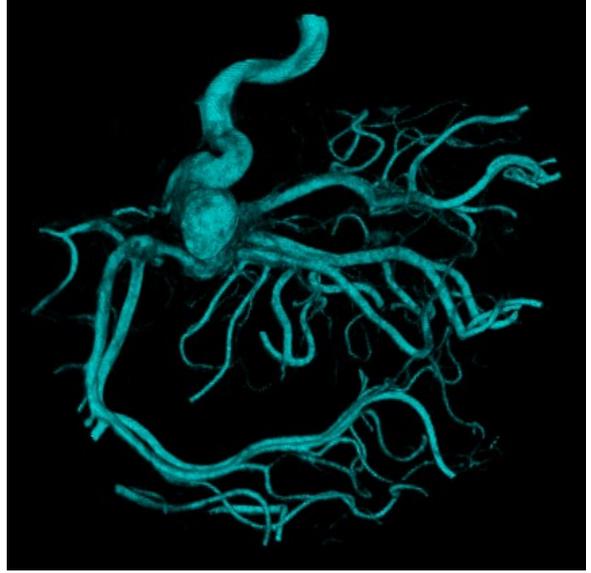
REFERENCES

- [1] Volume datasets repository, <http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/datasets.html>.
- [2] 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.
- [3] Brian Cabral, Nancy Cam, and Jim 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.
- [4] D.R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive kd tree GPU raytracing. pages 167–174. ACM Press, 2007.
- [5] IBM. *Cell Broadband Engine Programming Tutorial verion 2.0*, 2006.
- [6] 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.
- [7] 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.
- [8] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics (TOG)*, 9(3):245–261, 1990.
- [9] 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.
- [10] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proceedings of IEEE symposium on Volume visualization*, pages 81–90. ACM Press, 2000.
- [11] 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.
- [12] K. OConor, Carol OSullivan, and Steven Collins. Isosurface extraction on the cell processor. In *Seventh Irish Workshop on Computer Graphics*, 2006.

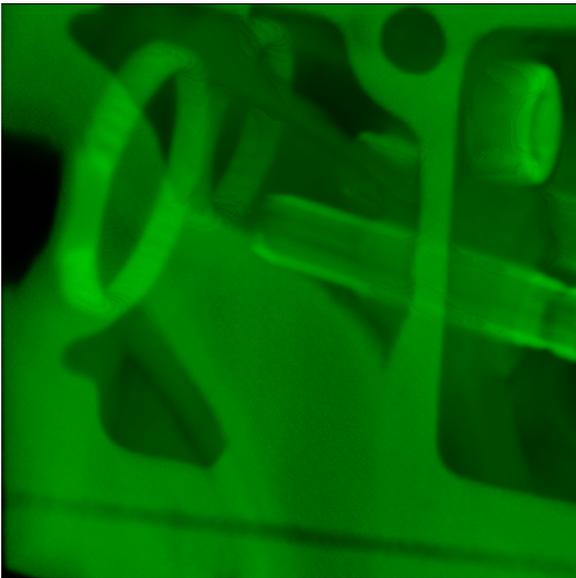
- [13] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Proceedings of Volume Graphics*, pages 187–195, 2005.
- [14] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Parallel Graphics and Visualization*, 2004.
- [15] Ming Wan, Aamir Sadiq, and Arie 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.
- [16] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *IEEE Visualization 1993*, pages 62–69, 1993.



