21st International Symposium on Computer Architecture and High Performance Computing

Irregular Grid Raycasting Implementation on the Cell Broadband Engine

Guilherme Cox[*], André Máximo[†], Cristiana Bentes[*] and Ricardo Farias[†]
[*]*Department of System Engineering, State University of Rio de Janeiro*
Rua São Francisco Xavier, 524, Bl. D, 5o floor – Rio de Janeiro, RJ, Brazil, 20550-900
Email: cris@eng.uerj.br, cox@eng.uerj.br
[†]*System Engineering and Computer Science Program-COPPE, Federal University of Rio de Janeiro*
Cidade Universitária, Centro de Tecnologia, Bl. H – Rio de Janeiro, RJ, Brazil, 21945-970
Email: andmax@cos.ufrj.br, rfarias@cos.ufrj.br

***Abstract*—Direct volume rendering has become a popular technique for visualizing volumetric data from sources such as scientific simulations, analytic functions, medical scanners, among others. Volume rendering algorithms, such as raycasting, can produce high-quality images, however, the use of raycasting has been limited due to its high demands on computational power and memory bandwidth. In this paper, we propose a new implementation of the raycasting algorithm that takes advantage of the highly parallel architecture of the Cell Broadband Engine processor, with 9 heterogeneous cores, in order to allow efficient raycasting of irregular datasets. All the computational power of the Cell BE processor, though, comes at the cost of a different programming model. Applications need to be rewritten, which requires using multithreading and vectorized code. In our approach, we tackle this problem by distributing ray computations using the visible faces, and vectorizing the lighting integral operations inside each core. Our experimental results show that we can obtain good speedups reducing the overall rendering time significantly.**

***Keywords*-parallel rendering; cell processor; raycasting;**

I. Introduction

Interpreting the surrounding world as visual images is an activity humans have been engaged in throughout history. Scientific visualization plays an important role in the scientific process, as simulations, experiments and data collection comprise an enormous and permanently increasing accumulation of information. Many application areas, including medicine, geology, biology, chemistry, fluid dynamics, molecular science or environmental protection, depend more and more on visualization as an effective way to obtain an intuitive understanding of problems.

There are many different techniques for visualizing three dimensional data. Basically these can be separated as indirect and direct methods. Indirect methods are based on the reconstruction of polygonal data, which can be rendered like isosurface data. In this method, only part of the volumetric dataset directly contributes to the output image. In direct methods, on the other hand, all the data has the potential to contribute to the output image. The aim of direct methods, called volume rendering, is to map a set of data values defined throughout a volumetric grid to some color and opacity values over the final image. Volume rendering has the advantage of visualizing the complete dataset, exposing its interior, producing high-quality images, but due to the excessive amount of sampling and composition operations performed, they demand high computational power.

Furthermore, depending on the type of the grid used to represent the volumetric data, memory requirements can also be another obstacle for volume rendering algorithms. Irregular grids, with tetrahedral and/or hexahedral cells, have vertices at arbitrary locations. For this reason, irregular grids can represent the field only where it is relevant. However, because of the lack of implicit adjacency in the irregular grids, the coordinates of the vertices need to be explicitly represented, as well as the connectivity among them. As a result, irregular volume grids require relatively larger storage space than regular grids.

Therefore, the main challenges in designing high-performance volume rendering algorithms for irregular grids are: (i) the computational power required for traversing the data in order to compute the color and opacity of each pixel on the screen, and (ii) the memory usage required for keeping all connectivity information for the data.

Significant research efforts have attempted to tackle these challenges using massively parallel graphics processors (GPU) [1], [2], [3], [4] and cluster-based parallelism [5], [6], [7], [8]. These studies take advantage of the continuing improvements in CPU and GPU performances as well as increasing multicore cluster-based parallelism.

Despite of the great advances achieved by these two approaches, some drawbacks must be pointed out. The implementation of a performance-aware raycasting algorithm in the GPU, requires the programmer to face with: the limited memory capacity, and the high-latency GPU-CPU data transfer. While the implementation on clusters requires fine-tuned implementations in order to solve the data distribution, load balancing and image composition communication problems.

So, in this work, we propose an alternative way for high-performance volume rendering of irregular grids rather than the use of Graphics Processing Units (GPUs) or cluster-based parallelism: exploring the power of the Cell Broadband Engine processor. The Cell BE processor is a brand new heterogeneous multicore architecture developed by IBM, Sony, and Toshiba [9]. It is used in the game console Playstation 3 (PS3), and was also the basis for the fastest supercomputer in the world, as announced in the last Top 500 [10]. All the computational power of the Cell BE processor comes at the cost of a different programming model. Applications need to be rewritten in order to explore the full potential of the Cell BE, which requires the use of multithreading, the vectorization the code, and the management of memory accesses. The overall performance strongly depends on the the effective use of Cell BE hardware which is largely left to the programmer.

