Exploring Parallelism in Volume Ray Casting: Understanding the Programming Issues of Multithreaded Accelerators

Guilherme Cox
Department of Computer Science
Rutgers University
guilherme.cox@rutgers.edu

Cleomar Silva
Department of Electrical Engineering
Pontifical Catholic University of Rio de Janeiro
cleomar@ele.puc-rio.br

Leandro Cupertino
Department of Electrical Engineering
Pontifical Catholic University of Rio de Janeiro
cuper@ele.puc-rio.br

Cristiana Bentes
Department of Systems Engineering
State University of Rio de Janeiro
cris@eng.uerj.br

Ricardo Farias
COPPE/Systems Engineering and Computer Science
Federal University of Rio de Janeiro
rfarias@cos.ufrj.br

ABSTRACT

Direct volume rendering of irregular 3D datasets demands high computational power and memory bandwidth. Recent research in optimizing volume rendering algorithms are exploring the high processing power offered by a new trend in hardware design: multithreaded accelerator devices. Accelerators like the Graphics Processing Units (GPU) and the Cell Broadband Engine processor (Cell BE) are used as integrated coprocessors, and the off-loading of the application from the CPU to the accelerator offers promising speedups. The difficulty in using these devices, however, is how to program them efficiently, since their architectural features may be completely distinct. In this paper, we present some new architectural-aware algorithms for irregular grid rendering based on the ray casting method, designed for the Cell BE and the GPU. We investigate the ray traversal inside each accelerator in terms of data access, load balancing, and code divergence, and find new opportunities for performance optimizations based on the ray casting specific needs. Our results show that squeezing these architectures for performance reveals their limitations and can significantly improve the ray casting efficiency.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*Parallel Programming*

PMAM 2012 February 26, 2012 New Orleans LA, USA

General Terms

Graphics Applications, Parallel Architectures

Keywords

Direct Volume Rendering, Parallel Ray Casting, Accelerator Programming

1. INTRODUCTION

Direct volume rendering with ray casting is a popular technique for visualizing 3D data volumes that captures the overall data domain, considering the volume as a medium in which light can be absorbed, or scattered as it passes through the volume. The images are produced with high quality and without losing the information inside the data. The volume data to be traversed by the rays are usually defined over a grid, and two approaches can be taken: regular and irregular grids. Since regular grids are not suitable to represent disparate fields, we focus here on irregular grid representations, composed by tetrahedral cells.

The ray casting of irregular grids, however, is particularly challenging due to the lack of implicit adjacency of the cells in these grids. The ray casting algorithm has to determine the ray intersection with each tetrahedron, and to handle the geometry and topology of the volume. Therefore, in order to generate high-quality images, irregular grid ray casting has to perform a huge amount of floating-point computations while the dataset is crossed through, resulting in a computationally expensive process. The performance is proportional to the size of the volume data, and the size of the final image to be projected on the screen. Generally irregular grid ray casting achieves only a low fraction of theoretical peak performance.

More contemporary approaches to the optimization of the ray casting computation are geared towards exploiting the high processing power offered by specialized hardware, such as multithreaded accelerator devices. Typically, the computationally intensive parts of the application are off-loaded to the accelerator, which serves as an integrated coprocessor.

These accelerator devices emerged as a new trend in hardware design, and two important examples of such devices are: the Graphics Processing Units (GPU), and the Cell Broadband Engine processor (Cell BE). They offer promising speedups and are available off-the-shelf. It is likely that most supercomputers will be equipped with such devices in the future. Consequently, it is vital that the ray casting algorithms deal with hybrid architectures composed by a mix of regular CPUs and specific accelerators.

The problem in such architectures is how to program these devices efficiently. Their architectural features may be completely different, e.g. number of cores, size of caches, and structure of the memory system. Although many efforts have been devoted to design algorithms that fit to the accelerators, programmers are still having a hard time in trying to get the most of these architectures. Accelerators are typically optimized for specific program structure and data patterns. They will perform poorly for computations that fall outside the optimized usage. GPUs are particularly sensitive to the number of threads that are created, to the global memory accesses, and to code divergence. Cell BE, on the other hand, is sensitive to load balancing, to scheduling, and to data accesses due to the lack of a caching mechanism on the hardware. There is, therefore, a critical need to understand how the ray casting performance is affected by these questions.

This paper builds upon previous work on parallel irregular grid ray casting algorithms for the GPU and the Cell BE proposed, respectively, by Maximo *et al.* in [1] and Cox *et al.* in [2]. We made a detailed performance study of these algorithms on the accelerator devices and observed that, aside from the speedups obtained, squeezing these architectures for performance reveals their limitations and new opportunities for optimizations based on the ray casting specific needs. So, we propose here new architectural-aware algorithms for these platforms. In the design of these algorithms, we closely investigate what happens during the ray traversal inside each accelerator in terms of data access, load balancing, and code divergence, in order to find room for new improvements and performance gains. Our algorithms include mechanisms that: (i) explore a hybrid CPU/GPU overlapping collaborative work; (ii) avoid code divergence in the GPU; (iii) include a software-managed caching mechanism in the Cell BE; and (iv) reduce the load balancing effects in both platforms. Our results show that these mechanisms can provide dramatic performance improvements. The algorithms proposed achieve high speedups even for the memory bound and computationally expensive process of irregular grid ray casting.

The remainder of this paper is organized as follows. In Section 2, we review the previous work on accelerating irregular grid ray casting. In Section 3, we provide a brief description of the two accelerators architectures, the GPU and the Cell BE. Section 4 describes the sequential irregular grid ray casting algorithm: VF-Ray. Section 5 describes the accelerated algorithms for the GPU and the Cell processor. In Section 7, we report our experimental results. Section 8 presents our conclusions and future research plans.

2. RELATED WORK

Ray casting of irregular data has been studied in the literature for many years. An important contribution was provided by Max in [3], where the volume rendering equations were formally described, accounting for the absorption and multiple scattering of the light as it passes through the volume. In terms of the ray casting algorithms, the first algorithms were implemented entirely in software and focused on the data structures to store the connectivity of cells. Garrity [4] proposed a way to compute the entry and exit of each ray, that was further improved by Bunyk *et al.* [5], by computing for each pixel a list of intersections on visible faces. Later, Ribeiro *et al.* [6] proposed consistent and significant gains in memory usage, and in the correctness of the final image, over Bunyk's approach. The algorithm proposed in [6], called VF-Ray, is the basis of the accelerated ray casting implementations.

