Microprocessors and Microsystems 37 (2013) 739–749

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Parallel processing of intersections for ray-tracing in application-specific processors and GPGPUs

Alexandre S. Nery [a,c,*], Nadia Nedjah [b], Felipe M.G. França [a], Lech Jóźwiak [c]

[a] LAM – Computer Architecture and Microelectronics Laboratory, Systems Engineering and Computer Science Program, COPPE, Universidade Federal do Rio de Janeiro, Brazil
[b] Department of Electronics Engineering and Telecommunications, Faculty of Engineering, Universidade do Estado do Rio de Janeiro, Brazil
[c] Department of Electrical Engineering – Electronic Systems, Eindhoven University of Technology, The Netherlands

ARTICLE INFO

Article history:
Available online 15 June 2012

Keywords:
Ray tracing
Parallel architecture
Application specific
ASIP
GPGPU
CUDA

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 30 frames per second. Therefore, current implementations of the algorithm are not yet suitable for interactive real-time rendering, which is required in games and virtual reality based applications. Nonetheless, the algorithm allows for massive parallelization of its computations, which is a strong reason of further improvements. Also, we present a parallel architecture for ray tracing based on a uniform spatial subdivision of the scene that exploits 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, both running different 3-D scenes. The performance of both implementations are reported and compared. Furthermore, a second GPU has been included in the GPGPU-based implementation, running the same parallel architecture. Thus, primary rays are split among both GPUs for parallel execution and their performance are also presented and compared.

1. Introduction

High-fidelity computer generated images is one of the main goals in the Computer Graphics field. Given a 3-D scene, usually described by a set of 3-D primitives (e.g. triangles), a typical rendering algorithm creates a corresponding image by several matrix computations and space transformations applied to the 3-D scene, together with many per-vertex shading computations [1]. All these computations are organized in pipeline stages, each one performing many SIMD floating-point operations, in parallel. The Graphics Processing Unit (GPU) is also known as a *Stream Processor*, because of such massively parallel pipeline organization, that continuously processes a stream of input data through pipeline stages. In the final stage, all primitives are rasterized to produce an image (a.k.a. frame). In order to achieve real-time rendering speed it is necessary to produce at least 60 frames per second (fps), so that the change between frames is not perceived and interactivity is ensured. The *Streaming Processor* model of current GPU architectures can deliver enough throughput of frame rates for most 3-D scenarios, but at the cost of a lower degree of realism in each produced frame. For example, important *Global Illumination* effects like shadows and reflections must be handled at the application level, because the hardware is based on a *Local Illumination* model and, thus, is especialized in processing 3-D primitives [10] at a high rate.

Although the ray tracing algorithm [28] is also a computer graphics application for rendering 3-D scenes, the algorithm operates particularly in opposition to traditional rendering algorithms [1]. For instance, instead of projecting the primitives to the viewplane, where the final image is produced, the ray tracing algorithm fires rays towards the scene and traces their path in order to identify what are the visible objects, their properties and the light trajectory within the scene, through several intersection computations. Thus, the ray tracing algorithm is a well-known rendering technique for generating high quality images from a 3-D scenario [28]. This algorithm is classified as a *Global Illumination Model*, among with others, such as *Path Tracing* and *Radiosity* [1]. In general, all these algorithms add more realistic lighting to 3-D scenes, such as shadows and reflections. 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)

* Corresponding author at: LAM – Computer Architecture and Microelectronics Laboratory, Systems Engineering and Computer Science Program, COPPE, Universidade Federal do Rio de Janeiro, Brazil.
E-mail address: alexandre.solon@gmail.com (A.S. Nery).

0141-9331/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2012.06.006

architectures [8,21], 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, although the result quality is not as good as the ray-tracing counterpart.

However, the main drawback 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 [12,3]. Still, depending on the complexity of the 3-D scenario to be rendered and also on the number of rays that are being traced, the algorithm execution can take up to several hours to produce a single image [1]. Thus, it is usually applied for off-line rendering, such as movie production [6], 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 [32] and Shared Memory Systems [4] that have been able to accelerate the algorithm, achieving real-time speed for some scenarios, applying pre-processing techniques, fast intersection computations [15], and spatial subdivision of the 3-D scene [33,30,31]. Parallel implementations in General Purpose Graphics Processing Unit (GPGPU) have also achieved substantial results [25,5,24]. The evolution and gradual overlapping of such graphics processing units to the general purpose niche in the recent decades is significant [13]. Some stages of the graphic's pipeline, such as the *Vertex* and *Geometry* processing stages, have recently evolved to programmable *Shaders*, that can be programmed to perform different algorithms [13]. So, the GPU is no longer dedicated to run graphic related algorithms, but also general purpose parallel algorithms that can benefit from the massively parallel architectue of modern GPGPUs. For instance, Data Level parallel applications in general achieve high acceleration when mapped to GPGPUs, because these applications perform well in SIMD machines [21]. 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* architectures, because for years they have been optimized to improve the execution time of sequential applications. 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 [20], have been improved to overcome such limitations, including cache hierarchy and recursion in hardware. Still, thread divergence contributes to worsen the performance of parallel applications in GPGPUs [29].

Thus, there are consistent approaches to accelerate ray tracing with custom parallel architectures in hardware, as in [26,17,35], operating at low frequencies, such as 50 MHz and 90 MHz. 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 parallel applications that are mapped to hardware through Hardware Description Languages (HDLs) and Synthesis Tools [7]. 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 [16], in a GPGPU and compare its performance results against the ASIP-based GridRT hardware implementation in FPGA. The 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. Also, we further extend the experimental results of the GPGPU-based implementation, because the ASIP-based implementation is limited to the area and timing constraints of the FPGA. Therefore, in GPGPU, we were able to execute hundreds of blocks of threads with more complex 3-D scenes and higher image resolution (more primary rays). Furthermore, we included a second GPU working in parallel with the first GPU. In such configuration, the rays are distributed among both GPUs, achieving even higher acceleration rates.

The remainder of this paper is organized as following: Section 2 briefly introduces the ray-tracing algorithm. Then, Section 3 explains the GridRT parallel architecture. After that, Sections 4 and 5 describe how the GridRT architecture is mapped to a hardware FPGA implementation and a GPGPU CUDA implementation, respectively. Finally, Section 6 presents some performance results for both implementations and compare them, while Section 7 draws the conclusion of this work.

2. Ray-tracing

The Whitted-style ray tracing algorithm [34] is briefly presented in Algorithms 1–3, each one describing a different stage of the Ray tracing computation. Further details can be found in [28,1].

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.

Algorithm 1. Ray tracing primary rays.

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

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 [27,23].