Our approach here is to implement on the Cell BE, a memory-efficient raycasting algorithm for volume rendering of irregular grids, that deals with two main challenges: (i) conforming to the memory model of the Cell BE, and (ii) exploring SIMD programming. We propose a new parallel algorithm that distributes ray computations to each core using the visible faces, and vectorizes the lighting integral operations inside each core. Our experimental results show that we can obtain good speedups in the rendering process.

The remainder of the paper is organized as follows. In Section II, we review the previous work in speeding up volume rendering and in the use of Cell processor. In Section III, we give a brief description of the Cell BE architecture. In Section IV, we discuss the Cell BE programming model. In Section V, we describe the raycasting algorithm used as the basis for our implementation. In Section VI, we show our Cell implementation of the raycating algorithm. In Section VII, we report our experimental results. Section VIII presents our conclusions and future research plans.

II. Related Work

As far as we know, this is the first Cell BE implementation of a raycasting algorithm for irregular data. So, we give here a brief summary of the literature in terms of: speeding up volume rendering, and programming for the Cell BE.

A. Speeding up Volume Rendering

In recent years, there has been a growing literature about algorithms and techniques for speeding up volume rendering, in particular with a focus on GPU implementations and cluster architectures. They have been concentrated on efficient and optimized implementations of volume rendering algorithms and data structures used to explore GPU features and to avoid clusters bottlenecks. The cluster-based works focused on solving: load balancing [7], [11], screen partition [12], [13], dataset division [6], [14], or image composition communication [5], [8]. The GPU-based works focused on: exploring the GPU hardware features [15], [1], dealing with the memory limitation problem [16], [3], and exploring GPU clusters [17], [4]. Both cluster-based and GPU implementations expose to programmers the specificity of the platform, just as Cell BE implementation. The difference is that in the Cell processor, dynamic management of local memories of the cores is needed, which complicates the rendering of irregular data. However, this exposed memory management opens up the opportunity for peformance gains.

The work by Meibner *et al.* [18] goes on a different direction and present an implementation of a parallel raycasting algorithm on a single-chip SIMD architecture, the FUZION chip. The works by Adinetz *et al.* [19] and Wald *et al.* [20] explore the SSE (Streaming SIMD Extension) support of Intel architecture to accelerate the rendering process. All these three works, however, focus on raytracing that handles only surface data.

B. Exploring the Cell BE Processor

Being a relatively new architecture, the full potential of the Cell processor is still being explored. Some initial works were developed by IBM on medical imaging [21], and FFT computation [22]. These works provided results indicating substantial speedups when compared to conventional processors.

In terms of volume visualization, the works by Benthin *et al.* [23] and O'Conor *et al.* [24] explored the Cell processor architecture for the ray-tracing algorithm for surface data. IBM also published a work in volume visualization [25]. They implemented a raycasting for terrain rendering, where again, only the surface data is considered. The recent work by Kim and Jaja [26] is the most related to ours. They implemented the raycasting algorithm on the Cell processor, which, however, only handles regular datasets.

III. The Cell BE Processor Architecture

The Cell BE is quite unique as a processor. It is a heterogeneous multicore architecture that combines a traditional PowerPC core with multiple mini-cores, that have limited, but SIMD-optimized, set of instructions. A single chip contains a Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The SPEs are optimized for compute-intensive tasks, and can operate independently from the PPE. However, they run threads spawned by the PPE, and depend on the PPE to run the operating system.

The PPE is a traditional 64-bit dual-thread PowerPC, with a Vector Multimedia extension (VMX) unit and two levels of on-chip 32KB cache L1 for instruction and another 32KB for data, and 512KB of L2 cache.

The SPEs are the primary computing engines of the Cell processor. Each SPE is a special purpose RISC processor with 128-bit SIMD capability that runs a Cell-specific set of instructions. It consists of a processing core, the Synergistic Processing Unit (SPU), a Memory Flow Controller (MFC), and a 256KB local storage memory area. The local storage is used to hold code and data being processed by the SPE, but it is not a cache. The SPEs cannot access main memory directly, so 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 storage, through a DMA operation. Data transferred between local storage and main memory must be 128-bit aligned. The size of each DMA transfer can be at most 16KB. Once the data is in the local memory, the SPU can use it by explicitly loading it into one of its 128 general-purpose registers, each 128 bits wide. The SPU instruction set is different from the PPE instruction set and consists of 128-bit SIMD instructions. Two SIMD instructions can be executed per clock cycle in the SPE.

