Parallel Processing of Ray Tracing on GPU with Dynamic Pipelining

Mekhriddin Rakhimov Fazliddinovich
Computer Engineering Faculty, TUIT, Tashkent, Uzbekistan
Email: raximov022@gmail.com

Yalew Kidane Tolcha
Department of Computer Science, KAIST, South Korea
Email: yalewkidane@kaist.ac.kr

Abstract—**This article describes the technologies of parallel processing of ray tracing using central processing unit (CPU) and graphics processing unit (GPU). The one problem of parallel processing of ray tracing is imbalance among the pixels computation which leads to performance degradation. A serious disadvantage of ray tracing is performance. Other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform spatial anti-aliasing and improve image quality where needed. There are some problems with the possibility of realization of the parallel processing of ray tracing in stream processing on multicore processors with the required acceleration.**

Index Terms—**ray tracing, parallel processing, acceleration, dynamic pipelining, the workload balancing, CUDA**

I. Introduction

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena [1].

Different ways to construct the image may vary the speed of, as well as quality, realistic and beautiful newly constructed image. Naturally, methods are to paint a more realistic picture, and require large computational resources. Of course, we do not consider the known bad practices, which are slow, and draw badly. We want only find best way to the workload balancing on CPU and GPU. However, the three-dimensional scene is not only one of the geometric details; it is not conceivable without light, because otherwise it simply, we would not have seen. A Z-buffer method allows drawing only the geometry of the scene. What to do? The exact physical model of light propagation is very complex. We can talk about some approximations to natural light. Requires that the shaded places where not exposed to direct light rays, it was dark, away from sources of light-light. To create a realistic, in terms of illumination, the image of the scene began to use pre-calculated texture, so-called lightmap, containing the values of static objects in the scene lighting. This texture is applied in place with the usual texture of the material, and it darkens depending on the position of the object on the stage, his illumination. Naturally, this requires complete static scenes and light sources because of miscalculation lightmap are extremely long. Unlike the Z-buffer, ray tracing was originally designed for the construction of realistic images with complex lighting model. But the ray tracing is slow processing on CPU. To expedite the processing of ray tracing, we must use parallel methods.

In graphics literature, many techniques were proposed to accelerate the computation, including specific data structures and more efficient algorithms. In particular, we are interested in exploring the parallelism of ray tracing in this paper. The parallelization of ray tracing comes from the fact that each pixel has no interaction with the other pixels and its color computation is totally independent. The back tracing of lights can be done in parallel for all the pixels. CUDA is potentially suitable for this job: GPUs have hundreds of cores which can trace the lights of the pixels simultaneously. Therefore, a naive approach to parallelize the problem in GPU is creating a thread for every pixel to render the color in parallel [2]. But has some problems with the workload balancing. We solve this problem with the separation of the pixels of object and the pixels of background.

This paper assesses the resulting acceleration and way of workload balancing.

II. Statement of a Problem

Most high-quality, photorealistic renderings are generated by global illumination techniques built on top

Manuscript received April 14, 2015; revised July 13, 2015.

of ray tracing [3]. Ray tracing has lots of advantages over the earlier rendering method. The advantage of ray tracing is that it traces the lights bouncing among objects, which allows a much more realistic simulation of lighting over other rendering methods. A natural capability of ray tracing is to simulate reactions, reflections, and shadows, which are difficult using other algorithms. The computation can be naturally parallelized due to the independency among pixels. Ideally, if every pixel requires the same computation workload and GPU has N cores, the speedup of the parallelized ray tracing would be N, compared with serial implementation in a single core processor (assume their clock rate is around the same). However, this assumption is not true: some pixels require more computation than the others. Imagine that the light from a pixel has no intersection with objects, the color of the pixel is just the background color; the computation workload of this pixel is very light. On the other hand, if the light intersects with an object, then the color of the pixel should be the color of the object, which may include the color of other objects that have reflections on it. The computation workload for such light is heavier.

