DATA PARALLELISM FOR RAY CASTING LARGE SCENES ON A CPU-GPU CLUSTER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TÜMER TOPCU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER SCIENCE
IN
COMPUTER ENGINEERING

MAY 2008

Approval of the thesis:

DATA PARALLELISM FOR RAY CASTING LARGE SCENES ON A CPU-GPU CLUSTER

submitted by **TÜMER TOPCU** in partial fullfillment of the requirements for the degree of **Master Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, **Graduate School of Natural and Applied Sciences**

Prof. Dr. Volkan Atalay
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Veysi İşler
Supervisor, **Computer Engineering Dept., METU**

Assist. Prof. Dr. Cevat Şener
Co-supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assist. Prof. Dr. Tolga Can
Computer Engineering Dept., METU

Assoc. Prof. Dr. Veysi İşler
Computer Engineering Dept., METU

Assist. Prof. Dr. Tolga Çapın
Computer Engineering Dept., Bilkent University

Assist. Prof. Dr. Tuğba Taşkaya Temizel
Information Systems Dept., METU

Dr. Onur Tolga Şehitoğlu
Computer Engineering Dept., METU

Date: 05.05.2008

ABSTRACT

DATA PARALLELISM FOR RAY CASTING LARGE SCENES ON A CPU-GPU CLUSTER

Topcu, Tümer

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Veysi İşler

Co-Supervisor: Assist. Prof. Dr. Cevat Şener

May 2008, 58 pages

In the last decade, computational power, memory bandwidth and programmability capabilities of graphics processing units (GPU) have rapidly evolved. Therefore, many researches have been performed to use GPUs in advanced graphics rendering. Because of its high degree of it parallelism, ray tracing has been one of the first algorithms studied on GPUs. However, the rendering of large scenes with ray tracing can easily exceed the GPU's memory capacity. The algorithm proposed in this work uses a data parallel approach where the scene is partitioned and assigned to CPU-GPU couples in a cluster to overcome this problem. Our algorithm focuses on ray casting which is a special case of ray tracing mainly used in visualization of volumetric data. CPUs are pretty efficient in flow control and branching while GPUs are very fast performing intense floating point operations. Using these facts, the GPUs in the cluster are assigned the task of performing ray casting while the CPUs are responsible for traversing the rays. In the end, we were able to visualize large scenes successfully by utilizing CPU-GPU couples effectively and observed that the performance is highly dependent on the viewing angle as a result of load imbalance.

Keywords: CPU-GPU Cluster, Ray Tracing, Ray Casting, Large Scene Rendering, Data Parallelism

ÖZ

ANA İŞLEM BİRİMİ-ÇİZGE İŞLEM BİRİMİ KÜMESİ ÜZERİNDE BÜYÜK SAHNELERİ IŞIN HESAPLAMA İÇİN VERİ KOŞUTLUĞU

Topcu, Tümer

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Veysi İşler

Ortak Tez Yöneticisi: Yrd. Doç. Dr. Cevat Şener

Mayıs 2008, 58 sayfa

Son birkaç yılda çizge işlem birimlerinin (GPU) işlem gücü, bellek bant genişliği ve programlanabilme yetenekleri hızlı bir şekilde gelişti. Bu gerçekler göz önüne alınarak GPUları ileri çizge sentezleme alanlarında kullanmak için pek çok araştırma yapıldı. Işın izleme, yüksek derecedeki koşutluğu sebebiyle GPUlar üzerinde çalışılan ilk konulardan birisi oldu. Ancak büyük sahnelerin ışın izleme ile gerçeklenmesi GPUnun bellek kapasitesini kolaylıkla aşabilir. Bu çalışmada ortaya konan algoritma, sahnenin parçalara ayrılarak ortaya çıkan her bir parçanın bir kümede bulunan CPU-GPU çiftlerine atandığı veri koşut bir yaklaşım kullanmaktadır. Algoritmamız daha çok hacimsel verilerin görsellenmesinde kullanılan ve ışın izlemenin özel bir durumu olan ışın hesaplama üzerine yoğunlaşmaktadır. CPUlar akış denetimi ve dallanmada oldukça verimliyken GPUlar yüksek miktarda kayan nokta işlemi gerektiren uygulamalarda oldukça hızlıdır. Bu gerçeklere dayanarak kümedeki GPUlar ışın hesaplama görevine, CPUlar ise ışınları hareket ettirme görevine atanmıştır. Sonuç olarak CPU ve GPUları verimli bir şekilde kullanarak büyük sahneleri görsellemeyi başardık ve performansın yük dengesizliği sebebiyle bakış açısına oldukça bağlı olduğunu gözlemledik.

Anahtar Kelimeler: CPU-GPU Kümesi, Işın İzleme, Işın Hesaplama, Büyük Sahne Görselleme, Veri Paralelliği

ACKNOWLEDGMENTS

I would like to express my inmost gratitude to my supervisor Assoc. Prof. Dr. Veysi İşler. His patience, vision, sweet communication and friendly approach is the key reason to vitalize this work. It is an honour for me to share his knowledge, wisdom and humanity.

I am grateful to Alphan Ş. Es, one of the most brilliant Ph.D. students of my supervisor. Without his support and guidance, this work would have been much harder to finish. I can't imagine myself dealing with those OpenGL and Cg errors without him. It was a great opportunity to work with him. I hope, I will be as wise as him one day.

We have been friends with Çağlar Ünlü since I started undergraduate school. Having him by my side during this thesis work was a big chance. Everyone would love to have someone who will make them laugh in desperate times.

Last but not least, I would like to thank my mother Prof. Dr. Saniye Topcu, my father Abdurrahman Topcu and my sister Dr. Pınar Topcu Yılmaz. They have always been by my side in my entire life and it was no different during my graduate program.

To everyone who have positively influenced my life. . .