Figure 1 shows the architecture of the Cell BE processor. The PPE accesses the main memory via the L1 and L2 caches. The PPE, SPEs, DRAM controller, and I/O controllers are all connected via four on-chip data rings, called Element Interconnect Bus (EIB). The EIB can transmit 96 bytes per cycle, for a bandwidth of 204.8 Gigabytes/second, allowing more than 100 outstanding DMA requests.

IV. Programming for the Cell BE

The Cell BE processor presents to the programmer a different programming model. The application performance depends on the effective use of the special features of the Cell BE processor.

The Cell BE processor can be programmed using standard C and relying on the libraries from the Sofware Development Kit (SDK) [27], that allows handling communication, synchronization, and SIMD computation. An existing application would run on the Cell processor by a simple recompilation of the code using only the PPE core, with no effort, but also without advantages from a performance point of view.

There are several key differences in coding for the Cell processor compared to traditional CPUs:

1) The power of the SPEs comes from SIMD vector operations. The SPEs are not optimized to run scalar code and handling unaligned data. High performance can only be reached if the data is organized in a way that is suitable for SIMD calculations.
2) SPEs have no cache memory. The DMA engine is exposed. The explicit memory access programming poses some extra work when compared to normal cache-based memory hierarchy. All the code and variables must be allocated in the local store. Larger data structures in main memory can be accessed, one block at a time, via explicit DMA transfers. Furthermore, the DMA calls in the code have to be designed to use double buffering or similar tricks to avoid stalling due to latency.
3) SPEs lack branch prediction hardware. This feature allowed the engineers to pack more computation cores into the chip. However, a branch costs around 20 cycles, which implies that they should be avoided in performance-aware codes.

Nevertheless, the PPE and the SPE are not binary compatible. Summing up, the programming model is an important aspect which distinguishes the Cell processor from other processors. The Cell architecture requires careful designing by the programmer to ensure that efficient SIMD code is generated, SPEs are being well exploited by parallel threads, and the data movement from main memory is orchestrated.

V. Visible Faces Raycasting

Our volume rendering implementation on the Cell BE processor is based on the raycasting algorithm. The basic idea behind raycasting is to cast rays from the viewpoint through each pixel of the image. For irregular volume data, as a ray moves forward, it penetrates the volume intersecting a number of cells, as illustrated in Figure 2. Every pair of intersections is used to compute the cell contribution for the pixel color and opacity. The ray stops when it leaves the volume or when it reaches full opacity.

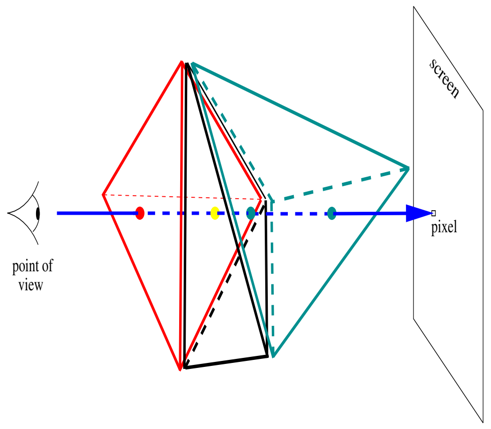

Figure 2. Raycasting scheme.

The great advantage of raycasting methods are that: the computation for each pixel is independent of all other pixels, and, for rendering irregular grids, the traveling of a ray throughout the grid is guided by the connectivity of the cells, avoiding the need of sorting the cells. The disadvantage is the high memory consumption. So, we decided to ground our raycasting Cell implementation on a recent memory-efficient raycasting algorithm for irregular grids, called Visible Faces Raycasting (*VF-Ray*) [28]. This algorithm uses a more compact and non-redundant data structure that provides consistent and significant gains in memory usage

The main focus of VF-Ray is to drastically reduce the memory consumption in order to render very large datasets, based on the fact that the decision of how to store the information about the faces of each cell is the key for memory consumption. This information is stored in a face data structure, that includes the geometry (usually three