Recently, with the widespread adoption of hardware accelerators, like GPUs and Cell BE, different implementations of irregular grid ray casting were proposed to explore these architectures. Weiler *et al.* [7] implemented a GPU-based ray casting algorithm that was further extended by Espinha and Celes [8]. Bernardon *et al.* [9] also proposed a GPU-based algorithm based on ray casting that rendered non-convex irregular grids. In a later work [10], they proposed another GPU-based algorithm for irregular grids to handle time-varying data. Some attempts have been made to deal with the problem of the GPU memory limitation. Weiler *et al.* [11], Fout and Ma [12], and Mensmann *et al.* [13] used data compression. Maximo *et al.* [1] implemented a new scheme for storing the face data. Scharsach *et al.* [14] divided the volume into bricks through which rays are cast independently, but focused on regular grids. Mensmann *et al.* [15] also focused on regular grids and proposed the use of stream processing with the volume segmented into into *slabs*.

Another way of dealing with large datasets in the GPU is to use out-of-core techniques. The classic approach of an out-of-core technique is to organize the massive volumetric data into an octree. This approach was widely studied in the context of regular grids, since it is easily breakable into smaller blocks [16, 17, 18, 19]. The work by Smelyanskiy *et al.* [20] proposed a threaded data-parallel implementation of ray casting that explores the architectural trends of multicores and GPUs, and an upcoming many-core processor. They tackled the communication overhead using compression and analyzed the cache behavior of their approach. They focused on regular grids and divided the computation by breaking the volume into bricks.

Recently, there are also some works on exploring multiple GPUs to render large datasets. Strengert *et al.* [21] developed a system in which the volume was divided into bricks and a wavelet was used for compression on small GPU clusters. Marchesin *et al.* [22] also proposed a multi-GPU visualization system based on the division of a volume data into bricks. Their work focused on adapting the volume rendering pipeline commonly used on clusters to a multi-GPU architecture. Muller *et al.* [23] developed a distributed memory volume renderer that runs on multiple GPUs also based on volume bricking, but with some optimizations like empty-space-skipping and load balancing. The recent work by Fogal *et al.* [24] extended Muller *et al.* ideas to allow the rendering of larger datasets, by removing the restriction that the data must fit in the combined texture memory of the GPU. The work by Moloney *et al.* [25] implemented a volume renderer that run on a 32-node GPU system. Their work took advantage of the image decomposition to improve load balancing and to accelerate the rendering using occlusion and culling. Eilemann *et al.* [26] presented a generic and flexible framework for parallel rendering which handles a variety of different data for a wide range of platforms including clusters of GPUs. All these work, however, focused on the rendering of regular datasets.

The power of the Cell BE processor has been exploited in [2] for ray casting of irregular grids and in [27] for regular grids. None of these previous work, however, explored the particularities of different accelerators architectures to improve the memory bound process of the ray traversing in a irregular grid ray casting.

3. ACCELERATORS ARCHITECTURE

In this section, we provide a brief description of the accelerators studied: the Graphics Processing Unit (GPU) and the Cell Broadband Engine (Cell BE).

3.1 Graphics Processing Unit (GPU)

A GPU can be viewed as a highly parallel, many-core stream-processing unit that support a great number of fine-grain threads. The NVIDIA CUDA [28] programming model was created for developing applications for this platform. CUDA allows the programmer to define special C functions, called *kernels*, that are executed in parallel by different CUDA threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves through synchronization and shared memory accesses. During execution, CUDA threads may access data in multiple levels of the memory hierarchy: private local memory, shared memory and global memory. Each thread has a private local memory. Each thread block has a shared memory visible to all threads on the block, and all threads have access to the global memory.

In the GPU architecture, all threads in one block run the same instruction on one streaming multiprocessor (SM). Each SM consists of a number of processing elements, called Stream Processors or SPs. The number of SPs in a SM depends on the architecture and model of the GPU. The threads in a thread block are time-sliced onto these SPs in groups of 32 threads called warps. All the threads in a warp execute the same instruction or remain idle. In this way, different threads can perform branching and other forms of independent work. L1 and L2 caches have been included in the recently released Fermi architecture.

3.2 Cell BE

The Cell BE is a heterogeneous multicore architecture that contains nine cores. One general purpose Power Processing Element (PPE) and eight special purpose Synergistic Processing Elements (SPEs). The SPEs are optimized for compute-intensive tasks, which emphasizes its power to SIMD processing. The SPEs can operate independently from the PPE, but they depend on the PPE to run the operating system and to start the threads. Each SPE includes a small private memory, the Local Store, with 256KB. All of the components of the Cell BE chip are internally connected through a high bandwidth bus, called Element Interconnect Bus (EIB).

Cell BE does not provide caching and the SPEs cannot access main memory directly. Since all the code and data being processed by the SPE must fit in this local memory, any data needed by the SPE, that is stored in the main memory, must be loaded explicitly by software into the local memory, through asynchronous DMA operations. The challenge is to coordinate the DMA operations in order to overlap the data transfers with computation.

4. VISIBLE FACES RAY CASTING (VF-RAY)

In the ray casting paradigm [29], a ray is cast from the viewpoint through each pixel of the image. As the ray moves forward, it intersects a number of cells in the volume data. For each cell, the two intersections with its faces are used to compute the contribution of the cell for the pixel color and opacity. The ray stops when it reaches full opacity, or when it leaves the volumetric data.

The accelerated ray-casting algorithms studied here are based on the sequential Visible Faces Ray Casting (*VF-Ray*) algorithm [6]. VF-Ray handles irregular grids composed by tetrahedral or hexahedral cells. Its main goal is to compute the internal faces intersections efficiently with a minimum memory footprint. The previous results of VF-Ray showed that its memory footprint was only from 1/6 to 1/3 of the memory used by the traditional ray casting approaches, with comparable performance.