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SYMBOLS

BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
GFLOPS	Billions of Floating Point Operations Per Second
GPGPU	General Purpose Computing on GPU
GPU	Graphics Processing Unit
POV	Persistence of Vision Raytracer
PLY	Polygon File Format
FBO	Frame Buffer Object
BSP	Binary Space Partitioning Tree

CHAPTER 1

INTRODUCTION

In computer graphics, photorealistic rendering methods are used to create images which imitate scenes from real life as much as possible. Achieving this goal is not easy as it requires simulating the interaction of light with all of the objects in the scene. This simulation process is called global illumination and its complexity increases enormously when there are transparent or reflective objects and particulate matters such as smoke or water vapor in the scene. There are several global illumination algorithms devised and the most popular ones are ray tracing [60], radiosity [15], Monte Carlo path tracing [25], photon mapping [24], beam tracing [19]. The internal of these algorithms may differ but all of them suffer from requiring a high number of floating point operations to be performed, making them hard to use in real time applications.

The everlasting evolution of computer technology gives hope to researchers for using global illumination algorithms in real time applications. Especially the recent advancements in graphics processing units (GPU) attracted too much attention making them a hot research area. Although GPUs were first designed for specific graphics computations like lighting, modeling, texturing, the state of the art GPUs are started to be used in other areas by developers and researchers. The efforts to use GPUs in these other areas are generalized under one name, general purpose computation on GPUs (GPGPU) which is now possible because GPUs no longer lack programmability.

Implementing widely used global illumination algorithm ray tracing on GPUs is one of the most popular subjects of GPGPU based researches. The main idea behind using GPUs for ray tracing is to exploit its enormous computation power. GeForce 8800 Ultra, the flagship of GeForce 8 Series, has a theoretical computation power of 345.6 GFLOPS [61] which is far beyond the computation power of today's CPUs. Unfortunately, using this raw power to its full potential is not an easy task since the architecture and programming model of a GPU is completely different from a CPU's.

Even if a GPU is effectively used, it is still not possible to perform ray tracing at high frame rates. The picture gets worse when we are talking about rendering a large scene because of the fact that today the memory available on most of the GPUs is limited to 256MB and ray tracing requires a large amount of memory. Based on these observations, the motivation and the goals of this thesis is given in the next section.

1.1 Global Illumination

Everything that we see in real life are rays of light (photon streams) cast by the sun or any other light source, bouncing around the detailed scenery of nature and finally hitting our eyes. The journey of light rays in nature includes absorbtion, reflection and refraction. Global illumination algorithms are used in computer science to imitate this process in the virtual world. This type of algorithms takes into account not only the light rays coming directly from the light source (direct illumination) but also the indirect illumination (reflection and refraction) to approximate the rendering equation (1.1) presented in [20] and [25] as close as possible.

$$L_o(x, \varrho) = L_e(x, \varrho) + \int_{\varpi} f_r(x, \varrho, \phi) L(x', -\phi)(\phi.\eta) d\delta \qquad (1.1)$$

- $L_o(x, \varrho)$ is the light outgoing in the direction ϱ at position x.
- $L_e(x, \varrho)$ is the light emitted in the direction ϱ at position x.
- $\int_{\varpi} ... d\delta$ is an integral over the hemisphere around position x of inward directions.
- x' is the position of the first surface hit by the ray leaving x in the direction of ϕ.
- $f_r(x, \varrho, \phi)L(x', -\phi)$ is the light coming from direction ϕ that hits position x and is reflected in the direction ϱ. The name for this function is Bidirectional Reflectance Distribution Function [36].
- $L(x', -\phi)$ is the light from the position x' in direction $-\phi$.
- $(\phi.\eta)$ is the attenuation of inward light where η is the surface normal at position x.

Basically rendering equation takes the law of conversation of energy as a basis and states that the outgoing light at position x in the direction of ϱ is the light that the surface emits plus the sum of all the incoming light that is reflected in the direction of ϱ.

1.2 Ray Tracing

Ray tracing is one of the most famous algorithms used to simulate the interaction of light with the environment. It is first introduced by Turner Whitted [60]. Basically ray tracing reverses what happens in the nature. In real life light rays are emitted whether we see them or not. However, in virtual environment imitating the same behavior would be a waste of it is not possible to generate and track infinite number of light rays. To overcome this problem, ray tracing uses the fact that the path taken by a light ray is reversible and starts generating lights having the eye as origin and than traces them. This way only the light rays arriving to the eye are taken into account.

Given the position of eye and an image plane, ray tracing starts with spawning rays, called primary rays, to each pixel on the image plane having the eye position as origin. After this process the nearest intersection in the clipped space is searched. If an intersection is found, a shadow ray is spawned to probe the light source and if the shadow ray is not obstructed by some other object, a direct illumination is added to the pixel's color. Also, depending on the type of the material, reflecting and/or refracting rays are spawned from this intersection point and these secondary rays are also traced. This recursive process is repeated for each ray until it hits a diffuse surface or misses all surfaces or recursion depth is greater than a threshold.