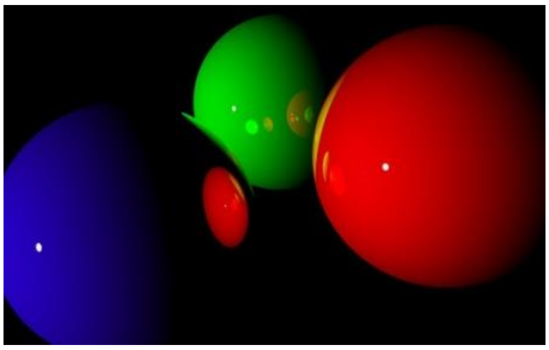
Figure 1. Comparison of workload for different pixels.

In Fig. 1, demonstrates the variety of workloads among pixels. Workload for pixels in the background region are minimum in the sense that the tracing will return after one step, as the light only "intersects" with the background. On the other hand, workload for pixels in the boxed region is high. The red ball, green ball and the blue ball reflect onto each other multiple times. When we trace the light starting from this region, the number of steps is much more than the background region. Unbalanced workload hurts the performance. The total execution time of the algorithm is bound by the pixels whose computing time is the longest. Thus, the benefit gained through parallelism is greatly limited. More importantly, adding more cores doesn't solve the problem as the "busiest" pixels are still computed by a single core. This means that future processors with more cores cannot improve the execution time, but only reduces the parallel efficiency, which is a very bad news for a parallel implementation [2].

Therefore, taking all these problems determine the following parameters: firstly we convert 3D scenes into 2D scenes and secondly the 2D scenes separate to pixels of background and pixels of objects.

III. The Concept of the Problem Decision

Firstly we considered questions of parallel processing at different pixels of the 3D scenes of ray tracing. In ray tracing processing systems operating in real time, a greater role is played by the acceleration computing processes. The use of GPU processors gives a significant increase in processing speed, but still very few algorithms and methods, especially in ray tracing for rendering images with computers, capable of working efficiently on multi-core processors in the mode of stream processing. The development of methods of ray tracing for rendering 3D images implemented at high speed on a new parallel system is necessary.

Stream processing in ray tracing technologies should be considered as new methods that include the following elements of the preparation and execution of rendering 3D scenes.

Adaptive partitioning is widely used in parallel computation to partition the job so that each partition has around the same workload. For the problem of ray tracing, ideally we would like to estimate, in reasonable time, the workload for the pixels so that we may partition the pixels into regions with various sizes that have the same workload. However, this is prohibitively difficult. The estimation can't be done without intensive computation, due to the complexity of light traveling among 3D objects [2]. People tried to develop different algorithm to identify objects to separate pixels with less computation from pixels with high computation. Such approaches introduce complex algorithm to find the block objects which incurs a lot of unnecessary computation. On the other hand, algorithms which are based on balance estimation statically and dynamically are introduced. In this approach even if the complex object identification is eliminated, groups of pixels at the boundary experience significant performance degradation.

In the ray tracing for rendering images the effectiveness of the developed parallel algorithm depends on the software implementation of the medium: the CPU, mechanisms for creating execution threads in the operating system, the number of threads [4]. Evaluating the effectiveness of ray tracing will test on GPU processors. GPU is a separate unit of a personal computer or game console, performing graphics rendering. Modern GPUs are very efficient process and display computer graphics. Due to the pipelined architecture specialized they are much more effective in the processing of graphical information than typical CPU. Graphics processor in modern display adapters used as three-dimensional graphics accelerator [5]. GPUs are becoming increasingly powerful and ubiquitous; researchers have begun exploring ways to tap their power for non-graphic or general-purpose (GPGPU) applications. The main reason behind this evolution is that GPUs are specialized for computationally-intensive and highly parallel operations—required for graphics rendering and therefore are designed such that more transistors are devoted to data processing rather than data caching [6]. For software development we used the architecture CUDA.

CUDA (Compute Unified Device Architecture) - software and hardware architecture of parallel computing, which can significantly increase the computational performance through the use of graphics processors from NVIDIA. CUDA SDK allows programmers to implement a special simplified dialect of the C programming language algorithms feasible on GPU NVIDIA, and include special features in the text of a C program. The CUDA architecture gives developers the discretion to arrange access to a set of instructions and graphics accelerator control his memory [7].