In the ray casting algorithm, each time a ray intersects a face of a cell, the face intersection is calculated using a line-plane intersection method from linear algebra. The plane is defined by the plane equation created from the three vertices of a triangular face of one tetrahedron[1]. The computations of these equations are costly, but the geometry and the coefficients of these equations can be reused, since the majority of the faces are intersected by more than one ray. The problem of reusing this data is that a structure to store all the faces would consume a lot of memory space.

The VF-Ray algorithm tackles this problem by storing the face data only for the traversals of the rays under the projection of a visible face (a face whose normal points towards the viewer). These rays have a high probability of intersecting almost the same set of internal faces, and they are called *visible set*. The structure containing the face data is maintained by VF-Ray only during the traversal of a visible set, and is called *face buffer*.

The algorithm starts in a preprocessing step, where the volumetric data is read and organized in memory in a set of data structures that contains: a list of all vertices, a list of all cells, and a list of external faces. For a certain point of view, the rendering process begins by rotating the data, according to the viewing direction. The external faces list is traversed to determine the visible faces of the data.

After that, the core of the VF-Ray renderer is executed, which is shown in Algorithm 1. For each visible face, f_v, the algorithm starts by projecting f_v on the screen determining the visible set of f_v. To perform this projection, the algorithm determines the rectangular *bounding-box* of each triangular visible face and, then, performs a *scan-convert* process to find the pixels under the projection of the face. The *scan-convert* sweeps the rectangular area, and checks for each pixel in the bounding-box if the ray cast through out this pixel intersects the triangular face. Then, for each ray r in the visible set, the algorithm has to compute the entry and exit points, e_{in} and e_{next}, of the ray in each cell. Initially, e_{in} is found in the visible face, f_v. Then, each

[1]The faces of hexahedral cells are broken into two triangles.

exit face, f_{next}, has to be found in order to determine the intersection e_{next}. The next face is found by the function *FindNextFace* that checks which of the other three faces of the tetrahedron that f_{in} belongs is the exit of the ray. Every time a face is traversed, its coefficients are saved in the *face buffer* (if they do not exist already). The lighting integral is computed from e_{in} to e_{next} using an emission-absorption optical model as proposed in [30]. This computation determines the contribution of the cell on the color and opacity of the pixel associated to r.

Algorithm 1 VF-Ray Main loop

for each visible face f_v **do**
  Scan-convert the bounding box of f_v to find the visible set
  for each ray r in visible set of f_v **do**
    $f_{in} \leftarrow f_v$
    $e_{in} \leftarrow$ intersection entry point of r in f_v
    repeat
      $f_{next} \leftarrow$ FindNextFace (r, f_{in})
      if f_{next} exists in *face buffer* **then**
        retrieve parameters of f_{next}
      else
        compute parameters of f_{next}
        Save f_{next} parameters in the *face buffer*
      $e_{next} \leftarrow$ intersect r with f_{next}
      Compute lighting integral from e_{in} to e_{next}
      $f_{in} \leftarrow f_{next}$ and $e_{in} \leftarrow e_{next}$
    until r exits the data
  Clear the *face buffer*

5. GPU ACCELERATED VF-RAY

The parallelization of the ray casting algorithm is relatively simple: every ray cast can be traced through the volume independently from every other ray. In the GPU, this potential parallelism has to map into the fine-grain GPU threads. For the VF-Ray algorithm, a straightforward mapping scheme would be to assign the computation of one visible face to each GPU thread and take advantage of the *face buffer*. This mapping scheme was proposed in [1] and we call this algorithm **VFRay-GPU**. Although VFRay-GPU was designed to parallelize VF-Ray to exploit the massive computational capacity of the GPU, some other architectural features of the GPU were not taken into account. So, we propose here a different approach, called **VFRay-CPU-GPU**, that is more conscious to the difficulties the GPU has in dealing with code divergence and load imbalance. We propose a different work distribution, and a hybrid CPU/GPU approach with an overlapping collaborative work among the CPU and GPU.

5.1 VFRay-GPU

The VFRay-GPU algorithm starts in the CPU by reading the volumetric dataset and computing the external faces of the data. The external faces are copied to the GPU texture memory. After that, the whole rendering process is assigned to the GPU, with three kernels.

In the first kernel, each thread reads one external face from the texture memory and determines if it is visible or not. Its goal is to generate the set of visible faces, that guides the order in which the rays are cast and the face data stored. The first kernel accounts for less than 5% of the total execution time.

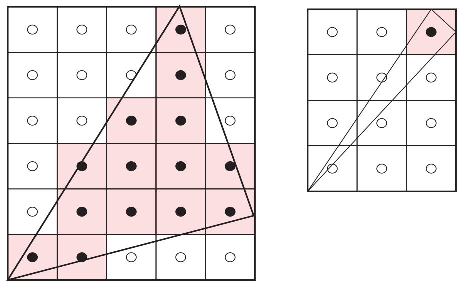

Figure 1: Code divergence example on the scan-conversion.

In the second kernel, each thread computes the bounding box for one visible face. The bounding-boxes are stored in the global memory. This kernel is very simple, with negligible execution time.

In the third kernel, each thread is responsible for performing the ray casting for all pixels in the *visible set* of its assigned visible face. It implements the main loop of the VF-Ray algorithm. Each thread starts retrieving the data for one visible face, f_v, which includes its vertices coordinates and its bounding-box. The algorithm, then closely follows the procedure presented in Algorithm 1. The thread performs a *scan-convert* to find the pixels under the projection of f_v, the visible set. For the pixels in the visible set, the *FindNextFace* function is called to return the intersection, e_{next}, with the next face. The contribution of the current cell to the color of the pixel is computed by solving the lighting integral. The algorithm continues this process until there is no next face, when the ray has left the volumetric data, and the thread ends.

The merit of VFRay-GPU is to provide low memory usage for the memory-consuming problem of irregular grid ray casting. This is an important issue for GPU computation, and VFRay-GPU presented in [1] notable results for rendering on the GPU, when compared to other GPU-based ray casting implementations. The parallelization scheme implemented in VFRay-GPU, however, can generate load imbalance, since the size of the visible sets greatly varies from face to face. In addition, the scan-convert process and the ray computation can generate a lot of code divergence. In Figure 1, we show an example of two different visible faces and their bounding-boxes. If two threads start to scan-convert these faces, they will check each pixel of the bounding-box. In the third pixel check, each thread will follow a different path and they are serialized. Most of the scan-conversion of these faces are serialized, which can significantly degrade the GPU performance.