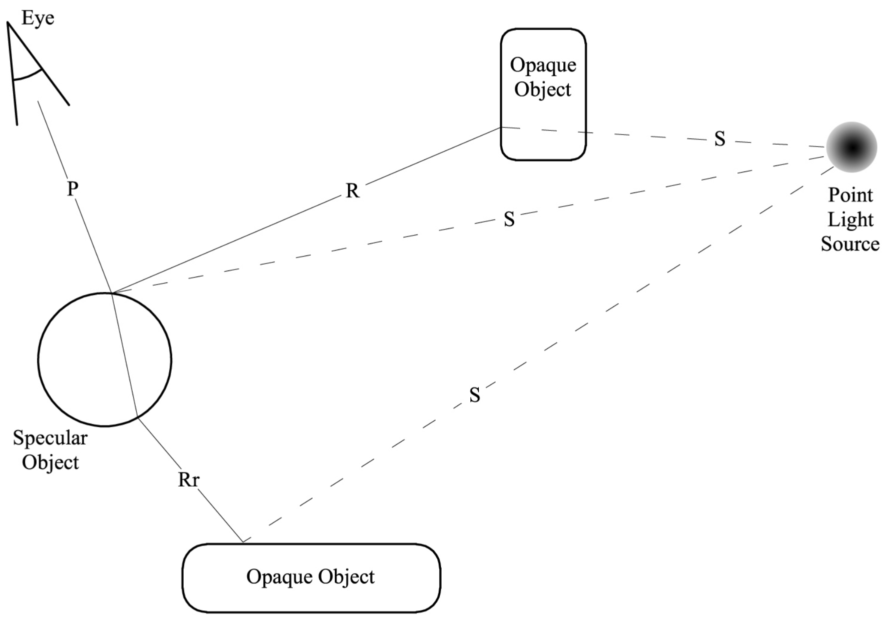


Figure 1.1: Ray Tracing Algorithm

To better explain the ray tracing algorithm, the execution flow is summarized in figure Figure 1.1. A primary ray (P) is spawned at the eye point and traced into the scene. The ray hits a specular object resulting in generation of a shadow ray (S), a reflected ray (R) and a refracting ray (Rr). The shadow ray reaches to the light therefore a direct illumination factor is added to the pixel's color value at which the intersection has occurred. The reflected ray hits an opaque object so no secondary rays are generated and also the shadow ray at this point is obstructed, therefore no color value is added to the final result. The refracting ray intersects with another object and the result of probing the light source is successful so the contribution of this intersection is also included in the final result. The process taking place in figure Figure 1.1 is also explained in Algorithm 1.

Ray tracing is considered to be a global illumination algorithm because it takes into account not only the direct illumination from the light sources but also the interaction of light between the objects, namely reflection and refraction. However, there are some other types of indirect illumination that ray tracing cannot handle. Caustics is one of these missing parts. It can be formally defined as envelope of light rays that has been reflected or refracted by specular surfaces previously, before hitting an opaque material. For example caustics can be formed by placing a glass of wine on the table. In this case the light rays that are passing through the glass will be refracted and focused to the table forming caustics as a result. You can also see caustics when you look at the bottom of a swimming pool. An example to the caustics generated by using POV [49] can be seen in Figure 1.2(c).

Another deficiency of ray tracing is diffuse reflection. Ray tracing focuses only on the specular reflections. However, what happens in real life is not limited to specular reflections. Even opaque/diffuse materials do reflect the light to the objects near them. For example when you put your hand close to the wall the wall's color will be effected by your hand's color. In Figure 1.2(a) there is no diffuse reflection from the surface. It can clearly be seen what happens when the surface's diffuse reflection is also included in the process in Figure 1.2(b).

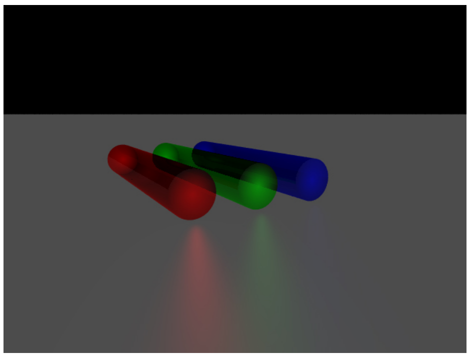
Ray casting is a special case of ray tracing. The only difference is that ray casting takes into consideration only the primary rays. Ray casting is generally used as a hidden surface removal method or a volume rendering method. In this work, we focus on performing ray casting on a CPU-GPU cluster.

1.2.1 Acceleration Structures

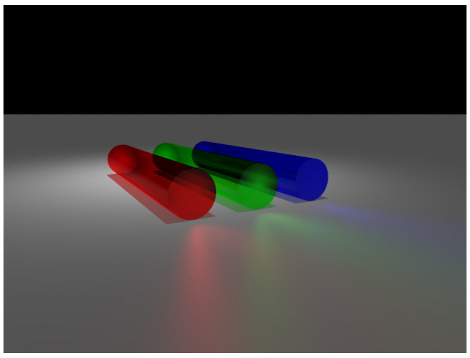
The original ray tracing algorithm suggests that a ray must be tested against each object in the scene for an intersection. To avoid this high computation cost, there are several

Algorithm 1 Whitted-Style Ray Tracing

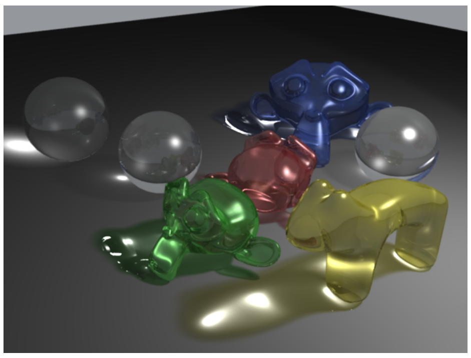
```
function render()
    for each pixel in the viewing place do
        spawn a ray from the eye point through the current pixel
        current pixel's color ← trace(ray)
    end for


function trace(ray)
    find nearest intersection
    compute intersection position and normal
    color ← shade(intersection position, normal)
    return color


function shade(position, normal)
    color ← 0
    for each light source in the scene do
        trace spawned shadow to light source
        if shadow ray intersects light source then
            color ← color + direct illumination
        end if
    end for
    if surface is reflective then
        spawn a new reflection ray
        color ← color + trace(new ray)
    end if
    if surface is transparent then
        spawn a new refraction ray
        color ← color + trace(new ray)
    end if
    return color
```



(a) Rendering Without Diffuse Reflection



(b) Rendering With Diffuse Reflection



(c) A Caustics Example

Figure 1.2: Deficiencies of Ray Tracing

acceleration structures proposed. Their ultimate goal is to significantly reduce the number of intersection tests.

Uniform grid is one of the first acceleration structures to be proposed. In this acceleration structure, the bounding box of the scene is divided into equally sized volumes called voxels. Each voxel is assigned to a list of triangles. Once the construction of the grid is finished, this grid structure is traversed. There are various approaches on how the grid is traversed. The most popular ones are 3D - Digital Differential Analyzer [1], Proximity Clouds [6] and Anisotropic Chessboard Distance [57].

In *Bounding Volume Hierarchy* (BVH), objects in the scene are partitioned instead of partitioning the space. BVH starts by construction of the BVH tree. The root of the BVH tree is the bounding volume of the whole scene, the intermediate nodes are bounding volumes of the assigned triangles and leaves are the triangles themselves. There are two known methods used for construction of the tree. One is a top-down approach proposed by Kay and Kajiya [26]. The other one is a bottom-up approach introduced by Goldsmith and Salmon [14]. Once the construction of the tree is finished, the traversal of the tree is straightforward. If a ray doesn't intersect the bounding box of a node, it is guaranteed that the ray cannot hit the children. So the traversal continues from the next node, which depends on the traversal method used. The traversal method chosen is usually depth-first.

