2011 14th Euromicro Conference on Digital System Design

A Parallel Ray Tracing Architecture Suitable for Application-Specific Hardware and GPGPU Implementations

Alexandre S. Nery*‡, Nadia Nedjah¶, Felipe M.G. França*, Lech Jozwiak‡
*LAM – Computer Architecture and Microelectronics Laboratory
Systems Engineering and Computer Science Program, COPPE
Universidade Federal do Rio de Janeiro, Brazil
†Department of Electronics Engineering and Telecommunications – Faculty of Engineering
Universidade do Estado do Rio de Janeiro, Brazil
‡Department of Electrical Engineering – Electronic Systems
Eindhoven University of Technology, The Netherlands
E-mails: solon@cos.ufrj.br, A.S.Nery@tue.nl, nadia@eng.uerj.br, felipe@cos.ufrj.br, L.Jozwiak@tue.nl

***Abstract*—The Ray Tracing rendering algorithm can produce high-fidelity images of 3-D scenes, including shadow effects, as well as reflections and transparencies. This is currently done at a processing speed of at most 30 frames per second. Therefore, actual implementations of the algorithm are not yet suitable for interactive real-time rendering, which is required in games and virtual reality based applications. Fortunately, the algorithm allows for massive parallelization of its computations. In this paper, we present a parallel architecture for ray tracing based on a uniform spatial subdivision of the scene and exploiting an embedded computation of ray-triangle intersections. This approach allows for a significant acceleration of intersection computations, as well as, a reduction of the total number of the required intersections checks. Furthermore, it allows for these checks to be performed in parallel and in advance for each ray. In this paper we discuss and analyze an ASIP-based implementation using FPGAs and a GPGPU-based parallel implementation of the proposed architecture. The performance of both implementations are reported and compared. *Index Terms*—Ray Tracing, Parallel Architecture, Application Specific, ASIP, GPGPU, CUDA.**

I. Introduction

The Ray Tracing algorithm is a well-known rendering technique for generating high quality images from a 3-D scenario [1]. This algorithm is classified as a *Global Illumination Model*, among with others, such as *Path Tracing* and *Radiosity* [2]. In general, all these algorithms add more realistic lighting to 3-D scenes. For that reason, the ray tracing algorithm has been for some time topic of research as the next substitute for current Graphics Processing Unit (GPU) architectures [3], [4], since the latter are based on *Local Illumination Models* and so are not capable of producing such important effects directly. Therefore, developers must add those effects at the application level.

However, the main disadvantage of ray tracing is its high computational cost, even though the algorithm has a high parallelization potential. For instance, the performance of parallel implementations of ray tracing generally scales linearly to the number of available processors [5], [6]. Still, depending on the complexity of the 3-D scenario to be rendered, the algorithm execution can take several hours to produce a single image [2]. Thus, it is usually applied for off-line rendering, such as movie production [7], while for real-time rendering, as required in video-games, the algorithm is usually not applied. Hence, sequential implementations of ray tracing are not feasible.

Despite that, there are parallel implementations on Clusters [8] and Shared Memory Systems [9] that have been able to accelerate the algorithm, achieving real-time speed for some scenarios, applying pre-processing techniques, fast intersection computations [10] and spatial subdivision of the 3-D scene [11], [12], [13]. Parallel implementations in General Purpose Graphics Processing Unit (GPGPU) have also achieved substantial results [14], [15], [16]. The evolution and gradual overlapping of such graphics processing units to the general purpose niche in the recent decades is significant [17]. On the other hand, the *Stream Processor* architecture model of GPUs is optimized for graphics applications, with focus on the *Local Illumination Model*. For instance, control flow and recursion, which are often required in ray tracing, are very well performed by existing *Von Neumann* sequential architectures [18]. Also, graphics processing units are optimized for linear memory access pattern, while in Ray Tracing the access pattern is in general random. For those reasons, the latest architecture generation of GPUs from NVidia, known as *Fermi* architecture [19], have been improved to overcome such limitations, including cache hierarchy and recursion in hardware.

Also, there are consistent approaches to accelerate Ray Tracing with custom parallel architectures in hardware, as in [20], [21], [22], operating at low frequencies, such as 50Mhz and 90Mhz. Hence, the low frequency of operation is compensated by the parallelism of the custom design and several limitations can be overcome by a custom design. In fact, custom parallel architecture designs have also emerged as a promising alternative to achieve acceleration for several

978-0-7695-4494-6/11 $26.00 © 2011 IEEE
DOI 10.1109/DSD.2011.71

parallel applications that are mapped to hardware through Hardware Description Languages (HDLs) and Synthesis Tools [23]. In general, the target device is a Field Programmable Gate Array (FPGA), which can be used to prototype the design, and later an Application Specific Integrated Circuit (ASIC) can be produced, operating at much higher frequencies.