5.2 VFRay-CPU-GPU

The VFRay-CPU-GPU algorithm was designed to address the load imbalance and the code divergence issues of the GPU. In our approach, we propose a new parallelization scheme where each thread is responsible for the computation of just one pixel. This scheme resulted in a better load balancing among the threads. In addition, we delegate part of the previous third kernel heavy work to the CPU. This hybrid CPU/GPU approach aims to exploit the different architectural abilities of each type of processor to deal with specific operations.

Just like VFRay-GPU implementation, VFRay-CPU-GPU algorithm starts by reading the volumetric data and computing the list of external faces, as a preprocessing stage. VFRay-CPU-GPU is also composed of three kernels. The first kernel determines the visible faces. The second kernel computes the bounding-box for the visible faces, in the same way VFRay-GPU algorithm does.

The third kernel is entirely different. The scan-convert process is not performed by the GPU, but assigned to the CPU. The CPU reads the list of visible faces and their bounding-boxes from the GPU global memory, performs the scan-convert and stores the list of pixels to be rendered in a structure called *pixList*. The overlapping CPU/GPU work begins in the following way: (1) CPU copies the *pixList* to the GPU global memory; (2) The third kernel runs on the GPU; (3) CPU builds the next *pixList*; (4) CPU waits for the GPU to finish the current *pixList*; (5) repeat the loop from step (1) until all the visible faces were processed. Note that, while the GPU kernel is rendering all the pixels in the current *pixList*, the CPU is creating the next *pixList*, and the communication between the CPU and the GPU is done via explicit data copies to the global memory.

Each thread performs the ray casting for only one pixel of the *pixList*. In order to manage the number of pixels in the *pixList*, the CPU takes care of building this list according to the number of threads to be launched on the GPU. If t threads will be launched, the CPU scan-converts a number of visible faces until the *pixList* is populated with at least t pixels. If the *pixList* contains more than t pixels, the remaining pixels are saved for the next list. The task of each thread of the third kernel, therefore, is to perform the traversal of one ray, following the Algorithm 1.

The parallelization scheme of VFRay-CPU-GPU, one pixel per thread, reduces the benefits of using the *face buffer* of VF-Ray, since the computation of one visible face is spread over different cores of the GPU that may not have access to the same shared memory. Nevertheless, it improves the load balancing of the ray casting, since all the threads receives the same amount of pixels to compute. Besides the parallelization scheme, the hybrid CPU/GPU approach also improves the rendering. First, because reducing the size of the third kernel allows the overlapping of the computation of scan-conversion with the ray traversal. Second, because determining if a pixel projects inside a visible face or not, and performing a loop over a set of pixels comprises conditional statements, which may be a major source of performance degradation in the GPU. VFRay-CPU-GPU takes profit from the higher performance of the CPU when dealing with branches and performs these operations in the CPU.

6. VF-RAY ON THE CELL BE

The heterogeneous architecture of the Cell BE, implies a hybrid programming model, since the PPE and the SPEs have different computational capabilities. The PPE is highly optimized for scalar computation and the SPEs are engineered for high speed vector and floating-point computation. This feature of the SPE and the limited size of its local memory, make the programming on the Cell BE to usually follow the practice of computing rather than using pre-computed results. Using a list of precomputed values can be inefficient on Cell BE programs, since it does not SIMDize well and consumes valuable local memory space. So, the straightforward parallelization of the VF-Ray on the Cell BE favor the recomputation of the face data on the SPEs and does not use the *face buffer*. This parallelization was proposed in [2] and we call this algorithm **VRay-Cell**. Although VRay-Cell explored the SIMD and the asynchronous memory transfer abilities of the SPEs, it did not explore the actual capacity of each SPE of storing data. We present here three three new implementations of the VF-Ray algorithm for the Cell BE architecture that explore the local memory of each SPE and implement a software-managed caching mechanism. The first implementation is called **VRay-Cell-Pixel** and distributes the work computation by pixel. The second and third implementations, called respectively **VRay-Cell-Face** and **VRay-Cell-Tile** propose different schemes for the distribution the work load among the SPEs.

6.1 VFRay-Cell

All the Cell BE implementations include two different programs, one that runs on the PPE, and another that runs on the SPE. The PPE program reads and stores the structures for the volumetric data, in the global memory, computes the list of external faces, creates the SPEs threads, and organizes and dispatches the work to the SPEs. The work of preparing the data for distribution is done by the PPE asynchronously with the rendering processing carried by the SPEs.

In VFRay-Cell, the projection of the visible faces is done by the PPE, since the projections are essentially scalar operations. After the projection of a visible face, the PPE is responsible for distributing the work among the SPEs. The work distribution done by the PPE assigns one pixel to each SPE to compute. The SPE algorithm is constantly waiting for a notification message from the PPE. This message may be either a pixel to be processed or a quit signal. The problem with the pixel computation is that a SPE cannot store in its local memory all vertices and cell data of the tetrahedra intersected by the rays. The SPEs have to be constantly fed with such data. This strategy leads to an intensive communication between the SPEs and the main memory, requiring the implementation of a data streaming mechanism in order to avoid SPE stalls while waiting for data. After the first intersection is computed, the rendering loop starts by calling *FindNextFace* function to find the next intersection. With the two intersections, the algorithm can compute the the lighting integral. But before computing the integral, the SPE requests the data for the next tetrahedron. This way, the SPE can overlap the computation of the lighting integral, with the data transfer. Overlapping these memory transfer with computation is vital to get full beneficial results from the Cell BE implementation.