KD-tree is another spatial subdivision technique. The construction of the KD-tree requires a top-down approach. Given a bounding box and a list of triangles, a splitting plane perpendicular to the one of the axes which splits the box into two pieces. The splitting plane is chosen based on a cost function. MacDonald and Booth [31] have proposed a cost function which assumes that a ray is more likely to hit the child with the bigger surface area. Haines [17] and Pharr [45] has proposed derivations to this cost function also. Once the splitting plane is set, each primitive contained by the parents are assigned to children according to the splitting plane. This process is recursively repeated on the children until desired depth is reached or the triangles assigned to a node drops below a specified threshold. During traversal, both children's bounding box are tested against the ray. The children that are first hit is traversed recursively. If an intersection is not found than the traversal continues from the other child.

Havran [18] has made a comparison of the acceleration structures we have discussed in order to discover the fastest acceleration structure. In his work, Havran concludes that KD-tree is the one showing best performance and BVH is the one with the worst performance.

1.3 Graphics Processing Unit

The idea of using GPUs for general purpose computation is not new. There have been computers like Ikonas [9], The Pixel Machine [47] and Pixel Planes-5 [53], which have been used in GPGPU researches. With the recent advancements in GPU technology, the graphics card used by normal users became much more sophisticated. Consequently, the research done on GPGPU area has become a very hot research area in the last few years. The researches performed are not limited to computer graphics discipline. Lengyel [30] using GPU for robot motion planning, Hoff [27] taking advantage of z-buffer techniques for the computation of Voronoi diagrams and Bohn [3] using it in the computation of artificial neural networks are just a few examples to such researches.

While the early generations of GPUs were only limited to process simple primitives, now they are much more flexible and faster. Current GPUs can be thought as highly parallel processors. They are especially good at arithmetically expensive operations. While Moore's law states that the speed of CPUs are doubled every eighteen months, the speed of GPUs are doubled every six to twelve months. Unfortunately, there are some restrictions imposed by the architecture of GPU. In [7], the restrictions of GPGPU are summarized as follows:

- Limited input/output registers.
- Inefficient random memory access.
- Inability to write dynamic memory locations.
- Lack of a sophisticated branch prediction units.
- Limited recursion capability.
- High communication cost between CPU and GPU.

All these restrictions must be thoroughly considered in GPGPU to avoid under utilization of GPU which can severely affect the overall performance of the implementation.

1.3.1 The Rendering Pipeline

The virtual environment in a computer consists of objects, camera information and light information. The method chosen to describe objects is usually triangle meshes. Vertex data and connectivity information are included in mesh data. Besides the position information, each vertex usually have additional information like color and normal. What the graphics pipeline does is simply to take this virtual environment information as an input and after a chain of operations, produce the two dimensional final image. The traditional pipeline mechanism that usually takes place every time we use our computers can be seen in Figure 1.3.

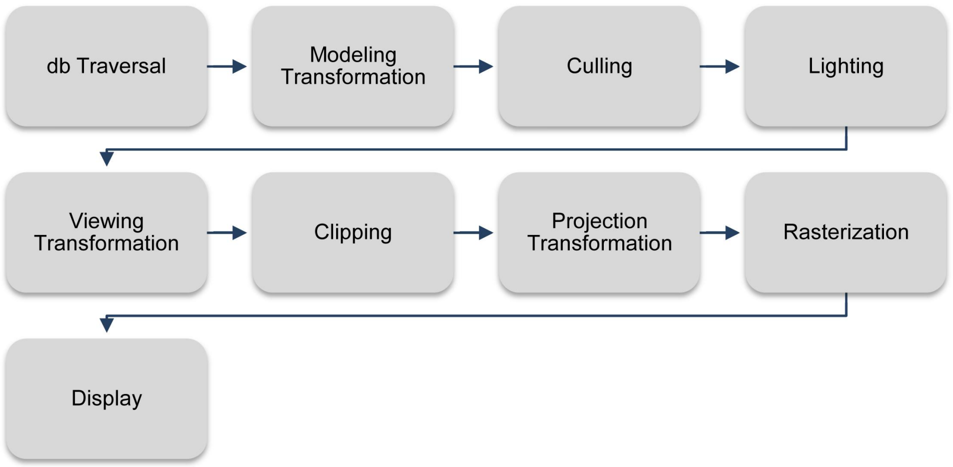


Figure 1.3: Traditional graphics pipeline (Courtesy of Alphan Es [7])

Figure 1.3 shows how a fixed function pipeline works. Once the input is given the fixed set of functions are called one after another. The only thing that can be done to change the course of pipeline execution is to set some states before feeding the input. For example lighting can be enabled or disabled by changing the corresponding states. But it is not possible to change the shading function from Gouraud shading to another shading method.

In order to be able to change the fixed functions in a graphics pipeline, the idea of using programmable processors is introduced. Today, this idea is the basis for modern GPU arhitecture. Figure 1.4 depicts the new GPU pipeline . With this new architecture, the user has the freedom to program everything listed in a green box in the figure to behave different. For example, user is no longer limited to use the shading technique that the pipeline forces to.