In this paper we propose and discuss implementation of our parallel custom macro-architecture for Ray Tracing, known as GridRT [24], in a GPGPU and compare its performance results against the ASIP-based GridRT hardware implementation in FPGA. The comparison results show that despite the lower performance of the custom ASIP architecture in FPGA, mainly due to 25 times lower clock frequency, the acceleration is significant and grows almost linearly with the number of ASIPs. These results also show that if the ASIP-based architecture would be implemented in an ASIC technology (instead of FPGA), comparable to the GPGPU implementation technology, then the performance of both implementations would be comparable.

The remainder of this paper is organized as following: Section II briefly introduces the ray-tracing algorithm. Then, Section III explains the GridRT parallel architecture. After that, Sections IV and V describe how the GridRT architecture is mapped to a hardware FPGA implementation and a GPGPU CUDA implementation, respectively. Finally, Section VI presents some performance results for both implementations and compare them, while Section VII draws the conclusion of this work.

II. Ray Tracing

The Whitted-style ray tracing algorithm [25] is briefly presented in Algorithms 1, 2 and 3, each one describing a different stage of the Ray tracing computation. Further details can be found in [1], [2].

In Algorithm 1, primary rays are created according to the *Virtual Camera* specifications, such as the viewplane *width* and *height*, as well as the camera position and view direction. Each primary ray corresponds to a pixel of the viewplane, where in the end of the computation the image will have been captured. Once the virtual camera has been setup pointing towards the 3-D scene, intersection checks are performed against each ray at a time, as in Algorithm 2. Notice that more than one intersection can be found for a single ray and, thus, the intersection that is closest to the ray origin must be selected. Otherwise, objects that are further from the observer's eye can mistakenly appear in front of the correct ones in the final image.

If an intersection is determined for a given ray, a corresponding *secondary ray* may be generated heading towards a new direction, according to Algorithm 3. Such secondary ray is going to be created depending on the properties of the intersected object's surface, whether it is specular or transparent. Intersection and shading computations are an essential part of the algorithm [26], [27].

Algorithm 1 Ray Tracing primary rays.

Require: scene, ray, depth
Ensure: pixel color
1: $viewplane \leftarrow setupViewplane(width, height)$
2: $camera \leftarrow setupCamera(viewplane)$
3: $rays \leftarrow generateRays(camera)$
4: $depth \leftarrow 0$
5: **for** $i = 1$ to viewplane's width **do**
6: **for** $j = 1$ to viewplane's height **do**
7: $image[i][j] \leftarrow$ **trace**$(scene, rays[i][j], depth)$ {**trace** function call}
8: **end for**
9: **end for**

Algorithm 2 Determining the closest intersection point.

Require: ray, depth, scene
Ensure: color
1: **if** $depth > max\_depth$ **then** {recursion control}
2: Black {default background color}
3: **else**
4: find closest intersection to the ray origin
5: **if** there is intersection **then**
6: compute point of intersection p
7: **shade**$(ray, p, scene, depth)$ {**shade** function call}
8: **end if**
9: **end if**

Also, for each intersection point, a *shadow ray* will be created towards each light source, to determine if the surface at the point of intersection receives direct illumination from one of the light sources, as in Algorithm 3. If so, Phong's shading model is applied. Otherwise, a crude approximation is performed based on the ambient component of Phong's model [27].

Finally, having computed a *secondary ray*, the **shade** procedure (Algorithm 3) recursively calls the **trace** procedure (Algorithm 2) to determine successive intersections against the new ray and different objects, until no further intersections are found. Then, at each level of recursion, the information of each intersection point is collected together with the intersected object properties. All those informations are merged into one single pixel color, which corresponds to one primary ray and its successive secondary rays.

III. The GridRT Macro-Architecture

The GridRT architecture [24], [21] is a massively parallel approach to intersection checks in ray tracing. It is inspired in the Uniform Grid spatial subdivision of the scene [28], which splits the 3-D scenario into 3-D regions of equal size, known as *voxels*. Each voxel contains a list of the 3-D objects that are inside or partially inside the voxel boundaries, as depicted in Fig. 1. Usually, an 3-D object is composed of several triangles [2]. Only those voxels that are pierced by a given ray have their objects (triangles) tested for intersections, greatly reducing the

Algorithm 3 Determining the corresponding pixel color and then recurse.

