Abstract

Central Processing Unit (CPU) has been the heart of any computer for many decades, and the functionality and higher processing power of CPU have been achieved by raising the CPU clock speed. However, the problems related to high power consumption, more memory bandwidth and heat dissipation force CPU designers to look for new architectural solutions, such as streaming compute model. Traditionally, most CPU architectures utilize Von Neumann Model, which in fact shows some inefficiency for massive computation, such as Ray-Tracing. With the streaming paradigm, GPU (Graphics Processing Unit) has begun to play more important role than ever before, as it can be optimized for stream computation. Furthermore, the extensive vector computation capability and the parallel graphics pipe-line provided by GPU become attractive to many scientific applications other than graphics. This growing use of a graphics card for stream computation leads to the implementation of GPU as a stream co-processor. In this research paper, I will primarily focus on the general concepts of ray-tracing, and the implementation of ray tracing on GPU as a specialized stream processor, and then I will examine the convergence between CPU and GPU for streaming computation, such as cell processors.

1 Introduction

Nowadays, various applications of computer graphics, such as 3D rendering, 3D modeling and 3D animation, have become common place. With the extensive use of computer graphics, the tradeoff between time and visual fidelity has become the primary concern of graphics community. In fact, rendering photo realistic imagery has never been easy; it is really a computationally expensive process. In 3D animated movie "Cars" [1], the average frame took 15 hours to render.

However, producing real-time photo-realistic rendering started to seem possible with the utilization of stream processing model, SIMD (single instruction multiple data) and multi-core architecture [2]. The gap between realistic and interactive (real time) rendering becomes narrower thanks to new architectural design such as cell processors.

1.1 Ray Tracing

Ray Tracing is a powerful yet simple 3D rendering technique which utilizes the physics of light. In real world, light is emitted from the light sources as photons. Ray tracing technique uses the same principle by modeling reflection and refraction of light by recursively following the path of light particle [3].

3D scene or world (i.e., a list of objects to be rendered) is usually rendered on 2D view window from a given viewpoint, usually called the eye or camera. Each point (pixel) in view window is caused by the simulated light ray reflected back from the objects towards the eye.

Although, ray tracing uses the physics of light, there is a key difference. In real world, every ray of light originates from light sources, or other objects as in reflected rays, towards the eye, but in ray tracing, all rays of light start from the eye, until the ray intersects the objects in the scene. However, because of the physical property of light, called reciprocity, the final result is the same.

Whenever, a ray from the eye intersects with an object (transparent, semi-transparent or opaque) in the scene, a new reflected or refracted ray is generated and traced until it reaches to the light source, thereby ray tracing is defined by recursive reflection and refraction.

Although the underlying principle is simple, in order to achieve high quality rendering, millions of rays need to be traced and calculated, which obviously is not an easy process. The situation can become worse, when there are many light sources in the scene, and indirect illumination caused by them.

1.2 Global Illumination

Global Illumination is another algorithm which can also capture indirect illumination. Ray tracing usually does not capture the effect of indirect lighting as it is an algorithm for direct illumination. Indirect lighting occurs when the light bounces off between two surfaces [3]. Although the effects of global illumination, such as color bleeding and caustics, are subtle, they are important for photo-realistic imagery. Therefore, many enhancement such as photon mapping techniques, which trace both rays from the eye and rays from the light source, and combine the path to produce the final image, has been incorporated into ray tracing algorithm.
2 Stream Model

Traditionally, Von Neumann Architecture has been the basic architecture for CPU, but it has been observed that Von Neumann style architecture creates bottle neck between CPU and main memory. In order to alleviate this bottle neck, many strategies has been incorporated into CPU design such as fast memory, cache etc. However, the primary focus on solving memory bottle-neck has resulted in lack of devotion on computation speed [4]. The most prominent example is the computation power of GPU. Even with the requirement of very little memory, GPU can perform very fast computation by exploiting parallelism and locality.

![Figure 2: Von Neumann Architecture](image)

Recently, stream processing and stream processor architecture have become popular. In fact, unlike traditional architecture, stream processors are designed to exploit the parallelism and locality in programs. Stream processors require tens to hundreds of arithmetic processors for parallelism and a deep hierarchy of registers to exploit locality. Streaming model also requires programs written for parallelism and locality.

![Figure 3: Stream Processor](image)

Stream programming model is based on kernels and streams. The kernel is the function which loads the data from input stream, computes and writes the data on the output streams.

2.1 Graphics Pipe Line

Even though, ray tracing can generate highly photorealistic imagery, most GPU employs different technique called rasterization, in which 3D rendering is achieved by shaders and various buffers, such as z- buffer, texture memory etc.