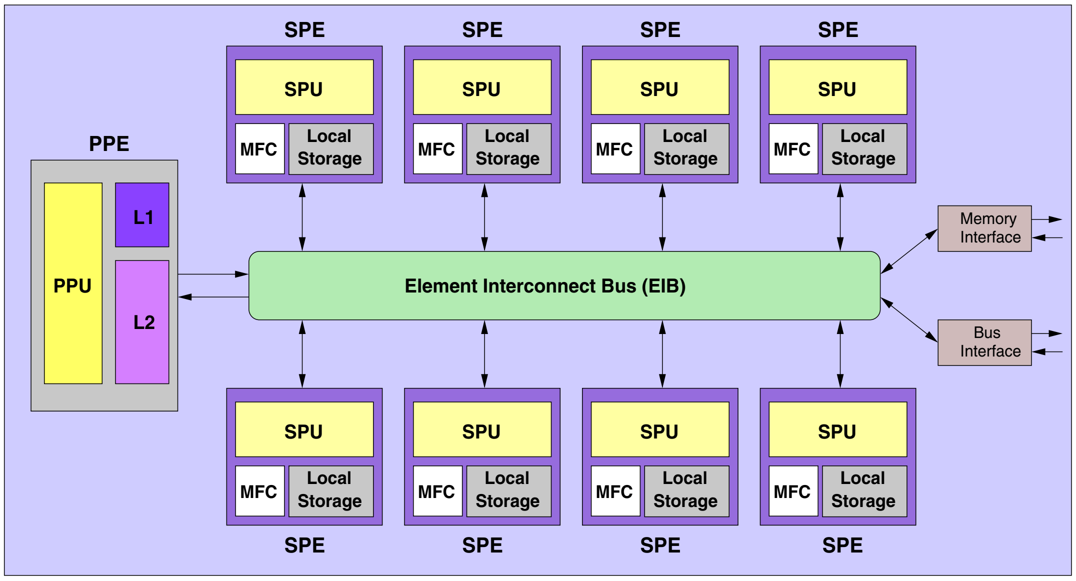

Figure 1. Architecture of the Cell B.E. processor.

vertices), and the coefficients (for the equation of the plane defined by the vertices) of each face. This is the most consuming data structure in raycasting.

VF-Ray improves cache performance by keeping in memory only the face data of the traversals of a set of neighboring rays. The decision of which rays are in this set is guided by visible faces computation. The pixels under the projection of a given visible face have high probability of reusing the same set of faces, for their traversal throughout the volumetric data. This set of pixels is called **visible set**.

In a preprocessing step, the volume data is read and a set of data structures is created in memory. This step is done only once, for all different point of views. The data structures created in this preprocessing step are:

- L_v: a list of all vertices;
- L_c: a list of all cells, where each cell c has a pointer to the cells that share a face with c, describing the connectivity of the cells;
- L_{ext}: a list of external faces of the volume.

The rendering of a single point of view starts by executing the operations to rotate the data in the axes x, y, and z, according to the angle of vision. After that, L_{ext} is traversed to determine the visible faces of the data. The visible faces are the external faces whose normals make angles greater than 90^o with the viewing direction.

Having computed all the visible faces, algorithm 1 shows the steps performed. For each visible face f_v, the algorithm projects f_v on the screen in order to define the visible set of f_v. For each ray r that corresponds to a pixel p in the visible set, the algorithm has to compute the entry and exit point, e_{in} and e_{next} of the ray in each cell. The exit face, f_{next}, is computed based on the entry face, f_{in}, and the connectivity stored in L_c. Every time a new face is traversed, its coefficients are saved in a face buffer, and the lighting integral from e_{in} to e_{next} is computed, using an optical model. This step computes the contribution of the cell in the color and opacity of pixel p.

Algorithm 1 Main loop of VF-Ray

```
1:  Project $f_v$ on the screen
2:  for each ray $r$ in visible set of $f_v$ do
3:      $f_{in} \leftarrow f_v$
4:      $e_{in} \leftarrow$ intersection entry point in $f_v$
5:      repeat
6:          $f_{next} \leftarrow$ FindNextFace ($f_{in}$)
7:          $e_{next} \leftarrow$ intersection point in $f_{next}$
8:          Save $f_{next}$ parameters in the face buffer
9:          Compute the lighting integral from $e_{in}$ to $e_{next}$
10:         $f_{in} \leftarrow f_{next}$ and $e_{in} \leftarrow e_{next}$
11:     until $r$ exits the data
12: Clear the face buffer
```

Previous results for VF-Ray [28] showed that it spent only from 1/3 to 1/6 of the memory used by the traditional raycasting approaches, with comparable performance.

VI. Implementing VF-Ray on the Cell BE

The main challenges in implementing VF-Ray on the Cell BE processor are: (i) Data structures have to be manually 128 bytes aligned; (ii) The time-consuming computations need to be SIMD-ized; (iii) The data structures that hold the dataset to be rendered, do not fit in the local storage of each SPE, so DMA transfers of the next cell have to be orchestrated; (iv) Performance-critical *if-then-else* branches that execute on the SPEs need to be eliminated.