Require: ray, point of intersection, scene, object index, depth
Ensure: color

```
for each light source l do
    compute shadow ray sr[l]
    find intersections against the shadow ray
    if no shadow ray intersection then
        color ← phong's shading model
    end if
end for
if object surface at p is specular then
    compute reflection ray r_2
    color ← color + trace(scene, r_2, depth + 1)
else if object surface at p is transparent then
    compute refraction ray r_2
    color ← color + trace(scene, r_2, depth + 1)
end if
color
```

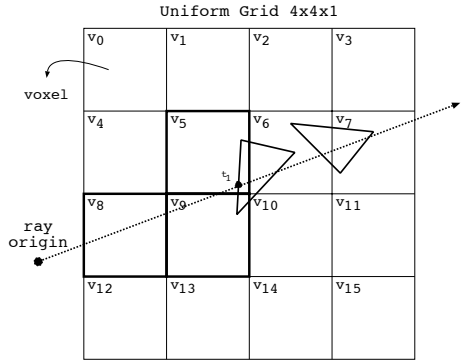

Fig. 1: The Uniform Grid sequential traversal.

number of intersection checks. Also, once an intersection is determined, no further voxels need to be visited for the given ray, since every other intersection cannot be smaller than the given one.

A. Parallelism

The standard uniform grid intersection algorithm proceeds sequentially, starting the search for intersections from the voxel that is closest to the ray origin to the furthest voxel, until an intersection is found or until the furthest voxel is reached without any results, as in Fig. 1. On the other hand, the GridRT macro-architecture maps each voxel onto a Processing Element (PE), responsible for computing intersection checks within its list of scene objects. Thus, all the PEs that are pierced by a ray are going to compute intersections in parallel along the same ray, as in Fig. 2. For that reason, it is necessary to discover which PE holds the result that is closest to the given ray origin.

One naive solution is to exchange the results between every PE that has been processing the same ray. This solution would require every PE to synchronize and, hence, wait for the others to finish their computation until they could exchange their

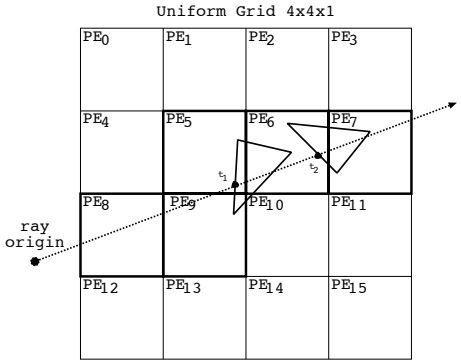

Fig. 2: Parallel intersection checks.

results. Instead, the GridRT architecture uses the traversal order that is inherited from the uniform grid traversal algorithm to determine the closest result. Therefore, every PE is aware of its position, based on the traversal list for a given ray. For example, in Fig. 2, the traversal list is L(8,9,5,6,7), from the closest to the furthest voxel (PE). Notice that each ray produces its own list of traversal, although some rays may traverse the same set of PEs. So, based on its position in the list, a PE can take one of the following actions:

- First in the list: if the first PE in the list finds an intersection, every successive PE in the list can abort its own computation. Thus, the first PE sends an interrupt message to the following in the list, which then aborts its computation and forwards the message to the next PE in the list, until the last PE is reached.
- Last in the list: if the last PE in the list finds an intersection, it must always wait for the previous PEs in the list to finish their computation before it can assume to have the closest result. Thus, the last PE must wait for a feedback message from the previous PE or an interrupt message as well, in order to take a decision upon its result.
- Middle of the list: if a PE that is located in the middle of the list finds an intersection, then it can send an interrupt message to the following PEs in the list, but must also wait for a feedback or interrupt message from the previous PE, before it can assume to have the closest result.

At the end of the computation of one ray, only one PE will remain active, while all the others either have finished their computation without obtaining any contributing result or have been aborted. In that way, it is no longer necessary to wait for every PE to finish the computation and exchange results. Parallel intersection checks can be performed and the result correctness is guaranteed. Moreover, the macro-architecture is not restricted to a specific target micro-architecture implementation of PE computations and, hence, can be mapped to a broader range of many-core architectures, such as application-specific instruction set processors (ASIPs) or GPUs, as it will be shown later in the paper.