![Figure 4: Graphics Pipeline](image)

Figure 4 shows how a general graphics pipe-line works [3]. Generally, the 3D scene is fed into the hardware as a sequence of triangles (which is the most basic geometric figure from which more complex figure such as circles, polygons, curves etc., can be generated by approximation) Each triangle consists of vertices each of which stored the coordinates and color information. Vertex processor generally transforms vertices from model (3D) coordinates to screen coordinates (2D) by matrix transformation. Then, rasterizer transforms each vertex into fragments, which, in fact, are pixels with color and depth information (stored in memory to be rendered on the screen). Fragment processor modifies color of each fragment with texture mapping. Each fragment is then compared to the value stored in Z-buffer (depth buffer), and displayed through frame buffer on screen.

2.2 GPU stream processing

The most exciting feature of GPU is the programmable fragment processor, which uses SIMD parallelism to implement common operations, such as Add, Multiply, Dot Product, Cross Product and Texture fetches. However, fragment programs are not allowed for data dependent branching [3].
In fact, GPU’s fragment processor can be viewed as a specialized stream processor. The stream elements, in this case, are the fragments. Parallelism is achieved through running each fragment program in parallel on each pixel. Also through dependent texture lookup, which is a memory fetch for texture information for each fragment, GPU can be exploited for locality.

2.3 Ray Tracing on GPU

After explaining the general concepts of ray tracing and stream computation model, we will look into how ray tracing can be achieved on standard GPU. Since, ray tracing is an algorithm which can be computed per pixel, parallel ray tracing is readily achieved, by independently computation of each ray in parallel to produce the final image. This is one of the reasons why ray tracing was first implemented on clustered computers and massively parallel share memory supercomputers. In fact, interactive ray tracing is possible on those high end computers, which uses high computation power with wide memory bandwidth. However, in 2004, Purcell proposed a solution how ray tracing can be implemented on standard GPU, by utilizing stream processing model on fragment processor [3].

Actually, GPUs are designed for rasterization, not for ray tracing. Rasterization transforms all objects to be rendered into fragments and the fragments are displayed on the screen, depending on the depth information. However, according to Purcell [3], we can build a ray tracer on the fragment processor by exploiting its parallelism and locality for ray tracing. In his model, the input to fragments programs are pixels, each of which corresponds the pixel caused by each ray, and the fragment processor computes recursive ray tracing in parallel on that pixels, and then final image is displayed on screen.

Figure 6: Purcell's Streaming Ray Tracer

Figure 7 shows ray tracing is divided into several core kernels (functions), which are implemented on fragment processor by utilizing vector and matrix calculation.

Although Purcell proposed how ray tracing can be implemented solely on fragment processor, there are many limitations on actual implementation especially due to speed required for ray tracer. Moreover, Purcell’s design is heavily dependent on dependent texture lookups, which are poorly optimized for ray tracing.

In fact, many designs have been proposed to improve the speed of computations on GPU by improving texture lookups, adding more parallelism etc. The other solution is to build a custom piece of hardware, which can incorporate stream model. Many architectural designs have been proposed which can bridge CPU and GPU, and I will discuss about one of the design architectures which uses Cell Broadband Engine.
3 Cell Processors

Nowadays, there seems to be a convergence between CPU and GPU, with streaming model and multi-core architecture. While GPUs, which in fact are stream processors, can now provide more programmable features other than the manipulation of graphics, CPU designs start to incorporate streaming model through more and more cores with streaming functionalities.

In fact, cell processor architecture is to exploit this trend by incorporating functionality of both CPU and GPU. In this sense, it has huge potential for achieving interactive (real-time) ray tracing.

3.1 Cell Processor Architecture (CPA)

Cell Processor Architecture is different from conventional CPU architecture in many ways [2]. The most important difference is that Cell Processor employs different cores unlike conventional multi-core architecture which uses multiple copies of the same core. In this sense, CPA is a heterogeneous system which consists of 64 bit PowerPC core Element (PPE) and 8 synergistic co-processor elements (SPE), which are designed for streaming work load.

Therefore, each SPE can perform stream processing or parallel processing while PPE can perform non-parallel processing, such as synchronization between SPEs.

In fact, SPE is a processor on its own; it has its own 32bit RISC instruction sets, 256 KB local storage 128x128 bit SIMD register files. Most instructions on SPE are SIMD. Direct Access to Main Memory from SPE is not allowed; in fact, main memory access is achieved through asynchronous transfer.

3.2 Ray Tracing on Cell Processor

Since CP architecture is designed for SIMD processing, ray tracing can be achieved easily on CP [2]. In heterogeneous approach each SPE can run different kernels for each pixel. In homogeneous approach, each SPE run the whole ray tracer for different pixel. Actually, homogeneous approach is better as each SPE can independently run and reduce the use of common bus.

4 Conclusion

The limitations on conventional CPU design, the requirement for computation of massive data, the extensive use of computer graphics for various applications lead to the convergence between CPU and GPU. Traditionally, CPU is designed for general computation with high programming capability and GPU is optimized for SIMD vector operations with fast stream processing. Exploiting features of both CPU and GPU, as in Cell Processor, can enable us to perform really computationally expensive process such as ray-tracing without using high end computers, such as supercomputers. Moreover, the distinction between CPU and GPU become narrower, and GPU now becomes a co-processor for massive computations with simple operation.

References