VF-Ray is implemented as a C++ application, with more than 20 classes and over 10,000 lines of code. Given its complexity, the Cell implementation was done in a step-by-step approach, reusing as much of the original code as possible.

The first step is to adjust the data structures. All data structures were aligned in memory. The structures with an arbitrary number of elements were transformed to 128-bit vectors. For example, the `class Point`, that comprises 4 float elements (the coordinates of each point in the grid along with their scalar values, α), was transformed into a 128-bit vector. In the same way, the `class Cell`, with four integers being indices for its vertices and another four integers being indices for its four neighboring cells, were replaced by two 128-bit vectors.

After that, we accomplished the following steps: explored SIMD facilities, parallelized ray computations on the SPEs, and orchestrated DMA transfers. Each of these steps will be explained in further details next.

A. Exploring SIMD Facilities

In the rendering process, there are some functions that can be highly optimized by the use of the vector functions available on the Cell BE processor. The function that normalizes and centralizes the data according to the screen dimensions is one of them. This function conveys a translation followed by the multiplication of all the points by a normalization factor. This factor is the inverse of the diagonal of the bounding box, surrounding the data in space. Both translation and multiplication operations of this function were gathered in one single vector instruction: `vec_madd`.

In the function *FindNextFace*, we have to compute the coefficients of all the three remaining faces of the cell in order to find the exit face of the ray, f_{next}. This process was vectorized by computing all three faces coefficients at once, with a couple of vector operations.

In the lighting integral function, we explored the fact that the same operation is done on the pixel's three color components, R, G, and B. So, these three components were packed in a vector, such that we can compute them in a single vector operation.

B. SPE Parallelization

The most time-consuming part of VF-Ray is the one presented in lines 5–11 of Algorithm 1. As each ray computation is independent in the raycasting process, we distribute the ray computations to the SPEs. However, the data structures L_v, and L_c, that describe the connectivity of the cells do not fit the SPEs local memories. So, we propose here a pipeline scheme in order to feed the SPEs with the appropriate data, and to avoid stalls.

Initially, the PPE determines the visible faces for the current point of view. The PPE projects each visible face on the screen creating its visible set. The pixels inside the visible set are, then, distributed to the SPEs for the ray computation. After that, the PPE is responsible to orchestrate the sending of rays to the SPEs. Algorithm 2 shows the code executed on the PPE.

Algorithm 2 PPE algorithm

```
1: Rotate the data for the current point of view
2: Find visible faces
3: for each visible face $f_v$ do
4:    Project $f_v$ on the screen and determine its visible set
5:    for each ray $r$ in the visible set of $f_v$ do
6:       Send $r$ to an idle SPE
```

The SPE is responsible for the computation of the entry and exit points of the ray in each cell, as well as the lighting integral. However, the SPE does not have in its memory the information about all the cells that a ray will intersect. For one cell, the SPE can compute the entry and the exit points of the ray, e_{in} and e_{next}. The face that contains the exit point indicates the next cell to be traversed. So, the SPE computation follows a pipeline. The SPE requests the information about the next cell to be traversed, and while the DMA is transferring it to the local memory, the SPE can compute the illumination integral for the pair e_{in} and e_{next}. This process is repeated, until the ray left the volume. In this case, the SPE will request another pixel to the PPE, and all the computation is repeated. Algorithm 3 detail the code executed on the SPE.

Algorithm 3 SPE algorithm

```
1: while there is a ray $r$ from the PPE do
2:    Find the entry face $f_{in}$ of $r$
3:    Find the entry point $e_{in}$ in $f_{in}$
4:    repeat
5:       $f_{next} \leftarrow$ FindNextFace ($f_{in}$)
6:       $e_{next} \leftarrow$ intersection point in $f_{next}$
7:       Request information about next cell
8:       Compute lighting integral from $e_{in}$ to $e_{next}$
9:       $f_{in} \leftarrow f_{next}$ and $e_{in} \leftarrow e_{next}$
10:   until $r$ exits the data
```

C. DMA Transfers

The parallelization proposed in section VI-B will only work correctly if the SPEs could be constantly filled with next cells. We solved this problem creating a pipeline, where the SPEs keep computing one pair of intersections, while the next cell is being transferred by the DMA. So, lines 7 and 8 in Algorithm 3 occur in parallel, overlapping computation and communication. This pipeline works fine because each DMA transfer moves data between main memory and the local storage of each SPE in an asynchronous fashion.

Furthermore, the DMA engine allows the request of multiple memory blocks in one operation. Each SPE has 32 communication channels that controls synchronization and signals with the PPE. The basic operations for requesting and sending data from/to the main memory are `mfc_get` and `mfc_put`, respectively.