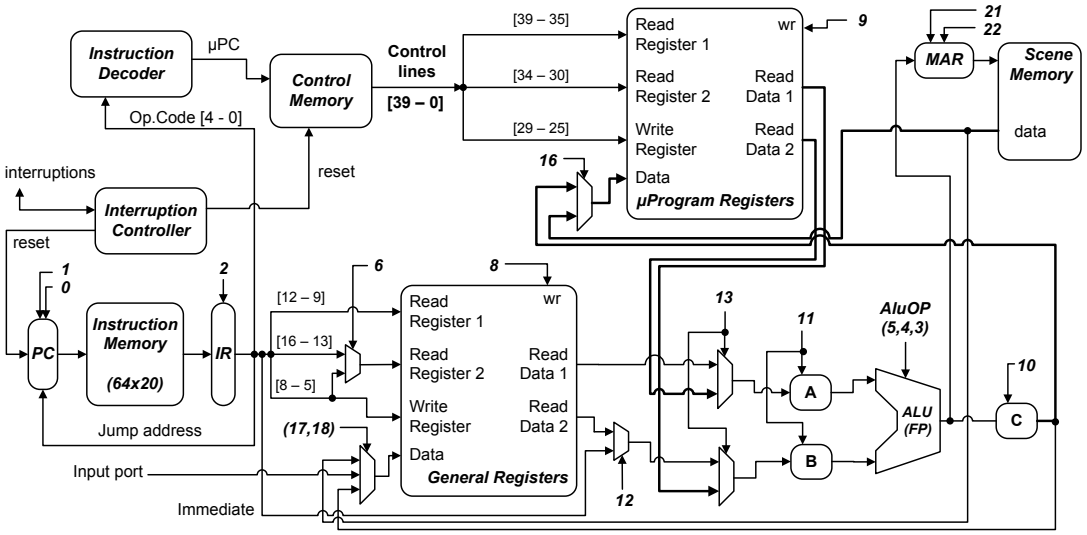

Fig. 3: The Processing Element processor. Each number is a circuit-level operation.

IV. GridRT with ASIP

To implement GridRT architecture presented in the previous section each PE can be implemented as a simple and very efficient ASIP (programmable hardware accelerator), specified in hardware description language (e.g. VHDL). Thus, the datapath of the PE depicted in Fig. 3 is controlled by a straightforward controller and every PE is responsible for computing intersection checks within its piece of scene data, which is stored into the *Scene Memory*. The *Control Memory*, where the microprogram is stored, has access to a dedicated register file. Such register file is available for storing intermediate results of a special instruction dedicated to ray-triangle intersection checks.

Every PE is connected to its direct neighbor through two interrupt lines. The first one is dedicated to the interrupt signals while the second to the feedback signals, as described in Section III-A. The path of both interrupt signals is selected for every new ray, based on its traversal list. Once all PEs are connected, the group of PEs acts as a massively parallel intersection co-processor for ray tracing, as depicted in Fig. 4 for a co-processor example with 4 PEs. Thus, the co-processor receives a ray together with its corresponding list of traversal and activation signals for each PE. Then, when the ray-triangle intersections have finished, the co-processor returns the closest result to the given ray origin.

The GridRT co-processor can be controller by any main processor. In our case it is a MicroBlaze RISC microprocessor IP from Xilinx [29], which can be synthesized in FPGAs, as shown in Fig. 5. Such connection is made through a Fast Simplex Link (FSL) channel [30], a low-latency point-to-point communication link available in MicroBlaze. Therefore, the MicroBlaze microprocessor executes the ray generation and the ray traversal algorithms, providing the necessary input data to the co-processor. The result for each ray is then read

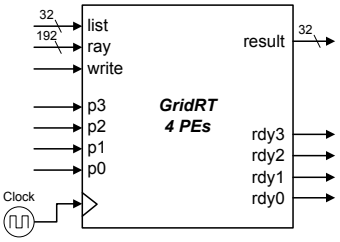

Fig. 4: The GridRT co-processor, with 4 PEs.

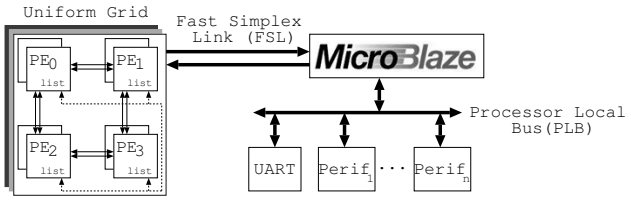

Fig. 5: The MicroBlaze microprocessor connected to a GridRT co-processor.

from the co-processor and transmitted to a host processor via a UART interface [31], for post-processing and visualization.

A. Parallelism and Interrupts

Following the parallel model of the GridRT architecture in Section III, parallelism is achieved through parallel intersection checks and exchange of signaling messages to determine the correct result. Thus, each PE is connected with its direct neighbors by two interruption lines. Signals are handled by a Interrupt Controller present in every processing element, as highlighted in Fig. 3. The first interrupt informs the current PE that a previous one has already computed an intersection

and, since no further intersection can be closer due to the traversal list order, the PE forwards the interruption and aborts its computation. The second interrupt deals with the situation when a PE has found an intersection, but has not received any feedback from a previous one. Thus, the second interruption signal informs the current PE that every former processor has finished the computation without results. At the end, only one remaining result prevails, while others had finished or were aborted. State machines are responsible for detecting an intersection result within the PE and building the interrupt signals path, as depicted in Fig.6. Simple multiplexers are used to select the interruption path based on the traversal list received together with the ray data. Thus, each PE knows what is the next PE and the previous PE according to its position in the list.