6.2 VFRay-Cell-Pixel

In the VFRay-Cell-Pixel algorithm, the projection of the visible faces is also done by the PPE, which is responsible for distributing the work among the SPEs. The PPE assigns one pixel to each SPE, by sending messages to the SPEs while there are pixels to be computed, just like in the VFRay-Cell algorithm. However, in VFRay-Cell, the computation of the intersection of the ray with a face, requires that the SPE always computes the coefficients of the equations that define the face algebraic representation. Since this computation is expensive, in VFRay-Cell-Pixel we include a *face buffer* mechanism. The problem with including the *face buffer* is that the limited memory of the SPE does not support the whole structure. So, we created in VFRay-Cell-Pixel a *face buffer cache* with a restricted size. This cache is implemented using a hash function to include a new face data in the cache. When there is a collision in the insertion, the old face data is discarded. The first time a face is intersected the face coefficients are computed using the SPE SIMD facilities. When a new visible face computation begins, the face buffer cache is flushed.

Regarding the tetrahedra data for the ray traversal, the VFRay-Cell-Pixel algorithm employs the same data streaming mechanism proposed in VFRay-Cell, and explores the asynchronous memory transfer abilities of the SPEs to feed them with the tetrahedra data.

6.3 VFRay-Cell-Face

In the VFRay-Cell-Pixel work distribution, the computation of a face visible set is spread over the SPEs. In this way, the benefits of the VF-Ray *face buffer* are diminished. The idea behind the *face buffer* is that the rays under the projection of a visible face have high probability of intersecting the same set of internal faces. So, we propose a different algorithm, VFRay-Cell-Face, where the work is distributed by face, rather than by pixel. In other words, in VFRay-Cell-Face, the work message from the PPE contains the information about all the pixels in a visible set.

The SPE algorithm computes the traversal of each ray in the visible set in the same way done by VFRay-Cell-Pixel, using a data streaming mechanism to overcome the SPE memory limitation. After all the pixels in the visible set are computed, the *face buffer cache* is flushed. Computing all the pixels of a visible set in the same SPE allows VFRay-Cell-Face to profit from data reuse in the *face buffer cache.*

6.4 VFRay-Cell-Tile

The third algorithm is called VFRay-Cell-Tile. This algorithm proposes a different scheme for the work distribution on the SPEs. Instead of distributing the pixels of a visible set, VFRay-Cell-Tile divides the screen into tiles, and distributes to the SPE the pixels that lie inside a tile. The idea behind this approach is that a distribution by face would generate load imbalance, just like the load imbalance observed in VFRay-GPU, due to the different sizes of the visible sets.

In VFRay-Cell-Tile, the PPE algorithm is slightly different. According to the image resolution, the PPE first divides the screen into $R \times R$ rectangular tiles, where the tile resolution is proportional to the image size. The PPE, then, computes the external and the visible faces. For each visible face, the algorithm associates the face with a tile on the screen. A face belongs to the same tile that its first vertex belongs to. After this association is done, the tile division changes. Instead of dividing the pixels in rectangular tiles, the algorithm, in fact, aggregate the visible faces in sets with similar sizes, as shown in Figure 2.

The PPE sends to each SPE one visible face set for computation. The SPE algorithm, then, computes the traversal of each ray in the visible faces set in the same way done by VFRay-Cell-Pixel and VFRay-Cell-Face. However, in VFRay-Cell-Tile, the *face buffer cache* is maintained not only in the computation of one visible face, but during the computation of all the faces in the set. In this way, the *face buffer cache* is flushed when the computation of the set finishes. The grouping of visible faces done by VFRay-Cell-Tile provides a more reasonable load distribution among the SPEs than the visible face distribution implemented in VFRay-Cell-Face.

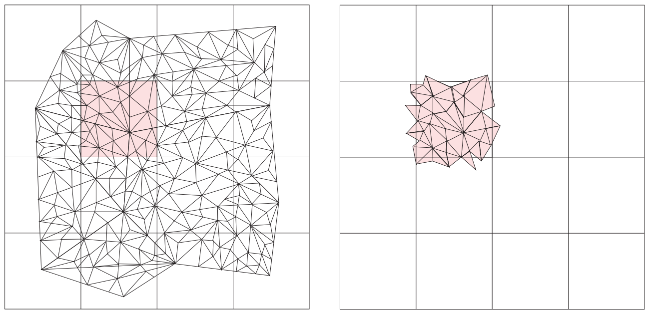

Figure 2: Grouping the visible faces guided by tile screen division.

7. EXPERIMENTAL RESULTS

In this section, we present the performance evaluation of the GPU and Cell BE algorithms. We first describe our experimental setup and datasets, and then present the results for the two accelerators separately. It is not our intention to compare the GPU and Cell BE results, since the two accelerators are completely different in terms of the number of cores and clock frequency. In addition, the Cell BE processor is a relatively outdated processor when compared to the GPU used in our tests. Our idea here is to show the performance of our algorithms in each of the accelerators and the necessary changes in the ray casting algorithm to take the best from such different architectures.

7.1 Experimental Setup

The implementation of VF-Ray on the GPU was written in C++ and CUDA, using CUDA driver 3.1 and SDK 3.0. The Cell BE implementation was written in C++, using SIMD extensions included with the IBM SDK 3.1. The GPU implementation was tested on the NVIDIA GTX 480 based on the Fermi architecture (480 cores, 1.4 GHz, 1.5 GB). The Cell BE implementation was tested on a Sony Playstation3 console, where the Cell BE processor operates with only six out of the eight SPEs. The CPU used as the baseline for the sequential execution was an Intel Quad Core 2.66GHz with 2MB of L2 cache and 2GB of memory.

We used four well-known representative tetrahedral datasets: SPX from Lawrence Livermore National Lab, Liquid Oxygen Post, Blunt Fin, and Delta Wing from NASA. SPX is a real irregular dataset, containing a hole in the grid, bringing extra difficulties to the renderer. Delta Wing, Liquid Oxygen Post and Blunt Fin are tetrahedralized versions of regular, curvilinear datasets. Liquid Oxygen Post, in particular, is very thin and presents different rendering complexity according to the viewing direction. The images rendered with VFRay-CPU-GPU for all these datasets are shown in Figures 3 to 6. Table 1 shows the number of vertices, external faces, and tetrahedra cells of each dataset. We rendered images with different resolutions, varying from 512×512 to 4096×4096 pixels, from different points of view.