NVIDIA heralded its "Fermi" architecture, released in 2010 on its GTX 480 video card, as a major advance in parallel processing. It was based on a collection of four Graphics Processing Clusters (or GPCs), each of which contained a raster engine and four Streaming Multiprocessor (or SM) units. Each SM, in turn, contained 32 CUDA processing cores, 16 texture units, and a polymorph engine. The GTX 680's GPCs use a similar design, but with a couple of key differences. Each SM is now a "next-generation Streaming Multiprocessor", which abbreviated as SMX; each SMX contains 192 CUDA cores, for a total of 1,536 cores in the entire Kepler GPU—which suggests potential for considerably greater performance; and the polymorph engines have been redesigned to deliver twice of the performance of those used in Fermi, for what NVIDIA calls "a significant improvement in tessellation workloads". But because all those CUDA cores also run at a lower clock speed than Fermi's did, the GPU as a whole uses less power even as it delivers more performance [8].

Based on the new Kepler architecture dynamic Parallelism in CUDA enables a CUDA kernel to create and synchronize new nested work, using the CUDA runtime API to launch other kernels, optionally synchronize on kernel completion, perform device memory management, and create and use streams and events, all without CPU involvement.

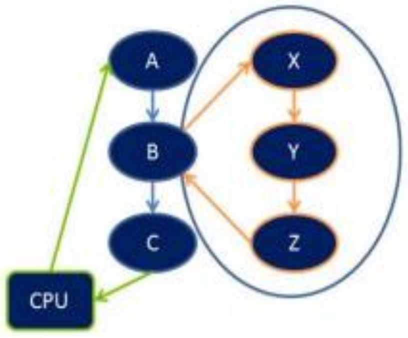

Figure 2. Dynamic parallelism in CUDA.

NVIDIA call the launching kernel the "parent", and the new grid it launches the "child". Child kernels may themselves launch work, creating a "nested" execution hierarchy. As shown in Fig. 2, CPU launches "parent" kernel (A B C). After this, kernel B in the GPU can also launches other "child" kernels (X Y Z). Launches may continue to a depth of 24 generations, but this depth will typically be limited by available resources on the GPU. All child launches must complete in order for the parent kernel to be seen as completed. For example in the Fig. 2, kernel C will not be able to begin execution until kernel Z has completed, because kernels X, Y and Z are seen as part of kernel B.

IV. Realization of the Concept

To implement the idea we will divide the work into some steps. So after converting 3D scenes into 2D scenes, we can process the pixels of object separately.

A. First Finding Objects

Most frequently in problems recognizers images are considered monochrome image that gives an opportunity to consider the image as a function of the plane (Fig. 3). If we consider a point set in the plane T, where the function x (x, y) expresses each pixel in the image of its characteristics - brightness, transparency, optical density, such a function is a formal record of the image.

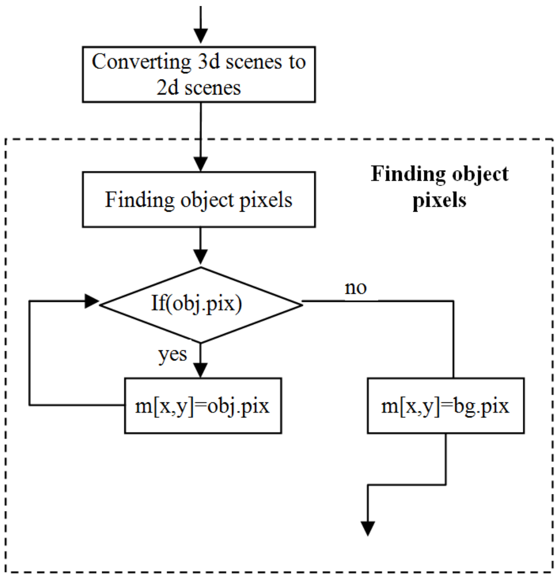

Figure 3. Block diagram of algorithm offending objects.

```
Color TracePath(Ray r) {
if (depth == MaxDepth)
        return Black; //Bounced enough times.
r.FindNearestObject();
if (r didn't hit anything)
        return Black; // Nothing was hit.
Material m = r.hitObject->material;
        Color emittance = m.emittance;
//Pick a random direction from here.
Ray newRay;
newRay.origin = r.hitPoint;
newRay.direction=
SampleRandomDirection (r.hitPoint);
Color reflected= TracePath(newRay);
// Compute the material interactionfloat
cos_theta = dot(newRay.direction,
r.normalWhereObjWasHit);
float attenuation = 2 * m.reflectance * cos_theta;
// Apply the Rendering Equation here.
return emittance + (attenuation * reflected);
}
```