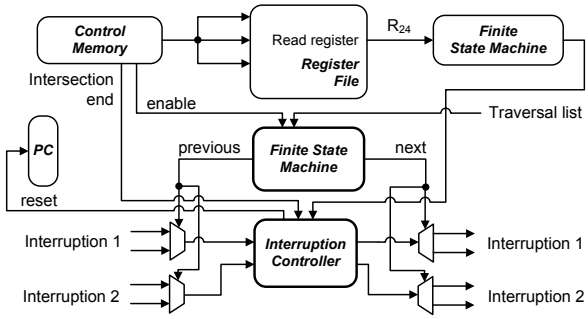

Fig. 6: Interrupt Controller detailed datapath.

V. GridRT in GPU

While the GridRT implementation in ASIPs maps each PE onto an ASIP, the counterpart GPU implementation maps each PE onto a Block of threads, that in turn is organized as a Grid of Blocks, according to the Compute Unified Device Architecture (CUDA) [17]. Such CUDA architecture model aims at performing a massive number of floating-point calculations simultaneously. Thus, it can be used across a wide range of applications that can be parallelized under the CUDA programming model. Here, we concentrate on taking advantage of the CUDA paradigm to implement the GridRT parallel model described in Section III-A.

A. CUDA thread assignment

In the CUDA programming model, all threads in a grid execute the same *kernel* function. Thus, each thread is assigned a unique identifier to distinguish it from others. Besides, groups of threads are organized into blocks and, hence, have access to a fast local *shared memory* and can be synchronized using a *barrier synchronization* function. On the other hand, threads in different blocks cannot be synchronized via barriers. Each block is also assigned a unique identifier. In modern GPUs, depending on the configuration that is specified when a kernel function is launched, each block or thread identifier can have up to three dimensions (x, y, z). For example, if the data to be processed is organized as a matrix $M(x, y)$, threads can be organized in two dimensions $(ThreadIdx.x, ThreadIdx.y)$, so that they can be easily assigned to its corresponding matrix data.

At the architectural level of current generation of hardware, blocks of threads are assigned to Streaming Multiprocessors (SMs), each one consisting of up to 32 CUDA Cores, as shown in Fig. 7. Once a block is assigned to a SM, it is split into *Warps*, which are groups of 32 threads with consecutive identifiers. Each block can include up to 1024 threads. A *Warp* is scheduled for execution by a *Warp Scheduler*. Thus, if an instruction, say *i*, that is being executed is waiting for a previous one whose completion is delayed due to a required long-latency operation, then a different *Warp* may be selected for execution of instruction *i*. In that manner, the resources of an SM are better exploited and this is called *latency hiding*.

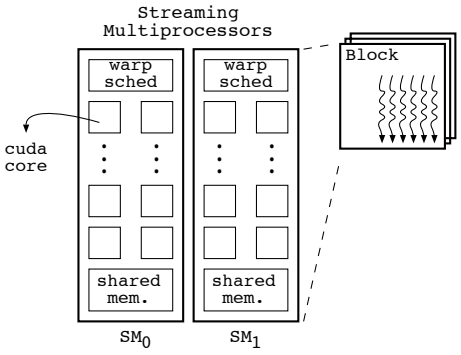

Fig. 7: Streaming Multiprocessors (SMs) organization, with each SM executing a blocks of threads.

B. GridRT–CUDA configuration

The GridRT implementation in CUDA maps a PE onto a block of threads, as depicted in Fig. 8. However, the threads of different blocks cannot coordinate theirs activities. Therefore, a different, yet similar, approach from the ones presented in Sections III-A and IV-A must be used to determine the result that is closest to the ray origin.

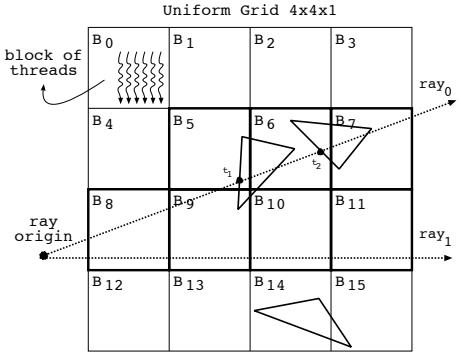

Fig. 8: GridRT–CUDA configuration.

A possible solution is to map each PE onto a thread, instead of blocks of threads. Hence, all the threads in charge of processing a ray could store their results in the shared memory and thereafter synchronize their work at the end of the given ray processing, checking which result is the closest to the ray origin. However, this solution has a major drawback: in order to make usage of the shared memory, only one block of threads, representing the whole GridRT, could be executed. All the other blocks would become useless, causing a waste of too many valuable resources. Therefore, we proposed a different approach. Instead of mapping each PE onto a thread, we decided to keep mapping each PE onto a block of threads and let the host processor to determine the correct result at the end of the whole computation. Now each ray has an array of results associated to it. The size of such an array corresponds to the maximum number of PEs, i.e. blocks, that can be traversed by a given ray. The size is determined by the total number of subdivisions according to each of the three axis (n_x, n_y, n_z) of the GridRT spatial structure, as defined in Eq. 1. For instance, considering the grid of Fig. 8, the maximum size of the array is $N = 7$, since the uniform grid subdivision is $n_x = 4$, $n_y = 4$ and $n_z = 1$.

$$N = n_x + (n_y - 1) + (n_z - 1) \quad (1)$$

When each block of threads has finished the intersection checks with respect to the the corresponding voxel, the result is stored in the array at the block associated entry. Thereafter, the block can proceed with the computation of a different ray, which also has a different array of results associated to it. In the end, the host processor copies the matrix of results from the GPU, wherein each row corresponds to the array of results computed by the block for a given ray. Considering once again the example in Fig. 8, the matrix is shown in Table I. Furthermore, each column contains the result that was computed by a block, also according to the list of traversal associated to each ray.