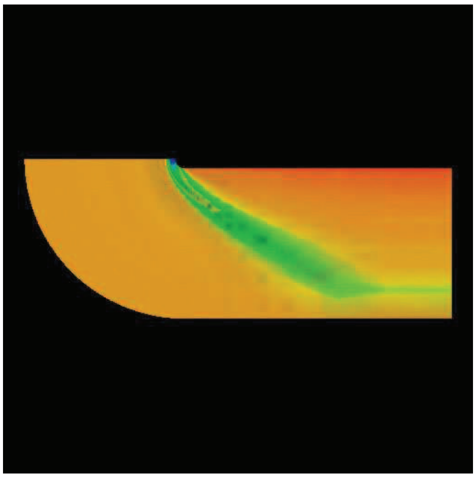

Figure 3: Blunt Fin image.

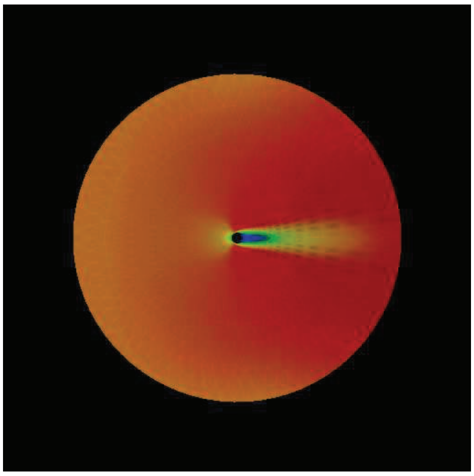

Figure 4: Oxygen Post image.

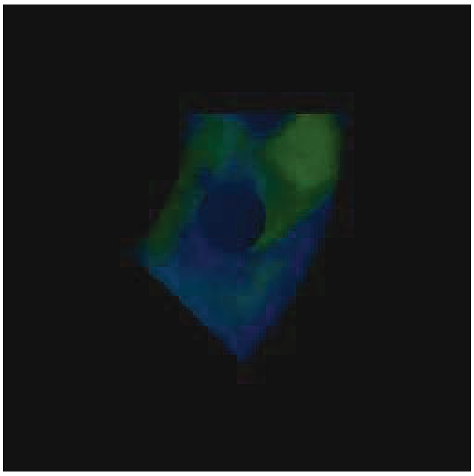

Figure 5: SPX image.

7.2 GPU Results

Table 2 shows the execution times for all datasets and images resolutions, for VFRay-GPU and VFRay-CPU-GPU. As we can observe in this table, VFRay-CPU-GPU outperforms VFRay-GPU for all datasets in all resolutions. The performance difference, however, is more significant for larger image resolutions. As the image resolution increases, the visible faces project to a larger amount of pixels, and, therefore, the effects of load imbalance are more prominent for VFRay-GPU. In addition, since the GPU performance is very sensitive to the presence of conditional branching, another important improvement of VFRay-CPU-GPU is to put the CPU in charge of the scan-conversion of the visible faces and the work distribution. The code divergence generated by the scan-conversion process, exemplified in Figure 1, runs almost sequentially in the third kernel of VFRay-GPU. To measure the effect of code divergence, we modified the third kernel of VFRay-CPU-GPU by inserting a loop control mechanism. This is done by making each thread render two pixels in the *pixList* instead of one. The loop control mechanism slowed down the execution in about 10 times compared to the results of the original VFRay-CPU-GPU. In addition to the reduction in the code divergence, the hybrid CPU-GPU strategy used in VFRay-CPU-GPU also reduced the number of registers used per thread.

Table 3 shows the execution times for the VF-Ray algorithm running on the CPU, and the speedups obtained by VFRay-CPU-GPU when compared to the CPU execution. As we can observe, VFRay-CPU-GPU provides impressive accelerations over the CPU execution for all the datasets. As expected, the best speedup results were obtained for the largest image resolution, where more pixels are computed.

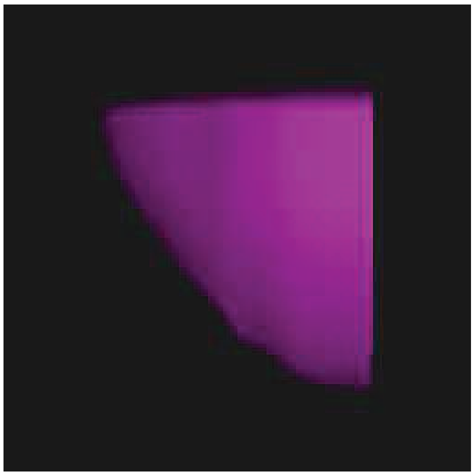

Figure 6: Delta Wing image.

Dataset	# Verts	# Ext Faces	# Tets
Blunt Fin	41 K	381 K	187 K
Oxygen Post	109 K	1.0 M	513 K
SPX	149 K	1.6 M	827 K
Delta Wing	211 K	2.0 M	1.0 M

Table 1: Datasets description.

7.3 Cell Results

Table 4 compares the execution times of VFRay-Cell and VFRay-Cell-Pixel for 6 SPEs, all datasets and all images resolutions. VFRay-Cell-Pixel obtained considerable gains over

	VFRay-GPU				VFRay-CPU-GPU			
Resolution	Blunt	Post	SPX	Delta	Blunt	Post	SPX	Delta
512 × 512	0.07	0.17	0.43	0.67	0.08	0.12	0.12	0.12
1024 × 1024	0.23	0.66	1.51	2.67	0.12	0.21	0.19	0.21
2048 × 2048	0.87	2.75	6.32	10.59	0.22	0.54	0.34	0.46
4096 × 4096	3.53	11.70	25.29	42.40	0.58	1.80	0.97	1.44

Table 2: VFRay-GPU and VFRay-CPU-GPU execution times (sec).

	CPU				VFRay-CPU-GPU Speedups			
Resolution	Blunt	Post	SPX	Delta	Blunt	Post	SPX	Delta
512 × 512	1.25	2.70	2.22	3.72	15.6	22.5	18.5	31.0
1024 × 1024	4.75	10.44	6.57	13.65	39.6	49.7	34.6	65.0
2048 × 2048	18.60	40.98	23.41	52.79	84.5	75.9	68.8	114.7
4096 × 4096	73.54	161.98	90.65	208.32	126.8	90.0	93.4	144.7

Table 3: CPU results (in seconds) and speedup for VFRay-CPU-GPU.