The programmable processors of a modern GPU are vertex processors, geometry processors and fragment processors. Vertex processors are responsible for processing vertex information and forwarding the generated output. Geometry processors are almost like the vertex processors except the fact that they can create or discard geometry primitives. Finally the fragment processors are responsible for modifying each fragment (pixel candidate)

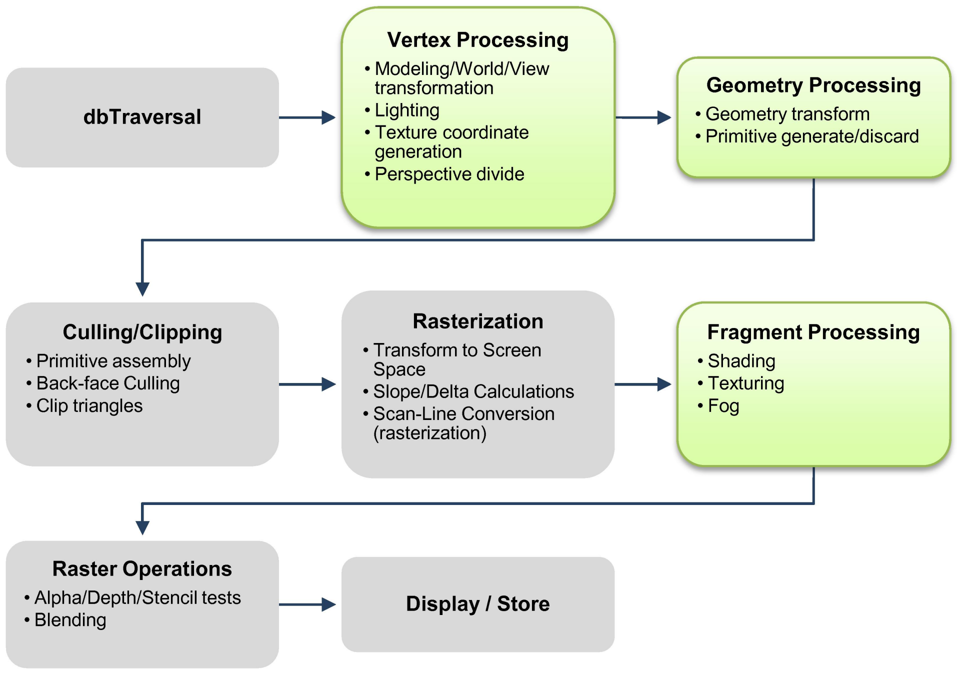


Figure 1.4: GPU graphics pipeline (Courtesy of Alphan Es [7]

depending on the program loaded on them.

1.3.2 Parallel Stream Processors and GPU

Before starting to program a GPU, one must completely understand the parallel stream processing paradigm. Traditional CPUs are based on the Single Instruction Single Data (SISD) architecture. This means that a CPU handles one operation at a time. Unlike the CPU, a parallel stream processor works on multiple data at the same time. It takes a set of data, which is called a stream, as input. Each element in the stream is processed by a function which is called a *kernel*. Kernels can accept a number of streams and they can produce a number of streams in return. The characteristics of an application which runs effectively on a stream processor are as follows [43]

- *Compute Intensity*: The number of arithmetic operations performed per I/O is very high.

- *Data Parallelism*: Each element in the stream can be processed while the other elements in the stream are being processed by another kernel. Also a kernel doesn't have to wait for the result from a previous element to process the current element.

- *Data Locality*: Any intermediate stream is accessed a couple of times immediately after its production and never needed again.

Image, audio, video and digital signal processing applications exhibit these characteristics making them good candidates for stream processing.

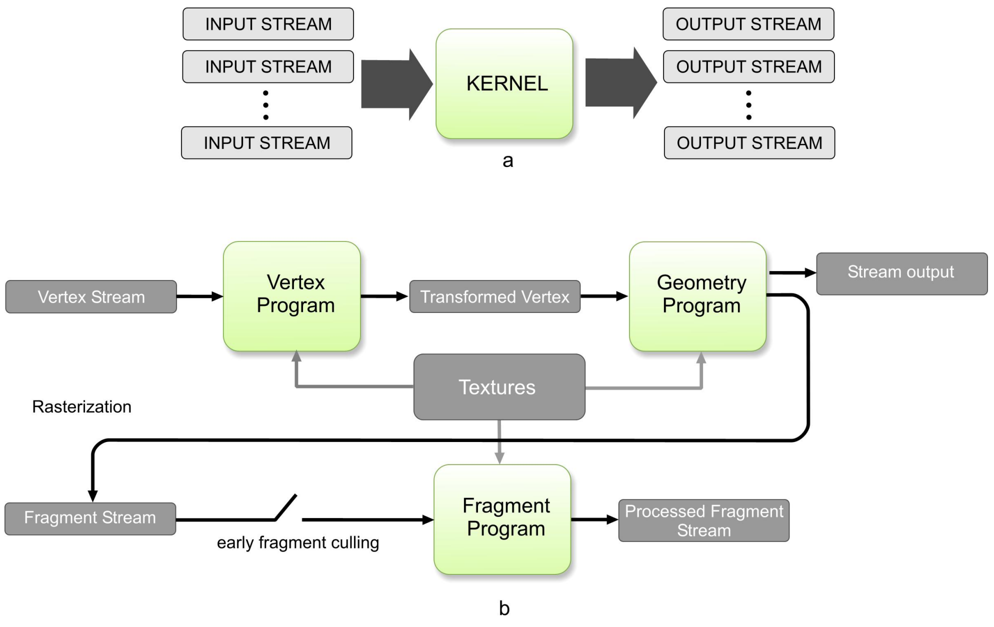


Figure 1.5: (a) Stream Processing (b) GPU as a stream processor (Courtesy of Alphan Es [7]

In Figure 1.5, parallel stream processing and GPU pipeline are visualized. The green boxes in Figure 1.5 represent the stream processors in a GPU. Vertex, geometry and fragment programs are actually what is being called as a kernel in stream processing. Each of these kernels can work on multiple input streams and output multiple streams. The biggest difference between a parallel stream processor and a GPU is the fact that if a number of different kernels are needed to be executed by a specific processor, the whole pipeline must be repeated after loading the next kernel on the corresponding processor. For example if we were to use two different kernels on fragment processors, we would have to complete the first pass with the first kernel loaded on the fragment processor, than feed the output acquired to the start of the pipeline and start the whole process again after loading the second kernel on the fragment processor. Another limitation comes from the fact that unlike the vertex processor, fragment processor can only output to a specific memory location (pixel) defined by the geometry. This makes it hard to implement scattering operation on a GPU.