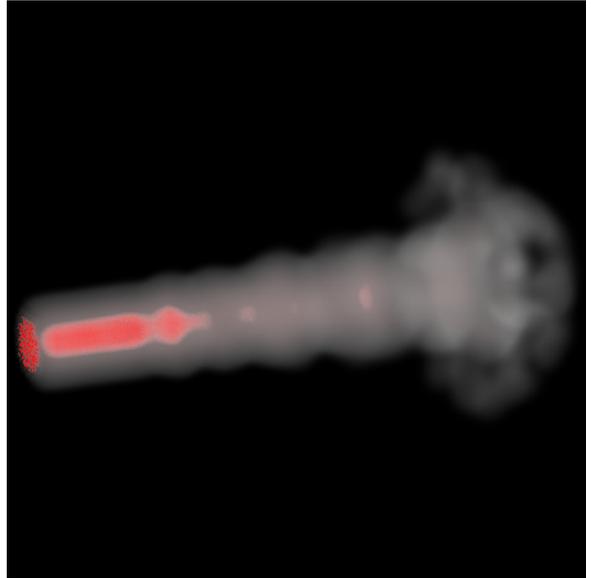
Foot



Aneurism

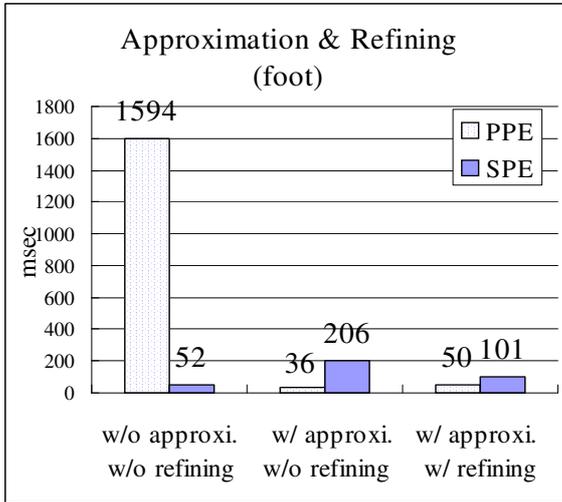


Engine

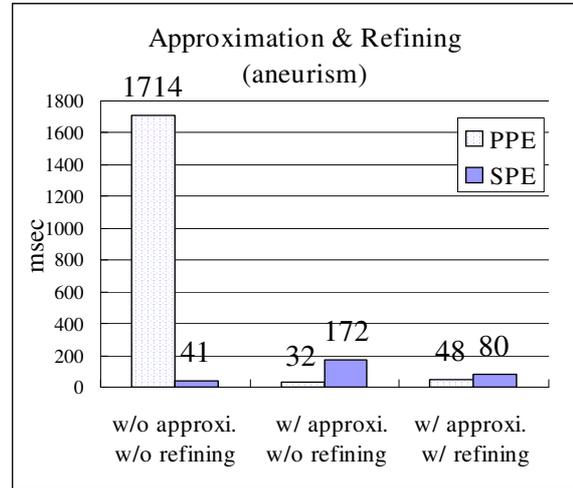


Fuel

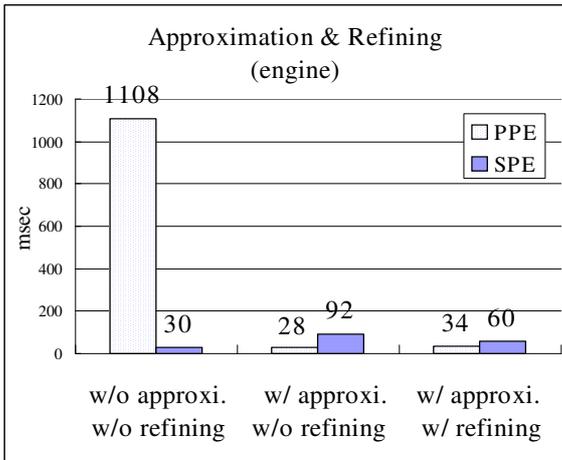
Figure 10: Rendered images from four datasets throughout the tests.



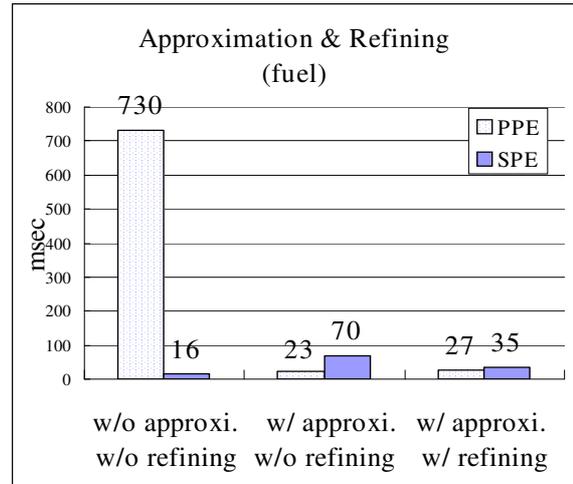
Foot



Aneurism



Engine



Fuel

Figure 11: Processing time in PPE and SPE for three different combinations of approximation and refining techniques.

