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 the 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,

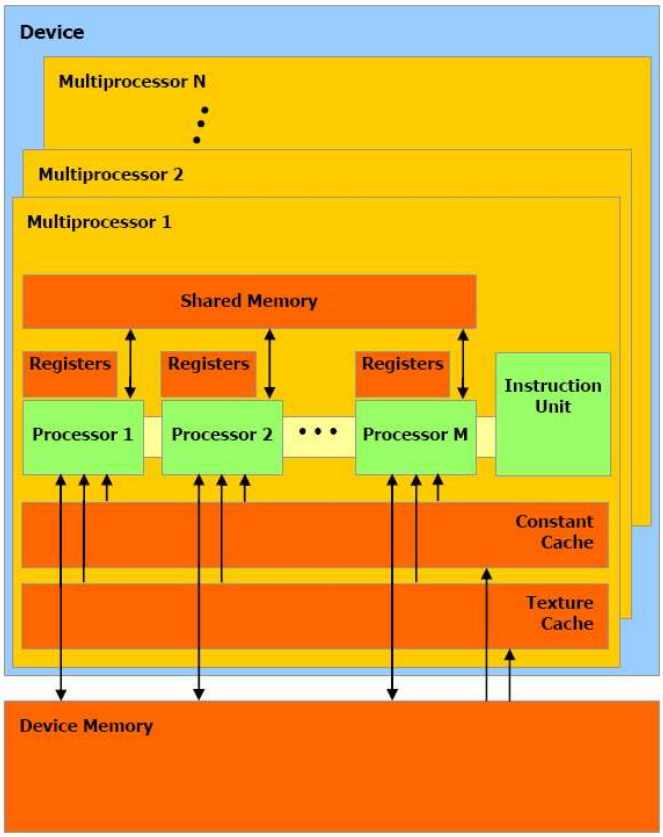

Figure 6.1: CUDA Hardware Architecture [NVI07].

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.

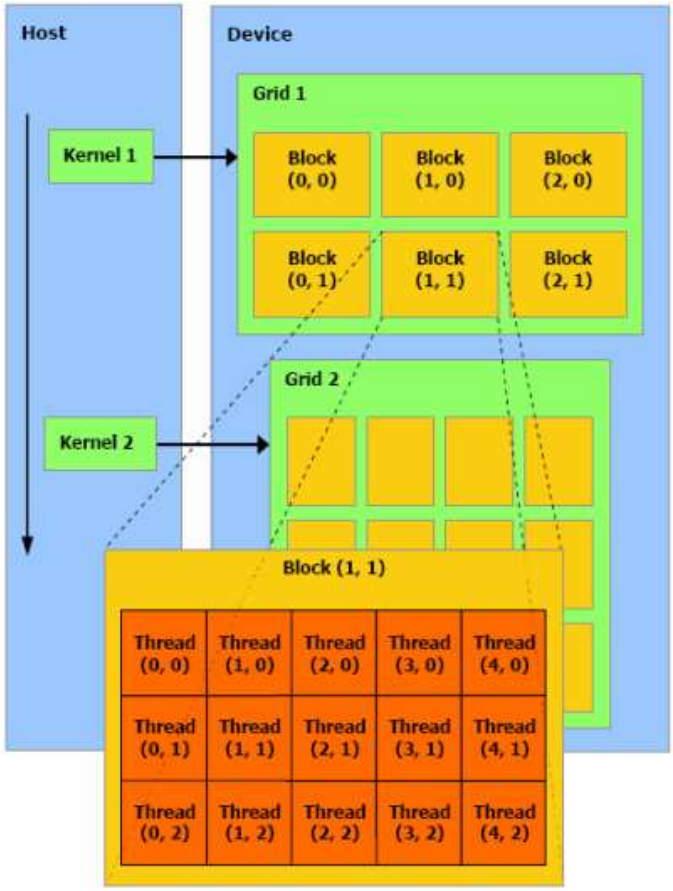

Figure 6.2: CUDA Programming Model [NVI07].

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 programing 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.

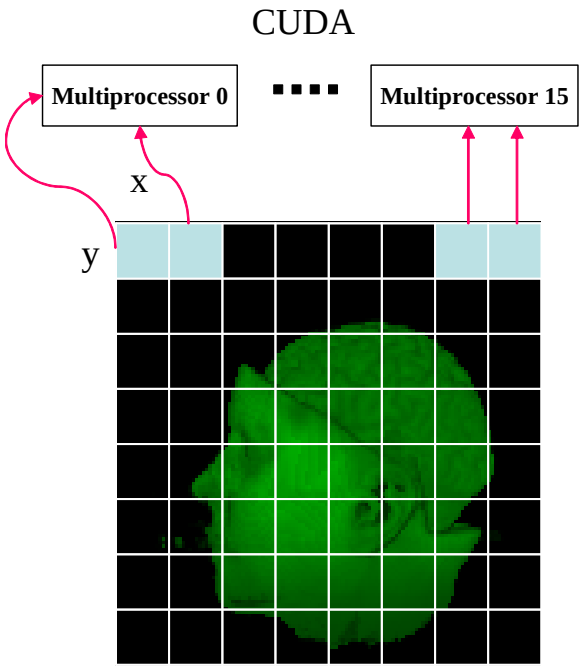

Figure 6.3: Work decomposition and assignment on CUDA. A tile consists of x by y block of threads and is dispatched into one of the multiprocessors.