D. Implementation Details

As the *FindNextFace* function is the most time-consuming function of the SPE code, we were careful about implementation details of it. Besides, the vectorization implemented, we also removed a performance-critical *if-then-else* branch in the coefficient computation, as branches have high latency in the SPEs. In the sequence below, we give an example of how to remove a branch in the Cell BE. The first piece of code is the original branch code, and the second one is the branch-free code:

```
if (value < 0)
    temp = p0;
    p0 = p1;
    p1 = temp;
```

```
test = !spu_cmpgt(value, zero);
temp = p0;
p0 = spu_sel(temp, p1, test);
p1 = spu_sel(temp, p1, spu_andc(all,test));
```

All variables are vectors. The command `spu_cmpgt` sets the variable test for one if $value < 0$ and zero, if not. The `spu_sel` command will select the first or second parameter, to attribute to the variable, on the left side of the equal sign, depending on the value of the test.

VII. Experimental Results

In this section, we evaluate the performance of our Cell implementation of the VF-Ray algorithm. Our experimentation platform is a Sony Playstation 3 (PS3) game console running Linux Fedora Core 9. The PS3 contains a Cell processor running at 3.2 GHz, with 256 MB RAM. On PS3, the Cell processor operates with only six out of the eight SPEs. The Cell approach of VF-Ray was written in C++, using SIMD extensions included with the SDK 3.1.

We have used seven tetrahedral datasets that are widely used in the literature: SPX, Blunt Fin, Oxygen Post, Delta Wing, Fighter, F117, and Torso. Table I shows the number of vertices, faces, and tetrahedra for each dataset, and the time taken, in seconds, to render the dataset in the PPE, using the vector unit (VMX), for an image with 4K×4K pixels.

A. Parallelization Performance

Figure 3 shows, for all datasets, the variation of the execution time when the number of SPEs increases from 1 to 6. As we can observe in this graph, the increase in the number of SPEs reduces significantly the execution time for all the datasets. Table II shows the speedups for these executions. The speedups are relative to the sequential execution in the PPE, using the vector unit (VMX). So, even considering that each SPE can perform 4 floating point instructions at the same time (because of the 128 bit vector units), the ideal speedup for using 1 SPE would be 2 (the PPE and SPE are computing), for 2 SPEs would be 3, and so on, since the PPE also can perform 4 floating point instructions at the same time. Sometimes, however, it is not possible to fulfill all the 4 floating-point units with computation. The lighting integral computation, for example, can only occupy 3 units, with the computation of the components R, G, and B of the pixel color.

Nevertheless, we obtained some super-linear speedups results. This occurs because the floating-point operations on the SPE are faster than the PPE. This, in conjunction with the SPE's large register file and implicit memory transfers, make applications to run faster on a SPE than a PPE [29].

Dataset	# Verts	# Faces	# Tets	Rendering Time (sec)
SPX	149 K	1.6 M	827 K	957
Blunt Fin	41 K	381 K	187 K	854
Oxigen Post	109 K	1.0 M	513 K	1768
Delta Wing	211 K	2.0 M	1.0 M	1330
Fighter	160 K	2.8 M	1.4 M	1090
F-117	48 K	480 M	240 K	431
Torso	168 K	2.1 M	1.0 M	921

Table I
Datasets description.

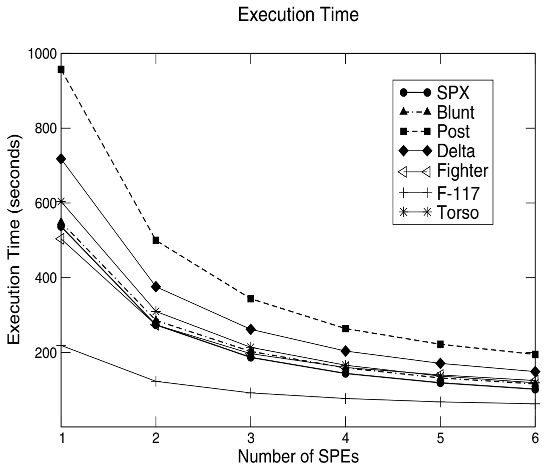

Figure 3. Execution times for all datasets.