TABLE I: Matrix of results copied from the GPU by the host processor.

Ray	Array	Index						
Ray_0	$List_0$	B_8	B_9	B_5	B_6	B_7	-	-
	$Result_0$	-	-	t_1	t_2	-	-	-
Ray_1	$List_1$	B_8	B_9	B_{10}	B_{11}	-	-	-
	$Result_1$	-	-	-	-	-	-	-
	⋮							
Ray_n	$List_n$	B_k	B_{k+1}	B_{k+2}	-	-	-	-
	$Result_n$	-	t_i	t_{i+1}	-	-	-	-

Once all the intersections checks have been computed, the host processor checks each ray of the matrix, searching for the nearest intersection results. Once an intersection is encountered, it proceeds to the next ray, since the order of traversal guarantees that the encountered result is the closest to the ray origin. After processing all the results, the host creates a new set of rays based on the intersection points found by the previous set of rays. Thus, several calls to the GridRT *function kernel* produce a new set of intersection points for *secondary rays* and *shadow rays*. In the end, the results are processed by the host processor in order to created the final image from the ray traced 3-D scene, with shadows, reflections and refraction effects.

It is important to note that every block of threads is not only responsible of parallel processing of rays, but also parallel intersection checks within the block. Depending on the number of blocks and threads per block that are set at the kernel execution, one block may execute up to 1024 threads, split into *Warps* of 32 threads. Thus, each thread can be assigned to process one ray-triangle intersection in parallel with others, only if the number of triangles that belongs to the block is also compatible. In general, the more subdivisions are applied to the uniform grid spatial structure, which means a higher granularity of voxels (Blocks or PEs), the less triangles are present per voxel. In consequence, it became possible to assign a thread to each ray-triangle intersection check. Taking advantage of the fast access *shared memory* within a block, the intersection results are then stored in this memory and, in the end, the closest result to the ray origin in respect to the given block is stored in the matrix, as described in Table I for the grid of Fig. 8.

VI. Experimental results

The GridRT–FPGA hardware architecture based on ASIPs was described in VHDL, while the GPGPU version was described in CUDA v3.2. The hardware architecture was simulated in ModelSim XE 6.3c and synthesized to a Xilinx Virtex-5 XCE5VLX50T FPGA, through Xilinx ISE Design Suite 11.1. The area cost is presented in Table II, for a GridRT co-processor of 1, 2, 4 and 8 PEs, including the MicroBlaze microprocessor and its peripherals.

TABLE II: Area cost for 1, 2, 4 and 8 PEs, in contrast with old vs. new GridRT architecture.

Resources	GridRT–FPGA			
	1	2	4	8
Slice Reg	5012	7578	12368	21964
Slice LUT	6168	9254	15521	28138
LUT-FF	3184	3828	6851	14866
BlockRAM	31	32	48	80
DSP48Es	9	13	21	37

Synthesis Target: Virtex-5 XCE5VLX50T.

Parallel execution of up to 8 PEs was possible on a single Virtex-5 FPGA, running only at 50MHz. Each PE is equipped with four floating point units from Xilinx IP catalog [32], delivering an average of 10 MFLOPs per Processing Element. Each intersection check takes 307 cycles, involving 58 floating point operations. If all 8 PEs are executing in parallel, an estimated peak of 80 MFLOPs can be achieved.

On the other hand, the Graphics Processing Unit (Nvidia GTX 465) used to execute the GridRT kernel operates at

1215MHz, which is almost 25 times faster than the FPGA hardware GridRT implementation, due to the clock cycle limitations of the FPGA board. Also, such GPU contains up to 352 CUDA cores, each one equipped with a floating point unit, which is 11 times more floating point units than those available in all 8 PEs of the GridRT dedicated hardware altogether. The execution times for primary rays processing are presented in Table III, from 1 to 8 PEs and Blocks, for the dedicated GridRT hardware and for the GridRT kernel in GPU, respectively. The 3-D scene that was rendered is a low-polygon count of the Stanford Bunny (Low-res Stanford Bunny 3-D scene), for a resolution of 320×240 [33]. In Table III, all times are given in seconds.