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.

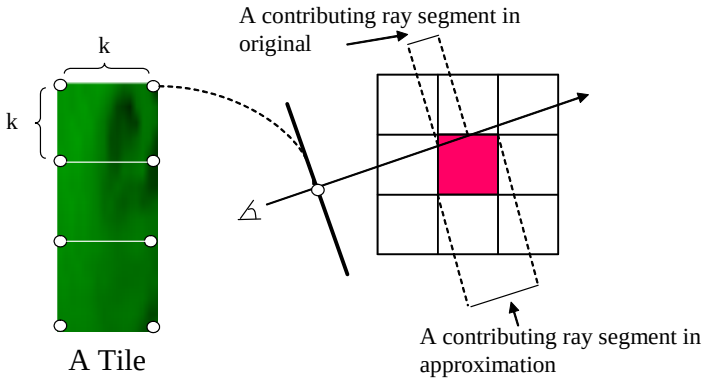

Figure 6.4: Approximation technique on CUDA.

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.

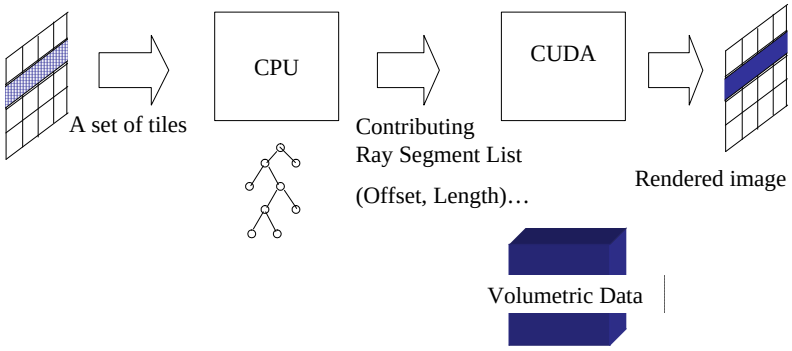

Figure 6.5: The streaming model for CUDA. Note that the streaming unit is a set of tiless compared to one tile in the case of Cell processor.

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.

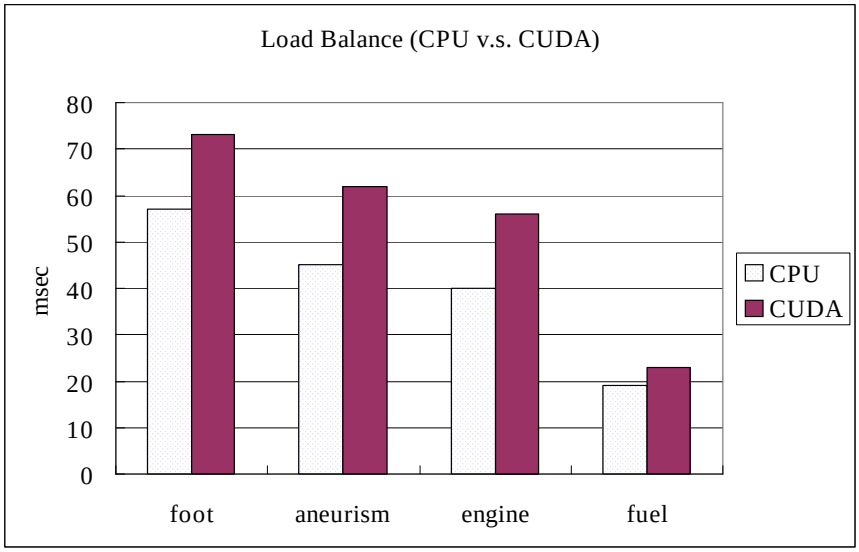

Figure 6.6: Load balance between CPU and CUDA.