As we can observe in the speedup table, the increase in the speedups are almost linear, indicating good scalability, and also indicating that the pipeline proposed could maintain the SPEs working almost all of the time. There is no contention to the main memory for two reasons. First, the PPE com-

Dataset	Number of SPEs					
	1	2	3	4	5	6
SPX	1.78	3.49	5.12	6.65	8.04	9.48
Blunt Fin	1.56	2.99	4.21	5.37	6.47	7.36
Oxigen Post	1.85	3.54	5.14	6.70	7.96	9.07
Delta Wing	1.85	3.54	5.08	6.52	7.78	8.93
Fighter	2.16	3.98	5.53	6.81	7.79	8.72
F117	1.97	3.50	4.68	5.60	6.34	6.84
Torso	1.52	2.97	4.30	5.55	6.72	7.74

Table II
SPEEDUPS FROM 1 TO 6 SPES.

putation is not a bottleneck, it computes only the next pixel to be processed. Second, the SPEs memory requirements are independent and do not overload the EIB (Element Interconnect BUS). The overall speedup is considerable and pays off the extra effort of the code changes.

In Table III we show, for all datasets and 6 SPEs, how the increase in the image size, has influence in the rendering execution time. As we can observe in this table, the execution time increases in the same proportion as the number of pixels increases for all datasets. This result confirms that our solution is not imposing extra overheads when more pixels are considered for computation.

Dataset	Image Sizes				
	256^2	512^2	$1K^2$	$2K^2$	$4K^2$
SPX	0.4	1.6	6.3	25.2	101.1
Blunt Fin	0.4	1.8	7.2	28.8	116.6
Oxigen Post	0.7	3.0	12.1	48.4	195.0
Delta Wing	0.6	2.3	9.2	36.8	149.5
Fighter	0.7	2.2	8.0	31.2	125.0
F117	0.2	0.7	3.0	13.1	63.0
Torso	0.4	1.8	7.3	29.5	119.0

Table III
EXECUTION TIMES (IN SECONDS) FOR DIFFERENT IMAGE SIZES.

B. Discussion

The execution time and speedup gains obtained are still modest when compared to the potential of the architecture to accelerate graphics applications. The impressive gains obtained by the raycasting implementation for regular datasets in the Cell BE proposed by Kim and Jaja in [26], however, cannot be reproduced for irregular datasets. Their implementation obtained huge accelerations with two special optimizations, approximation and refining, that removed memory latency overheads. These optimizations can only be accomplished when the geometry of the data is fixed. Furthermore, regular data does not require access to the data connectivity, reducing DMA transfers. The raycasting of irregular grids, on the other hand, is more challenging, since during ray intersection computation, the connectivity information has to be constantly searched. Making the memory latency overhead a great obstacle for performance.

Nevertheless, this is a preliminary version of a irregular grid raycasting implementation in the Cell BE. The three-level memory architecture, which decouples main memory accesses from computation and is explicitly managed by the software, increases the burden of programming, but, the small local store of the SPE did not pose a practical limitation. A thoughtful implementation would explore some benefits of this memory architecture, like long block transfers that can achieve higher bandwidth than individual cache-line transfers, and overlap of communication and computation scheduled by software. Our approach opens the door for future optimizations in the DMA transfer, by exploring prefetching of data from SPE and the reuse of the face data in the local storage.

Despite performance issues, the data format of the SPEs imposes another obstacle for graphics programming. Although, SPEs are fully IEEE-754 double precision compliant, for single precision data, the results are not fully IEEE-754 compliant (different overflow and underflow behavior, and support only for truncation rounding mode).

VIII. CONCLUSIONS

In this paper, we proposed a new implementation of the VF-Ray raycasting algorithm for irregular data that exploits the highly parallel architecture of the Cell BE processor.

As the Cell BE processor imposes a different programming model, our approach concentrates on reducing memory latency by efficiently transferring data to the SPEs, and on exploring the SIMD facilities of the cores. Our algorithm distributes ray computations to the SPEs driven by visible faces, and vectorizes the lighting integral operations inside each SPE. Our experimental results show that we can achieve good speedups on the PS3. In conclusion, to unveil the Cell BE full performance, careful programming is required. Nevertheless, under the proper implementation, Cell BE processor demonstrates good potential for implementing high-performance rendering.

For future work, we are working on optimizing our approach, and on a parallel implementation that explores a cluster of PS3.

ACKNOWLEDGMENTS

We would like to thank Prof. Thadeu Penna and the Complex System Group of the Institute of Physics at Fluminense Federal University for providing us full access to the PS3 used in our experiments.

REFERENCES

[1] R. Espinha and W. Celes, "High-quality hardware-based raycasting volume rendering using partial pre-integration," in *SIBGRAPI '05: Brazilian Symp. on Computer Graphics and Image Processing*, 2005, pp. 273–281.

[2] R. Marroquim, A. Maximo, R. Farias, and C. Esperança, "Volume and Isosurface Rendering with GPU-Accelerated Cell Projection," *Comp. Graphics Forum*, vol. 27, pp. 24–35, 2008.