TABLE III: Dedicated hardware and GPGPU kernel execution times.

Architecture	Number of PEs and Blocks			
	1	2	4	8
GridRT–FPGA	337	184	130	82
GridRT–CUDA	-	-	4.57	2.76

From Table III, it is possible to observe that as more processing elements are added to the architecture, the rendering time is almost linearly reduced. The GPGPU implementation is up to 30 times faster than the ASIP-based FPGA implementation. This can be explained by the generality and programmability overhead intrinsic of FPGA technology, resulting in 25 times slower clock than for GPGPU, and by the massive floating-point parallelism provided by the GPGPU. If the ASIP-based GridRT would be implemented in ASIC technology, instead of FPGA, comparable to the GPGPU technology, then it could most probably run with more than 25 times faster clock, and be faster than the GPGPU implementation, as its hardware is much simpler as the hardware of GPGPU. The execution times for configurations of 1 and 2 blocks of threads could not be measured, since the execution is terminated due to kernel execution timeout. This limitation can be avoided by including a second graphics card to the system, in order to leave the GPGPU dedicated to execution of CUDA kernels. Otherwise, the same GPU has to be shared between many applications of the Operating System and thus cannot execute long time CUDA kernels (up to tens of seconds).

Table IV presents further kernel execution times of up to 216 blocks of threads for the GridRT. As we can observe from Table IV, the GridRT–CUDA implementation achieves acceleration when up to 27 blocks of threads (or Processing Elements) are employed. Beyond that, the performance degenerates, as also depicted in Fig. 9.

The performance degeneration can be explained by the work granularity level that each block of threads is operating at. Parallel intersection checks are performed by all threads within a block, but *loops* and *conditional branches* dominate most part of the computation. It is well-known that such conditional constructions are not well suited for the *Stream Processing* model (Single Instruction Multiple Data, SIMD). Control flow

TABLE IV: GridRT–CUDA kernel execution times.

Architecture	Blocks of threads					
	1	2	4	8	12	18
GridRT CUDA	–	–	4.57	2.76	2.48	1.91
Architecture	Blocks of threads					
	27	36	48	64	125	216
GridRT CUDA	1.32	1.44	1.82	2.37	2.54	4.65

*All times are in seconds. Low-res Stanford Bunny 3-D scene.

has been for years optimized for *Von Neumann* architectures and are better executed by such. Also, the performance degenerates because more blocks of threads are competing to be executed by the streaming multiprocessors available in the Nvidia GTX 465 GPU. A dedicated GPU card or an even more powerful GPU with more CUDA cores would very likely achieve acceleration for more blocks of threads.

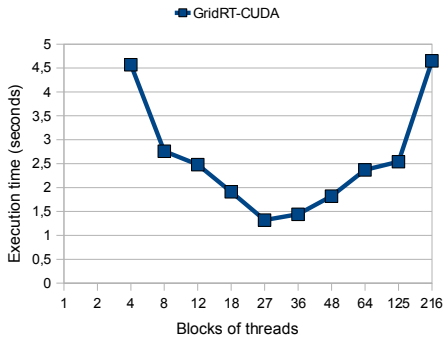

Fig. 9: GridRT–CUDA execution results.

VII. Conclusions

In this paper, two implementations of our massively parallel GridRT architecture for Ray Tracing are discussed: the ASIP-based FPGA implementation and GPGPU (CUDA). These two implementations are analyzed an compared regarding performance. The ASIP-based GridRT implemented in a single Virtex-5 FPGA can execute up to eight processing elements in parallel, running at 50MHz only. When implemented in an ASIC technology, instead of FPGA, it could most probably run more than 25 times faster. As more PEs are included, the better is the performance achieved. However, recent advancements in GPGPU architectures, such as in the Nvidia *Fermi* architecture, have enabled an efficient implementation of several parallel applications, including ray-tracing itself. Therefore, we mapped the GridRT architecture to GPGPU using CUDA, with minor modifications in the synchronization of results, performed by the host processor. The GPGPU implementation with 25 times faster clock achieved 30 times higher performance than the FPGA ASIP-based GridRT, which shows the architecture potential towards real-time ray-tracing.

One of the reasons for such speed up gain is explained by the processing power gap that exists between the FPGA programmable hardware implementation and the GPGPU. For

instance, the first is running at only 50MHz and can hold up to eight processing elements. The latter runs at frequencies up to 1215MHz and can execute many blocks of threads. In total, the Nvidia GTX 465 GPGPU that we have used in our checks have 352 CUDA cores, each one containing at least one floating-point unit. In total 11 times more floating point units than employed in a GridRT FPGA implementation of eight PEs. Thus, if the same implementation technology and processing power was available to the ASIP counterpart implementation, its performance would be comparable to that of GPGPU. The other reason for the speed up gap is that the ASIP-based GridRT can only perform parallel intersection checks for one ray at a time for now, while the GPGPU implementation can already perform parallel intersection checks for more than one ray.