The above pseudo code is taken from Wikipedia which shows naïve implementation of path tracing in this algorithm. It could be easy to group pixels after the nearest object is found but that is not simple rather it is complex. Different algorithm is being developed to reduce the complexity of finding object and it is still open research area.

B. *Static Block Assignment*

In ray tracing, the color in an image most likely changes gradually from pixel to pixel which indicates that the difference between workloads of adjacent pixels tends to be the very small. Therefore dividing the pixel in to groups making each group to have similar computation time will increase the performance as a whole. In static block assignment workload is more evenly distributed compared to the naive implementation. This contributes to the speedup compared to any straight forward implementation [2].

C. *Dynamic Block Assignment*

The dynamic implementation is similar to the static implementation, in the sense that the pixels are also divided into groups. Unlike the static method, which computes for pixels in predetermined groups, the dynamic approach allows the blocks of threads to fetch groups dynamically.

With dynamic scheduling higher throughput can be achieved with some overhead in scheduling. Such type of implementation can be improved by reducing the involvement of CPU in scheduling treads each time before Kepler NVDIA introduced Fermi architecture. In this architecture dynamic scheduling of threads is not possible. Every time when new scheduling is needed host device involve. This introduces significant performance degradation.

D. *Dynamic Pipeline Assignment*

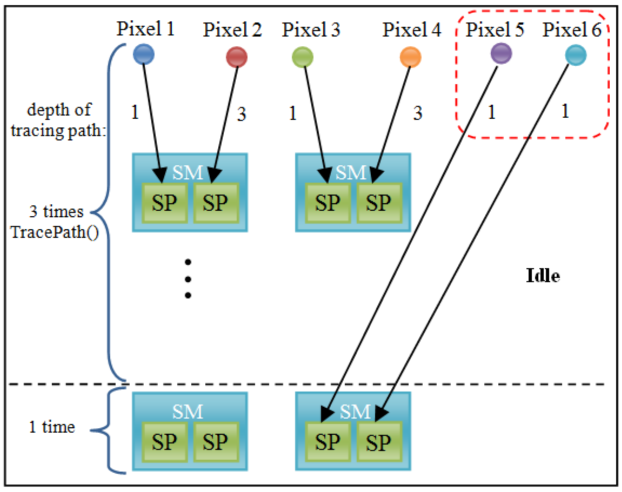

Figure 4. Dynamic scheduling.

In this article we introduce a new concept based on the dynamic scheduling capability of the new Kepler GPU architecture. In the speculative allocation, pixels are allocated based on their neighborhood. In this case pixels with different computation could be assigned to same block which makes the block to take maximum of the computation time. This can be observed in the diagram presented (the execution of parallel threads will take the maximum amount of time allocated for either of two). In the Dynamic Pipelining algorithm, our new approach, we dynamically assign threads a pixel for a single trace. In such approach if there is sufficient computation space for all pixels, it will take only the maximum tracing time. For example let's assume we have two SM which consist of two SP and 6 pixels to be calculated. On the baseline [1], a pixel which has the higher number of depth of trace path cause performance imbalance as shown in Fig. 4. Then, the number of processing of the TracePath is 4 times because the imbalanced computation is assigned to same block.

But if we dynamically regroup the threads upon the depth of computation for each pixel as shown in Fig. 5, we can reduce the number of processing of the TracePath.