One of the most important details in the GPU pipeline is *early fragment culling*. Not all of the fragments feeded to fragment processor need processing. To avoid processing the unnecessary fragments, early fragment culling must be used. This is done by utilizing depth or stencil buffers. The fragments that don't need processing any more are masked in the depth buffer. This way, they are going to be discarded before reaching the fragment processor.

1.3.3 GPU Programming Languages

The straightforward way to program GPUs is to use the graphics API which is usually either DirectX [33] or OpenGL [39]. When GPUs were first introduced the programmers were limited to use low level assembly like languages to program vertex and pixel shaders. Today both DirecX and OpenGL have their own high level shading languages High Level Shading Language (HLSL) [34] and OpenGL Shading Language (GLSL) [40]. The advantages of using a high level shading language are as follows:

- It is easier to understand, change and program.
- The code written is not platform dependent.
- The compiler takes care of low level optimizations automatically.

Cg (C for Graphics) is another high level programming language that was introduced by NVIDIA. We have preferred to use Cg in our implementation as it was very similar to the C programming language that we are already used to. Reader is strongly encouraged to read [12] and [44] as they were a great source for us in trying to learn Cg. Other GPU programming languages that are widely used are Brook for GPU [28] and Sh [22].

CUDA [37] by NVIDIA and Close To the Metal (CTM) [2] by AMD are both focused on trying to minimize the overhead introduced by the graphics API used while providing more direct access to the GPU. Unfortunately, they are both in early stages of their lifetime at the moment and they still need time for further development.

1.4 Goals

Our first goal is to implement ray casting on a single GPU. To achieve reasonable computation times we will make use of Extended Anisotropic Chessboard Distance [11].

When large scenes are to be ray casted, memory limitation becomes a serious issue. Our second goal is to overcome this problem and use ray casting on the GPU cluster to render large scenes in a data parallel manner.

When computation is taking place on the GPU, the CPU is almost idle waiting for GPU to finish its work. Considering this fact, as a last goal we are going to extend the work we have accomplished on the GPU cluster to the CPUs in the cluster. For accomplishing this goal we will make use of the CPUs to traverse the rays that have been identified to be outside of the scene assigned to the CPU-GPU couple. This way we will transform the GPU cluster into a CPU-GPU cluster which will make better use of the available hardware.

We believe that in the end, this work will further advance the previous ray tracing on GPU researches and set a good example of how it is possible to use a CPU-GPU cluster effectively for general purpose computing.

1.5 Outline

Chapter 2 is dedicated to background discussion of parallel ray tracing approaches and GPU based ray tracers.

Our implementation details are revealed in Chapter 3. We explain the data structures we have used in depth in this chapter. We also share the details of how ray tracing is mapped onto the CPU-GPU cluster in the third chapter.

In Chapter 4, we give the results we have achieved and discuss them thoroughly. The results achieved in single GPU and CPU-GPU cluster are compared with each other. We will also present the results we achieved with large scenes that is not possible to ray trace on a single GPU.

Finally we conclude our work and suggest future research areas in Chapter 5.

CHAPTER 2

RELATED WORK

In this section we will focus on giving background information about parallel ray tracing and GPU based ray tracers. Both of these topics are interconnected with each other and included in our work. Therefore a good understanding of these concepts is a must.

2.1 Parallel Ray Tracing

When ray tracing was first introduced, it was known for its high computation cost. Acceleration structures are developed to decrease this high cost. But they are not enough for a good speed up by themselves. Also there is the high space cost problem of ray tracing which makes it impossible to render large scenes that cannot fit in the memory of a single processor. To remedy these costs as much as possible, the idea of parallelizing the ray tracing is proposed. The key idea in parallel ray tracing is to render an image using multiple ray tracers at the same time. Each of these ray tracers are assigned a part of the image and each of them solves a part of the final image.

There are three different methods to create a parallel ray tracer. Each of these methods have their own advantages and disadvantages which we will explain in the next sections.

2.1.1 Demand Driven Ray Tracing

Since all the rays in a ray tracer are independent from each other, the obvious way to go is to replicate all the scene data on each processor and assign the rays to be traced respectively to all of the processors [42, 16, 48, 32, 51]. Once processors finish their tasks, the results are combined and the final image is displayed.

The tasks can be distributed before the computation begins. However, since the complexity of each part of the scene can differ significantly, load imbalances can easily occur. To overcome this problem, processors can be assigned new tasks as they become idle. This way it is ensured that the idle time for a processor is minimal.

The biggest advantage of task driven approaches is that the communication is minimal since all the processors work independent of each other. This also means that the control of parallel computation is really simple. It is all about sending the rays and receiving the results, nothing more. Another advantage is that there is no need for a modification to the original algorithm. The only thing that changes is the number of rays being processed. Last but not least, the speed up expected to be achieved by using this type of parallelization is linear.

The biggest disadvantage introduced by demand driven approaches is the fact that the whole scene data must be replicated on each processor. If the scene is too large to fit in the memory of a single processor, this approach cannot be used. This draw back makes the use of demand driven approaches limited.

2.1.2 Data Parallel Ray Tracing

Data parallel ray tracing makes it possible to use ray tracing on arbitrarily large scenes. In data parallel ray tracing, the scene itself is partitioned instead of partitioning the tasks. Each processor is assigned a volume in the scene and each processor trace the rays going through its volume assigned [8, 52, 5, 23, 51].

Once each processor receives the sub volume it needs to handle, the processor must first control if the ray it is currently processing is passing through the volume assigned to the process. If it is, than ray tracing process is applied to the ray. Otherwise, the ray is simply skipped.

The advantage of data parallel approach is that it is not limited by the size of the scene as long as there are enough processors. In other words, the disadvantage of demand driven approach is remedied by this approach. Unfortunately, load balancing becomes a severe problem in this method. The efficiency of the implementation is heavily dependent on the way the scene is partitioned. If a processor is assigned a volume which is not visible it means that this processor will be idle throughout the whole process.

In order to solve the load imbalance problem, Salmon and Goldsmith [54] proposes to subdivide the scene based on some cost criterion. These criteria are as follows:

- The data must be distributed such that each processor's load will roughly be the same.
- The memory consumed by each processor must be roughly the same

- The communication cost introduced by the fact that some rays will pass through multiple processors must be minimized

Unfortunately, there is still no algorithm to satisfy all of these requirements yet. Therefore, load imbalance remains as the major issue of data parallel ray tracing approaches.