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

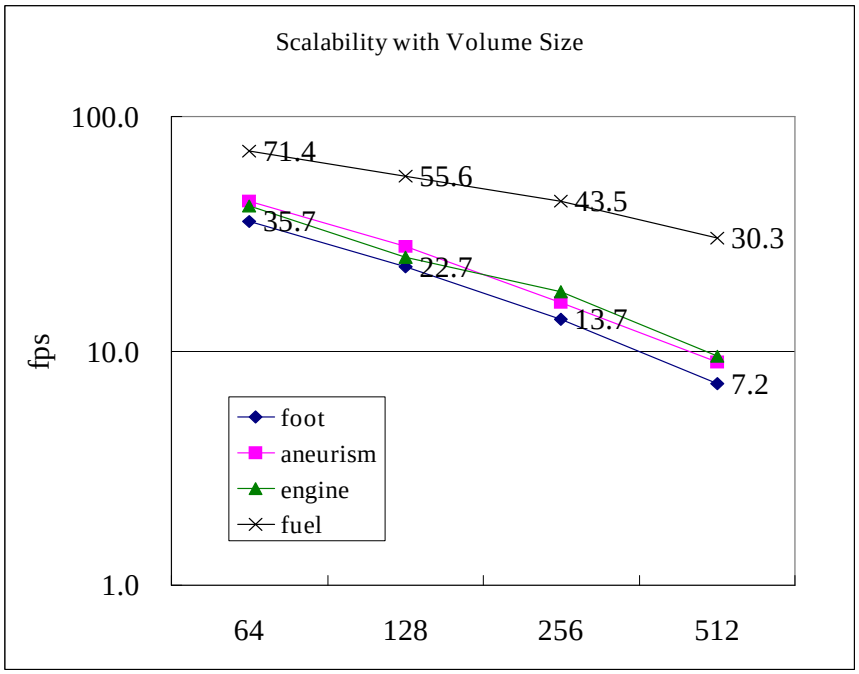

Figure 6.7: Performance with respect to the volume size.

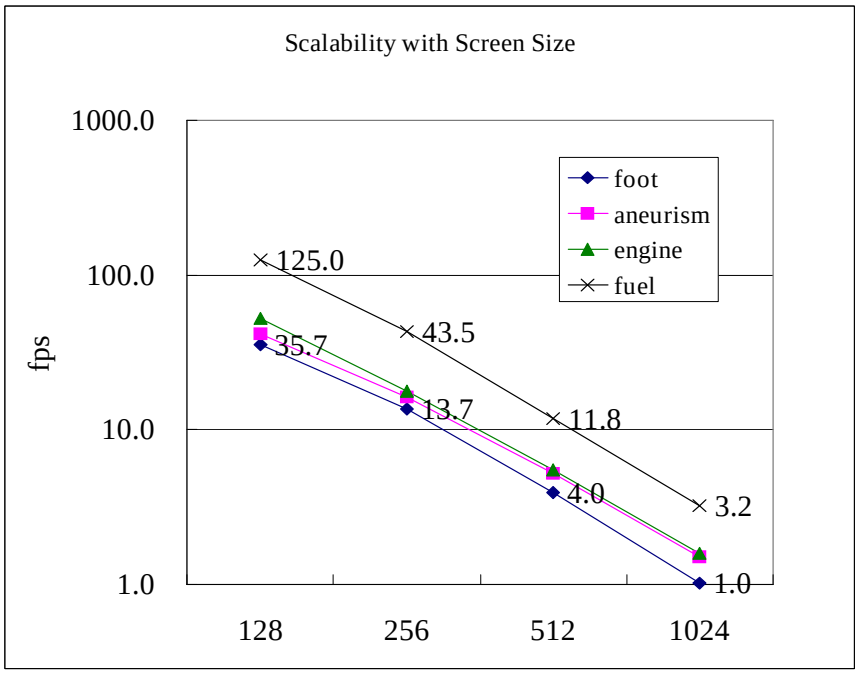

Figure 6.8: Performance with respect to the screen size.

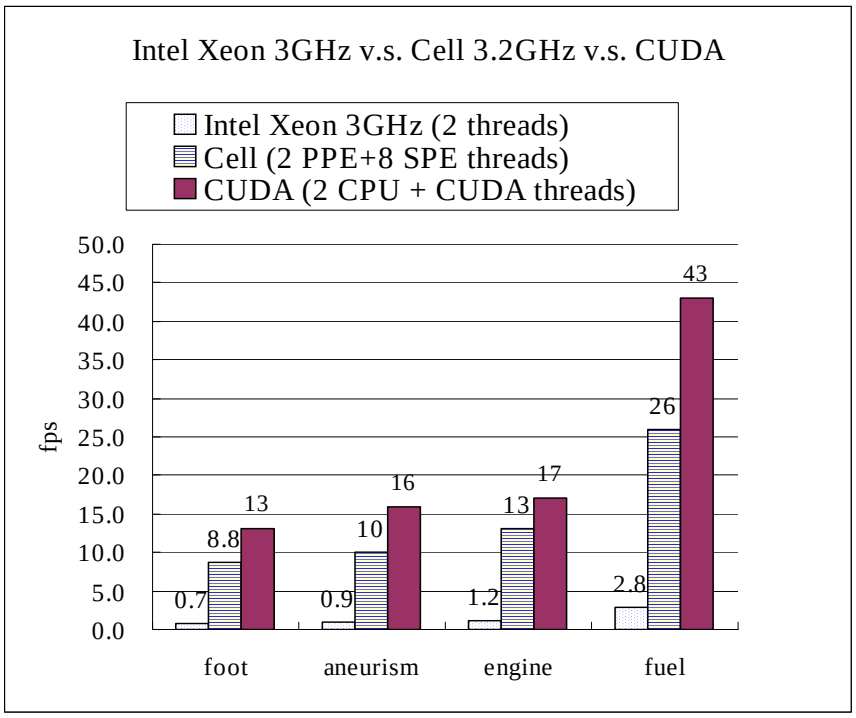

Figure 6.9: Performance comparison (CPU v.s. Cell v.s. CUDA).

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 that 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.