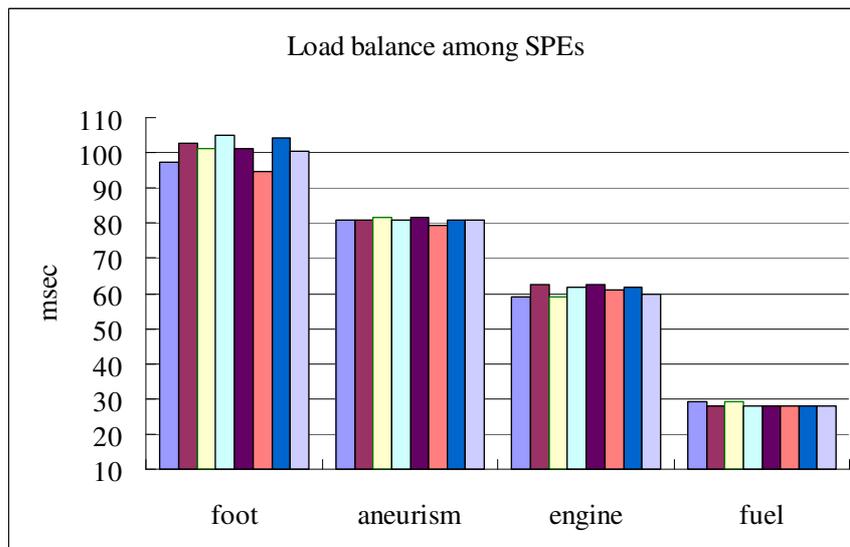


Figure 12: Load balance among eight SPEs.

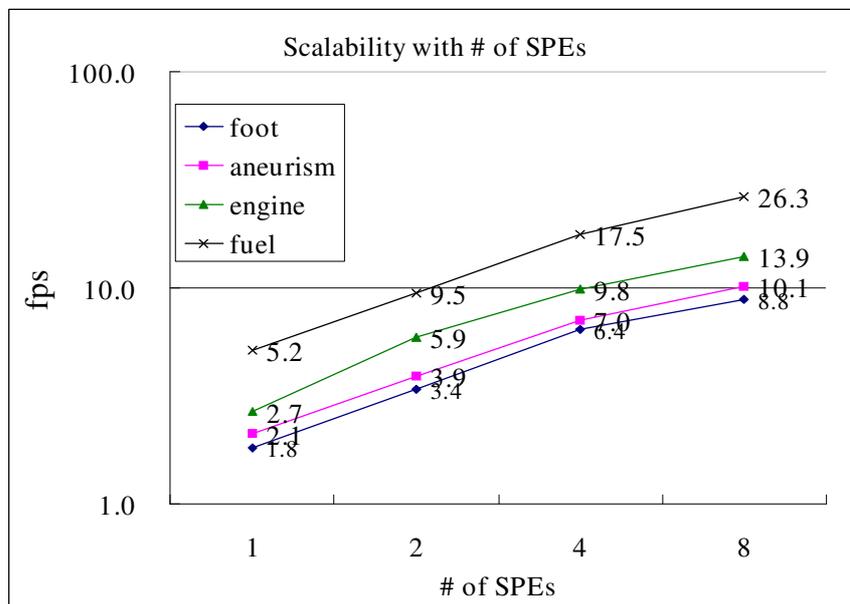


Figure 13: Speed up with respect to the number of SPEs.

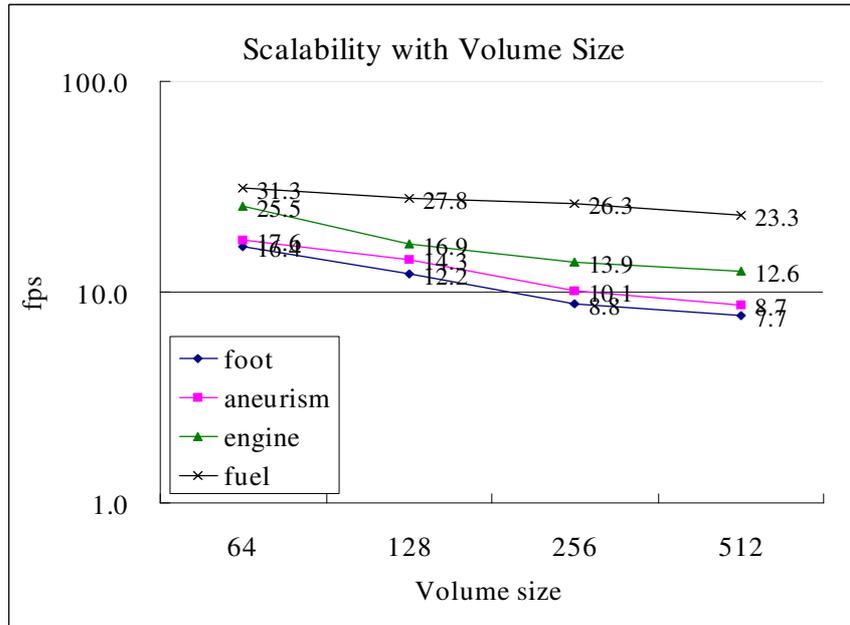


Figure 14: Performance with respect to the volume size.