2.1.3 Hybrid methods

As the name already suggests, hybrid methods are a mix of both data parallel and demand driven approaches. The goal for this kind of methods is to avoid disadvantages of the previous two approaches explained while trying to keep the advantages of them.

Reinhard and Jansen [52] first try to change the demand driven part so that it is possible to apply data parallel techniques. To achieve this goal, they group together coherent rays. This way the rays that are most likely to pass through the same objects are grouped which in return reduces the number of calculations required. In [52], rays are enclosed into a bounding cone and the objects intersecting the cone are found. The individual rays in the cone are ray traced using only these objects found. In the parallel case, the only thing to be done is to send the ray bundle and the objects intersected to a processor. After this point, rendering is done just as it was in a demand driven ray tracing technique.

2.2 Ray Tracing on GPUs

Carr et al. have tried to exploit the idea that CPU and GPU can work at the same time in their work [4]. They have performed experiments to reveal the advantages and disadvantages of these two different architectures. As a result, they found out that GPU is better at handling intersection tests compared to CPU. Using this fact, they have implemented a system which is called *The Ray Engine*. The pixel shaders are utilized for ray-triangle intersection tests. Vertex shaders are used to ignore triangles that are not related to the ray bucket sent. The CPU part uses octree as an acceleration structure and rays are bundled as batches for the GPU to perform intersection tests. Although they were able to reach a speedup of 33% compared to pure CPU implementation, the images produced had artifacts because of the 24-bit precision limit of the GPU they have used at that time. The reported speed is 120 million ray-triangle intersection tests per second on a Radeon 8500.

Purcell has implemented a complete ray tracer on the GPU [50]. Purcell's work is very similar to our single GPU implementation. The GPU is visualized as a stream processor and four kernels are implemented for ray tracing which are ray generator, traverser, intersector and shader kernels. Once the required data structures are transferred to the GPU memory, everything is handled by these kernels running on the GPU. The biggest difference with our and his implementation is the fact that while he has used geometric primitives to describe the scene, we are using triangle meshes which is harder to ray trace. For example, a ray-sphere intersection test is much faster than a ray-triangle intersection test. Although Purcell has not reported any artifacts on the final image, he was using a GPU with 24-bit precision just like Carr et al. [4].

While Purcell's work was making use of uniform grid acceleration structur, Ernst et al. [10] implemented a GPU ray tracer based on kD-acceleration structure which is claimed to be the best performer on a CPU among the other acceleration structures [18]. Unfortunately, their kD-Tree traversal implementation on the GPU have a high space cost proportional to the maximum depth of the stack which is a requirement enforced by the original kD-Tree implementation. It also requires multiple passes to be performed for pushing into the stack.

Another work following Purcell's organization is Foley et al.'s work [13] which is also based on kD-Tree. To alleviate the disadvantages of using a stack that Ernst et al. have experienced, Foley et al. have proposed two different traversal kernels. Both of these kernels have the same goal, eliminate the requirement for a stack. The first kernel is called *kd-restart*. When no intersection is found in a leaf node, the search is simply restarted from the root of the tree in this kernel. The other traverser kernel is called *kd-backtrack*. A modified kd-tree is constructed for this kernel to be utilized. Each node in the tree stores both the bounding box information and a pointer to the parent. This way, when a hit is not found in a node, the search is restarted from the parent, instead of restarting from the root. Naturally, kd-backtrack has a bigger space requirement for the tree. After the experiments, they found out that for the scenes that have objects uniformly distributed uniform grid performed better while their implementation performed better on scenes with non-uniform distribution. Also they have observed that the performance of the traverser kernels proposed were similar which was due to the fact that in the average case both kernels terminates after visiting only a few nodes.

Thrane and Simonsen [58] were the first ones to introduce the BVH acceleration structure on a GPU ray tracer. They have eliminated the requirement for a stack in their implementation and compared their implementation's performance to [13] and a uniform grid based GPU ray tracer that they have implemented again. After the experiments, they concluded that BVH is superior to other acceleration structures when it is implemented on a GPU which is a bit surprising as BVH is the worst performer on a CPU according to [18].

Es [7] has also implemented a GPU ray tracer based on the the uniform grid acceleration structure. In his work, four different traverser kernels are implemented which are 3D - Digital Differential Analyzer [1], Proximity Clouds [6], Anisotropic Chessboard Distance [57] and Extended Anisotropic Chessboard Distance which is introduced by Es himself again [11]. In his work, Es concludes that EACD traverser is superior to other traver kernels especially when a smaller voxel size is chosen. He also compares the performance observed to the performance seen in other researches indirectly by looking at the specs of the computers used and shows that his implementation is a better performer for even at scenes with large empty spaces and moderately even triangle density in non-empty voxels.

Unlu has implemented a demand driven ray tracer on a GPU cluster recently [59]. For the single GPU implementation, Unlu used the BVH based GPU ray tracer proposed by Thrane and Simonsen [58]. As expected, he observed that when the implementation is run on simple scenes, the total frame time is dominated by API and network costs causing non-linear speedups. The implementation performs best when it is used on complex scenes and number of tiles is chosen carefully.

CHAPTER 3

RAY CASTING ON A CPU-GPU CLUSTER

In this chapter, we will first explain how we implemented our single GPU Ray tracer. In next part, we will reveal the details of our CPU-Cluster algorithm and explain how we make use of CPU and GPU couple.