VFRay-Cell. For larger image resolutions, the VFRay-Cell-Pixel runs almost two times faster than VFRay-Cell. These results confirm that the caching of face data controlled by software brings significant performance gains to the ray casting algorithm. In other words, the overhead introduced by the *face buffer* mechanism is compensated by the elimination of the recomputation of the faces parameters.

The gains obtained by the caching mechanism we introduced, however, do not increase linearly with the increase in the dataset size. In fact, the gains depend on the dataset shape and on the point of view. The problem arises with the irregularity of the datasets. A dataset in which the rays have a long length for their traveling path can produce worse results, due to the overhead of searching for the face in the *face buffer cache*. This overhead may surpass the cost of recomputing the face.

Table 5 shows the execution times of VFRay-Cell-Face and VFRay-Cell-Tile (tile division 256 × 256), for 6 SPEs execution, all datasets and images resolutions. The times in this table show that both VFRay-Cell-Face and VFRay-Cell-Tile outperform VFRay-Cell-Pixel. These results confirm that the better use of the face cache data improves the ray casting efficiency. When we compare VFRay-Cell-Face with VFRay-Cell-Tile, we observe that VFRay-Cell-Tile performs slightly better, due to the better load balancing achieved. However, the tile size can have influence in the load balancing, smaller tiles tend to generate more imbalanced work, and for a tile division smaller than 128 × 128, VFRay-Cell-Face outperforms VFRay-Cell-Tile.

7.4 Discussion

The ray casting computation comprises the computation of the rays intersections with the volumetric data and the computation of the lighting integral. Within the rendering loop, about 30% of the time is spent in the lighting integral, while the other 70% of the time is spent in finding the next intersection. The tradeoff in designing an efficient algorithm for irregular grid ray casting is that the irregular nature of the grid implies in large data structures to handle the geometry and topology of the volume, and the more information is kept in memory, the faster the algorithm computes the next intersection of the ray. One of the fastest sequential algorithms for ray casting was proposed by Bunyk *et al.* in [5], but its memory consumption would be prohibitive for running in accelerator devices with current datasets.

The interesting issue about the modern accelerators is that their architectures favors processing over memory in their die area. So the ray casting algorithm must be carefully redesigned to fit in these architectures. Based on the observation that the decision of how to store the face data is the key for memory consumption, our study observed different behavior depending on the accelerator architecture. The basis of the VF-Ray algorithm is the *face buffer*. The use of this buffer, however, had opposite significance in the GPU and Cell BE algorithms. This was observed in the distribution of the ray computation in the different algorithms. When the computation is distributed one pixel to each core, the benefits of using the *face buffer* is reduced since the data from the previous pixel computation is not reused. In the GPU architecture, where there is a great number of cores and a very small cache shared by all cores[2], it was better to recompute the face data than to reuse it. The GPU was more sensitive to load imbalance and code divergence, but has enough cores to recompute the faces parameters. In the Cell BE architecture, on the other hand, there is a moderate number of cores and each core has a limited amount of memory inside it. In this case, it was better to distribute the work by tiles and take benefit from the *face buffer* than performing the computation pixel-by-pixel.

8. CONCLUSIONS

The new trend in hardware design, multithreaded accelerators, offers opportunities and challenges for parallel irregular grid ray casting algorithms. In this work, we presented a detailed study on irregular grid ray casting algorithms designed on the VF-Ray approach for the Cell BE and the GPU architectures. This study allowed us to better understand the accelerators limitations and propose some new architectural-aware algorithms, where we focus on handling CPU/GPU collaborative work, code divergence in the GPU, data caching and load balancing.

The overall results of our study demonstrated that the most natural parallelization scheme of the VF-Ray algorithm on the Cell BE and on the GPU presented some performance issues and left room for improvements. The VF-Ray algorithm introduced the *face buffer* structure to overcome the memory bound process of traversing a irregular grid. Our results showed that the use of this structure had different impact in the accelerated algorithms. In the GPU architecture, the *face buffer* had limited relevance,

[2]We run our experiments on the Fermi architecture

	VFRay-Cell				VFRay-Cell-Pixel			
Resolution	Blunt	Post	SPX	Delta	Blunt	Post	SPX	Delta
512 × 512	1.83	3.02	1.64	2.32	1.32	1.57	1.51	1.64
1024 × 1024	7.22	12.10	6.31	9.22	5.23	5.86	4.42	5.72
2048 × 2048	28.81	48.41	25.21	36.85	20.71	22.84	14.86	22.04
4096 × 4096	116.61	195.02	101.10	149.5	83.21	91.20	56.44	88.26

Table 4: VFRay-Cell and VFRay-Cell-Pixel execution time results (sec).

	VFRay-Cell-Face				VFRay-Cell-Tile			
Resolution	Blunt	Post	SPX	Delta	Blunt	Post	SPX	Delta
512 × 512	1.26	1.43	1.12	1.39	1.12	1.46	1.15	1.36
1024 × 1024	4.99	5.49	3.53	5.09	4.40	5.54	3.58	4.89
2048 × 2048	19.92	21.69	13.04	19.82	18.03	21.68	13.28	19.31
4096 × 4096	79.79	86.67	51.26	79.26	70.58	86.98	52.27	77.13

Table 5: VFRay-Cell-Face and VFRay-Cell-Tile execution time results (sec).

with a huge number of cores it was better to recompute the data. In the Cell BE, on the other hand, the use of a *face buffer cache* provided important performance gains. Besides the *face buffer* effects, one of our observations is that load imbalance, allied to the cost in handling threads with divergent computations, can easily degrade the ray casting performance in the accelerators. The GPU showed to be very sensitive to code divergence, and both accelerators required a careful work distribution scheme to avoid load imbalance. Speedup results demonstrated that being more conscious to the details of the accelerator architecture can provide significant performance improvements. Despite of the high speedup achieved, generalizing an algorithm for these platforms is difficult, there is still a significant learning curve and optimization effort involved in porting codes to these environments.