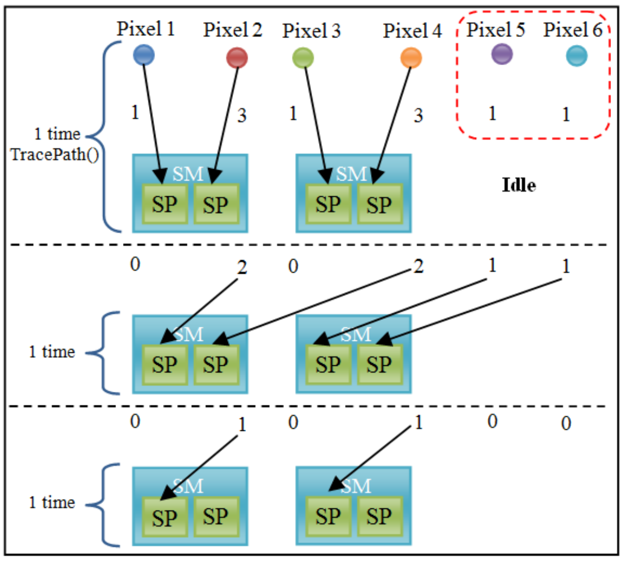

Figure 5. Dynamic pipeline scheduling.

V. Conclusion

By using CUDA Dynamic Parallelism, algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism can be more transparently expressed. Runtime solution can be assigned to the next grid. This will significantly reduce the time required to reschedule by minimizing the communication between CPU and GPU.

Due to the fact that, in recursive ray tracing, each ray tracing depends on the previous ray, the performance is bounded by the maximum depth.

Ignoring the scheduling and by properly track and design pixels distribution among the workers we expect the performance on the dynamic pipelining is limited only by the maximum depth. Therefore the maximum computation will be the equal to the time required to process the pixel with longest recursive array.

REFERENCES

[1] A. Appel, “Some techniques for shading machine renderings of solids,” in *Proc. AFIPS Conference,* NY, 1968, pp. 37-45.

[2] L. Chen, H. Das, and S. J. Pan, “An implementation of ray tracing in CUDA,” CSE 260 Project Report, Dec. 4, 2009.

[3] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, “Ray tracing on programmable graphics hardware,” in *Proc. ACM SIGGRAPH,* 2002, pp. 703-712.

[4] M. Musaev, “Parallel computing in digital signal processing,” *Scientific Technical and Information-Analytical //TUIT Bulletin*, no. 3, pp. 5-10, 2013.

[5] M. Ebersole, “Intro to GPU computing,” presented at the NVIDIA meeting, Santa Clara, CA, Sep. 25, 2012.

[6] NVIDIA. NVIDIA CUDA compute unified device architecture programming guide, version 1.1. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1 1/NVIDIA CUDA Programming Guide 1.1.pdf

[7] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, “Efficient computation of sum-products on GPUs through software-managed cache,” in *Proc. 22nd Annual International Conference on Supercomputing - ICS '08*, 2008, pp. 309-318.

[8] M. Murray. (Mar. 23, 2012). Nvidia’s Kepler architecture: 6 things you should know. [Online]. Available: http://www.pcmag.com/article2/0,2817,2402021,00.asp

Mekhriddin Rakhimov Fazliddinovich, from Uzbekistan, was born on August 20, 1988. He obtained Bachelor of Information Technology from the Tashkent University of Information Technologies (TUIT) in 2012, and Master in Applied Informatics from Tashkent University of Information Technologies (TUIT) in 2014. He published some papers in different journals. He is junior researcher and assistant-teacher at the department Computer systems, Tashkent University of Information Technologies (TUIT), Uzbekistan since 2014. Currently his interest is in the field of parallel image processing using a new parallel computing platform and programming models. He is currently researching parallel digital image processing and parallel processing of ray tracing technique for rendering images with GPU processors.

Yalew Kidane Tolcha, from Ethiopia, was born on October 11, 1987. He received his Bachelors from Addis Ababa University (AAU), Addis Ababa Institute of Technology (AAiT) with Electrical and Computer Engineering in 2011. He has worked as an assistant lecture at AAU, AAiT starting from same year he graduated. He is currently attending his Masters in Computer Science at School of Computing, Korean Advanced Institute of Science and Technology (KAIST). His research interests are Internet of things, big data analysis, and Cyber physical system.