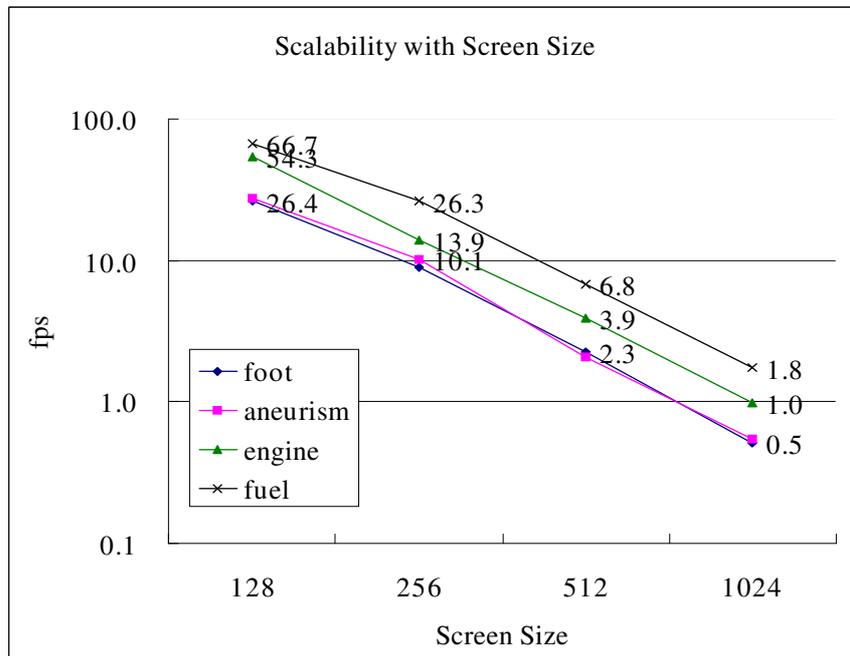


Figure 15: Performance with respect to the screen size.

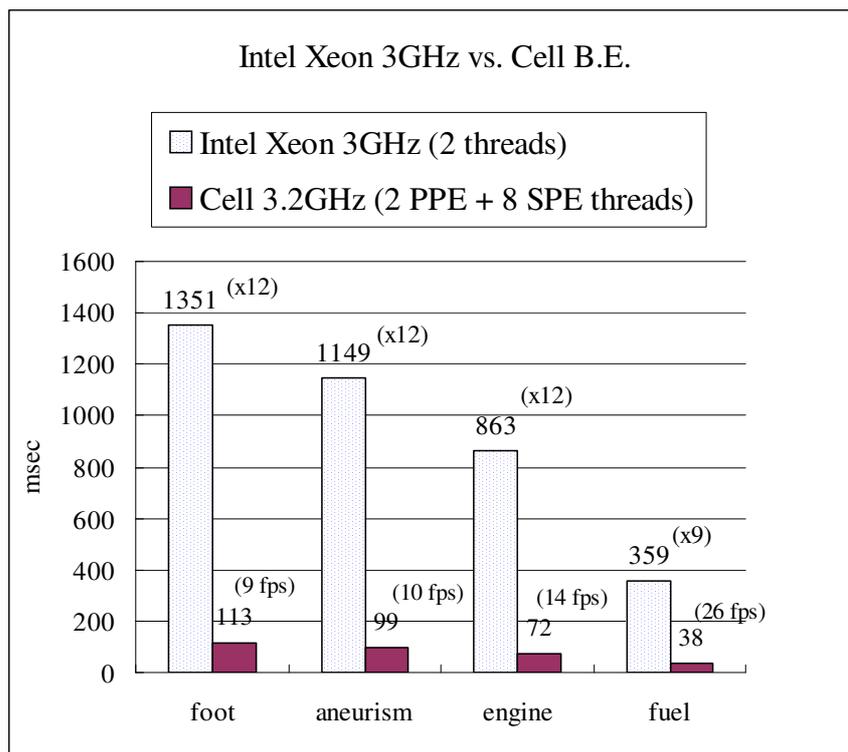


Figure 16: Performance comparison with Intel Xeon dual processor 3GHz with SSE2.