While modern accelerators have opened the door to cheap and powerful data-parallel architectures, these devices do not necessarily always present good performance results. Our future work aims to investigate further optimizations on the hybrid CPU-GPU version and a multi-GPU implementation.

9. REFERENCES

[1] A. Maximo, S. Ribeiro, C. Bentes, A. Oliveira, R. Farias, Memory efficient gpu-based ray casting for unstructured volume rendering, in: IEEE/EG International Symposium on Volume and Point-Based Graphics, 2008, pp. 55–62.

[2] G. Cox, A. Maximo, C. Bentes, R. Farias, Irregular grid raycasting implementation on the cell broadband engine, in: Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing, 2009, pp. 93–100.

[3] N. Max, Efficient light propagation for multiple anisotropic volume scattering, in: In Proceedings of the 5th Eurographics Workshop on Rendering, 1994, pp. 87–104.

[4] M. P. Garrity, Raytracing irregular volume data, in: Proceedings of the Workshop on Volume Visualization, ACM Press, 1990, pp. 35–40.

[5] P. Bunyk, A. Kaufman, C. Silva, Simple, fast, and robust ray casting of irregular grids, Advances in Volume Visualization, ACM SIGGRAPH 1 (24) (1998) 30–37.

[6] S. Ribeiro, A. Maximo, C. Bentes, A. Oliveira, R. Farias, Memory-aware and efficient ray-casting algorithm, in: Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing, 2007, pp. 147–154.

[7] M. Weiler, M. Kraus, M. Merz, T. Ertl, Hardware-Based Ray Casting for Tetrahedral Meshes, in: Proceedings of the 14th IEEE conference on Visualization, 2003, pp. 333–340.

[8] R. Espinha, W. Celes, High-quality hardware-based ray-casting volume rendering using partial pre-integration, in: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing, 2005, pp. 273–281.

[9] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, C. T. Silva, GPU-based Tiled Ray Casting using Depth Peeling, Journal of Graphics Tools 11.3 (2006) 23–29.

[10] F. F. Bernardon, S. P. Callahan, J. L. Comba, C. T. Silva, An adaptive framework for visualizing unstructured grids with time-varying scalar fields, Parallel Computing 33 (6) (2007) 391–405.

[11] M. Weiler, P. N. Mallon, M. Kraus, T. Ertl, Texture-encoded tetrahedral strips, in: Proceedings of the IEEE Symposium on Volume Visualization and Graphics, 2004, pp. 71–78.

[12] N. Fout, K.-L. Ma, Transform coding for hardware-accelerated volume rendering, IEEE Transactions on Visualization and Computer Graphics 13 (6) (2007) 1600–1607.

[13] J. Mensmann, T. Ropinski, K. H. Hinrichs, A gpu-supported lossless compression scheme for rendering time-varying volume data, in: IEEE/EG International Symposium on Volume Graphics, Eurographics Association, 2010, pp. 109–116.

[14] H. Scharsach, M. Hadwiger, A. Neubauer, S. Wolfsberger, Perspective isosurface and direct volume rendering for virtual endoscopy applications, in: Eurographics/IEEE-VGTC Symposium on Visualization, 2006, pp. 315–322.

[15] J. Mensmann, T. Ropinski, K. H. Hinrichs, An advanced volume raycasting technique using gpu stream processing, in: GRAPP: International Conference on Computer Graphics Theory and Applications, INSTICC Press, Angers, 2010, pp. 190–198.

[16] E. Gobbetti, F. Marton, J. A. I. Guitian, A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets, Visual Computer 1 (24) (2008) 797–806.

[17] W. Li, K. Mueller, A. Kaufman, Empty space skipping and occlusion clipping for texture-based volume rendering, in: Proceedings of the 14th IEEE Visualization, IEEE Computer Society, Washington, DC, 2003, pp. 42–50.

[18] C. Lux, B. Fröhlich, Gpu-based ray casting of multiple multi-resolution volume datasets, in: Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 104–116.

[19] J. Xue, K. Lu, J. Tian, An efficient out-of-core volume rendering method based on ray casting and gpu acceleration, in: IEEE Youth Conference on Information, Computing and Telecommunication, 2009, pp. 130–133.

[20] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. M. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V. W. Lee, A. D. Nguyen, L. Seiler, R. Robb, Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures, IEEE Transactions on Visualization and Computer Graphics 15 (6) (2009) 1563–1570.

[21] M. Strengert, M. MagallÃşn, D. Weiskopf, S. Guthe, T. Ertl, Hierarchical visualization and compression of large volume datasets using gpu clusters, in: In Eurographics Symposium on Parallel Graphics and Visualization, 2004, pp. 41–48.

[22] S. Marchesin, C. Mongenet, J.-M. Dischler, Multi-gpu sort-last volume visualization, in: Symposium on Parallel Graphics and Visualization, 2008, pp. 1–8.

[23] C. Muller, M. Strengert, T. Erl, Optimized volume raycasting for graphics-hardware-based cluster systems, in: Eurographics Symposium on Parallel Graphics and Visualization, 2006, pp. 59–66.

[24] T. Fogal, H. Childs, S. Shankar, J. Krueger, D. Bergeron, P. J. Hatcher, Large data visualization on distributed memory multi-gpu clusters, in: High Performance Graphics, 2010, pp. 57–66.

[25] B. Moloney, M. Ament, D. Weiskopf, T. Moller, Sort-first parallel volume rendering, IEEE Transactions on Visualization and Computer Graphics 17 (2011) 1164–1177.

[26] S. Eilemann, M. Makhinya, R. Pajarola, Equalizer: A scalable parallel rendering framework, IEEE Transactions on Visualization and Computer Graphics 15 (2009) 436–452.

[27] J. Kim, J. Jaja, Streaming model based volume ray casting implementation for cell broadband engine, Scientific Programming 17 (1-2) (2009) 173–184.

[28] NVIDIA, CUDA, `http://developer.nvidia.com/object/cuda.html` (2010).

[29] S. D. Roth, Ray Casting for Modeling Solids, Computer Graphics and Image Processing 18 (2) (1982) 109–144.

[30] N. Max, Optical models for direct volume rendering, IEEE Transactions on Visualization and Computer Graphics 1 (1995) 99–108.