Summing up, we demonstrated that the GPGPU implementation of our GridRT macro-architecture for ray-tracing is able to deliver a high performance, 30 times higher than that of our ASIP-based FPGA implementation. However, since the GPGPU implementation introduces more hardware overhead comparing to an ASIP-based ASIC implementation, the ASIP-based ASIC implementation is expected to have lower area and power consumption.

ACKNOWLEDGMENT

The authors are grateful to FAPERJ (Fundação de Amparo à Pesquisa do Estado do Rio de janeiro), CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) for their continuous financial support.

REFERENCES

[1] K. Suffern, *Ray Tracing from the Ground Up*, 1st ed. Natick, MA, USA: A. K. Peters, Ltd., 2007.

[2] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 3rd ed. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[3] H. Friedrich, J. Günther, A. Dietrich, M. Scherbaum, H.-P. Seidel, and P. Slusallek, "Exploring the use of ray tracing for future games," in *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*. New York, NY, USA: ACM Press, 2006, pp. 41–50.

[4] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, March 2007.

[5] J. Hurley, "Ray tracing goes mainstream," *Intel Technology Journal*, vol. 9, no. 2, pp. 99–107, Maio 2005.

[6] C. B. Cameron, "Using fpgas to supplement ray-tracing computations on the cray xd-1," *HPCMP Users Group Conference*, vol. 0, pp. 359–363, 2007.

[7] P. Christensen, J. Fong, D. Laur, and D. Batali, "Ray tracing for the movie 'cars'," *Symposium on Interactive Ray Tracing*, pp. 1–6, 2006.

[8] I. Wald, P. Slusallek, and C. Benthin, "Interactive distributed ray tracing of highly complex models," in *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*. Springer, 2001, pp. 277–288.

[9] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 37–46.

[10] S. Maxim, S. Alexei, and K. Alexander, "Ray-triangle intersection algorithm for modern cpu architectures," in *Proceedings of GraphiCon, 2007*, 2007, pp. 33–39.

[11] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," in *Computer Graphics Forum*, 2001, pp. 153–164.

[12] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes," in *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. Washington, DC, USA: IEEE Computer Society, 2003, p. 11.

[13] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*. New York, NY, USA: ACM, 2006, pp. 485–493.

[14] K. Ralovich, "Implementing and analyzing a gpu ray tracer," in *CESCG '07: Central European Seminar on Computer Graphics for students*, 2007, p. 6.

[15] M. Christen, "Ray tracing on gpu," Ms.C., University of Applied Sciences Basel, 2005.

[16] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM, 2005, p. 268.

[17] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[18] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a multicore architecture for real-time ray-tracing," in *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 176–187.

[19] Nvidia, "Whitepaper nvidia gf100," http://www.nvidia.com/object/IO_86775.html, 2010, last access: january, 2010.

[20] J. Schmittler, I. Wald, and P. Slusallek, "Saarcor: a hardware architecture for ray tracing," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 27–36.

[21] A. S. Nery, N. Nedjah, and F. M. G. França, "A massively parallel hardware architecture for ray-tracing," *International Journal of High Performance Systems Architecture 2009 - Vol. 2, No.1 pp. 26 - 34*, pp. 26–34, 2009.

[22] S. Woop, J. Schmittler, and P. Slusallek, "Rpu: a programmable ray processing unit for realtime ray tracing," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*. New York, NY, USA: ACM, 2005, pp. 434–444.

[23] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69–76, 2008.

[24] A. S. Nery, N. Nedjah, and F. M. G. França, "A parallel architecture for ray-tracing," in *IEEE Latin-American Symposium on Circuits and Systems - LASCAS'10*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 96–99.

[25] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[26] P. Shirley, *Ray-Object Intersections*, 1st ed. Natick, MA, USA: A. K. Peters, Ltd., 2000, pp. 34–38.

[27] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, pp. 311–317, June 1975.

[28] A. Fujimoto, T. Tanaka, and K. Iwata, *ARTS: accelerated ray-tracing system*. New York, NY, USA: Computer Science Press, Inc., 1988, pp. 148–159. [Online]. Available: http://portal.acm.org/citation.cfm?id=95075.95111

[29] Xilinx, "Microblaze processor reference guide," http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf, 2008, last acess: november, 2009.

[30] ——, "Fast simplex link v2.11b," http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf, 2009, last access: november, 2009.

[31] ——, "Xps uartlite," http://www.xilinx.com/support/documentation/ip_documentation/xps_uartlite.pdf, 2009, last access: november, 2009.

[32] ——, "Floating-point operator v5.0," 2008, last access: january, 2009. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

[33] S. C. G. Laboratory, "The stanford 3d scanning repository," http://www-graphics.stanford.edu/data/3Dscanrep/, 2009, last access: february, 2009.