[3] A. Maximo, S. Ribeiro, C. Bentes, A. Oliveira, and R. Farias, "Memory efficient gpu-based ray casting for unstructured volume rendering," in *IEEE/EG Int. Symp. Volume and Point-Based Graph.*, 2008, pp. 55–62.

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

[5] J. K. Lee and T. S. Newman, "Acceleration of opacity correction mechanisms for over-sampled volume ray casting," in *EGPGV '08: Symposium on Parallel Graphics and Visualization*, 2008, pp. 22–30.

[6] S. Marchesin, C. Mongenet, and J. Dischler, "Dynamic load balancing for parallel volume rendering," in *Eurographics Symp. on Parallel Graphics and Visualization*, 2006.

[7] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh, "Hybrid sort-first and sort-last parallel rendering with a cluster of PCs," in *Proc. of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[8] H. Yu, C. Wang, and K.-L. Ma, "Parallel volume rendering using 2-3 swap image compositing for an arbitrary number of processors," in *Proceedings of Proceedings of IEEE/ACM Supercomputing 2008 Conference*, November 2008.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[10] "Top500 site. `http://www.top500.org`." [Online]. Available: http://www.top500.org

[11] B. Lambronici, C. Bentes, L. Drummond, and R. Farias, "Dynamic screen division for load balancing the raycasting of irregular data," in *Proceedings of the IEEE Cluster*, 2009.

[12] F. Abraham, W. Celes, R. Cerqueira, and J. Campos, "A load-balancing strategy for sort-first distributed rendering," in *17th Brazilian Symposium on Computer Graphics and Image Processing*, 2004, pp. 292–299.

[13] C. Mueller, "Hierarchical graphics databases in sort-first," in *PRS '97: Proceedings of the IEEE symposium on Parallel rendering*, 1997, pp. 49–57.

[14] M. Roth, P. Rieb, and D. Reiners, "Load balancing on cluster-based multi projector display systems," in *14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2006, pp. 55–62.

[15] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva, "GPU-based Tiled Ray Casting using Depth Peeling," *Journal of Graphics Tools*, vol. 11.3, pp. 23–29, 2006.

[16] N. Fout and K.-L. Ma, "Transform coding for hardware-accelerated volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, November 2007.

[17] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Oumulti-gpu sort-last volume visualization," in *EGPGV '08: Symp. on Parallel Graphics and Visualization*, 2008, pp. 1–8.

[18] M. Meibner, S. Grimm, W. Straber, J. Packer, and D. Latimer, "Parallel volume rendering on a single-chip simd architecture," in *PVG 01: Symp. on Parallel and Large-Data Visualization and Graphics*, 2001, pp. 107–113.

[19] A. Adinetz, B. Barladian, V. Galaktionov, L. Shapiro, and A. Voloboy, "Abstract physically accurate rendering with coherent ray tracing," in *International Conference on Computer Graphics and Vision, GraphiCon*, 2008.

[20] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," *Computer Graphics Forum*, vol. 20, no. 3, 2001.

[21] S. Sakamoto, H. Nishiyama, H. Satoh, S. Shimizu, T. Sanuki, K. Kamijoh, A. Watanabe, and A. Asara, "An implementation of the feldkamp algorithm for medical imaging on cell," in *IBM White Paper*, October 2005.

[22] A. C. Chow, G. C. Fossum, and D. A. Brokenshire, *A Programming Example: Large FFT on the Cell Broadband Engine*, IBM, May 2005.

[23] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, "Ray tracing on the cell processor," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[24] K. O'Conor, C. O'Sullivan, and S. Collins, "Isosurface extraction on the cell processor," in *Seventh Irish Workshop on Computer Graphics*, 2006, pp. 57–64.

[25] B. Minor, G. Fossum, and V. To, *Terrain Rendering Engine (TRE): Cell Broadband Engine Optimized Real-time Ray-caster*, IBM, May 2005.

[26] J. Kim and J. Jaja, "Streaming model based volume ray casting implementation for cell broadband engine," *Scientific Programming*, vol. 17, no. 1-2, pp. 173–184, 2009.

[27] *IBM SDK for Multicore Acceleration for Fedora 9*, IBM, 2008.

[28] S. Ribeiro, A. Maximo, C. Bentes, A. Oliveira, and R. Farias, "Memory-aware and efficient ray-casting algorithm," in *SIBGRAPI '07: Brazilian Symp. on Computer Graphics and Image Processing*, 2007, pp. 147–154.

[29] I. Systems and T. Group, *Cell BE Programming Handbook Including PowerXCell 8i*, 2008.