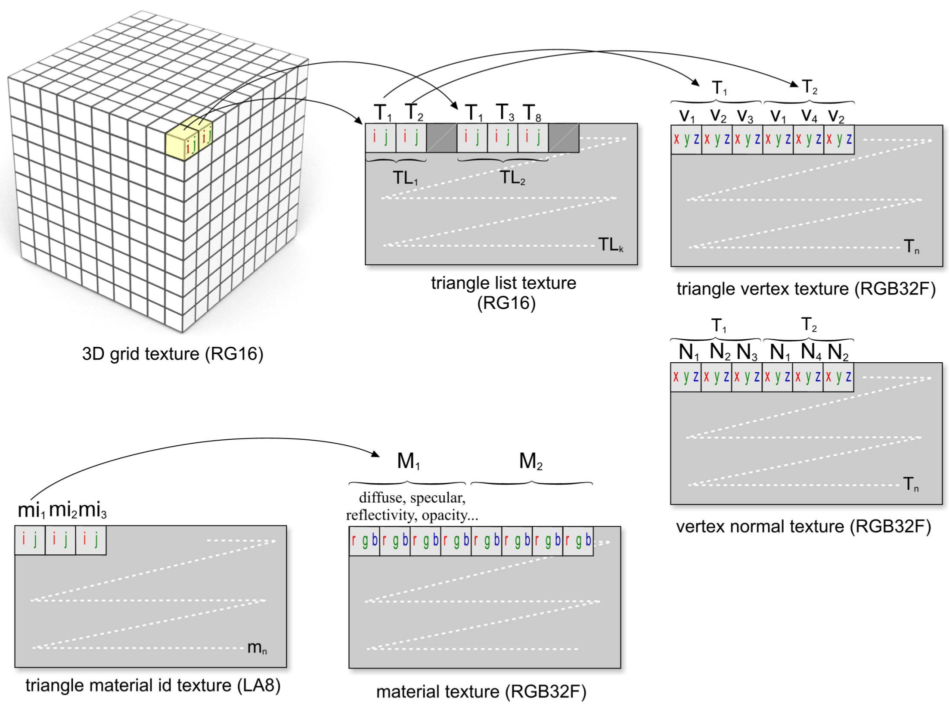
3.1 Single GPU Implementation

For our GPU ray tracer implementation, we have used the ray tracer proposed by Alphan Es [7]. We have used the GPU programming code written in Cg [38] shader language without making any significant changes. The CPU interface is developed using OpenGL graphics library.

One of the most popular methods for modeling virtual environments is using triangle meshes. We have used the same method for description of the scene in our GPU ray tracer implementation. The file format we have chosen for representing triangle meshes is Polygon File Format (PLY) which is also known as Stanford File Format. Each PLY file contains information about vertices, normals and connectivity. All of the PLY files we have used are in ASCII file format.

The scene database and the acceleration structure are stored as 1D, 2D and 3D textures in our GPU ray tracer. The organization we have used for representing the scene is illustrated in Figure 3.1. The 3D grid texture represents our uniform grid acceleration structure and it is created by the CPU in the preprocessing phase. Each element in this texture is an index to the start of a triangle list which is located in the triangle list texture. If the voxel is empty, the index is simply (-1,-1). Since there are two values required for indexing the 2D triangle list texture, the color format chosen is Red-Green with 16-bit integer components (RG16).

Each element in the triangle list texture contains the index required to find the vertex and normal values for a triangle. For the triangle list texture again RG16 format is chosen which enables us to index 2^{2x16} vertices. The vertex and normal textures are in Red-Green-Blue format with single precision floating point components. For each triangle there is also the material information. This information is represented by the material index and material textures.



3D grid texture (RG16)
T1 T2 T1 T3 T8
TL1 TL2 TLk
triangle list texture (RG16)
T1 T2
V1 V2 V3 V1 V4 V2
triangle vertex texture (RGB32F)
N1 N2 N3 N1 N4 N2
vertex normal texture (RGB32F)
M1 M2
diffuse, specular, reflectivity, opacity...
material texture (RGB32F)
mi1 mi2 mi3
mn
triangle material id texture (LA8)

Figure 3.1: Organization of the scene database in the GPU memory (Courtesy of Alphan Es [7])

A ray is defined by its origin and direction. Therefore, we created one texture for the origins of rays and another for the directions of rays in our GPU implementation. Figure 3.2 illustrates the fact that in a ray tracer, each screen pixel corresponds to a ray passing through that pixel. Therefore, for creation and processing of rays in the GPU, a screen sized quad is drawn on the screen. This way once a fragment program is loaded to the GPU and draw operation is started, one fragment is generated for each pixel. Ultimately, the fragment program loaded is executed for each of these fragments generated.

Our GPU implementation is the same as Es's implementation except the fact that he

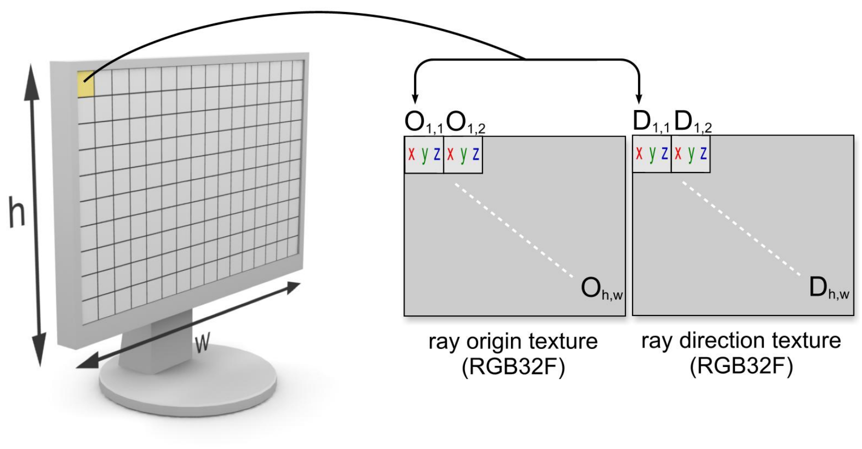


Figure 3.2: Rays are represented by two textures (Courtesy of Alphan Es [7])

uses Pbuffers [41] which are now obsolete. Instead, we are using state of the art Frame Buffer Objects (FBO) [55] for kernel I/O. The main advantage of FBO over pbuffer is that it does not require a context switch. Therefore, switching between different objects is faster. A comparison of both architectures is made by Simon Green from NVIDIA [56].

In our implementation, z-culling with the help of depth bounds test is used to avoid execution of fragment programs on unnecessary fragments which increases the overall performance in the end. During the process, each ray is given a state which is specified by the corresponding depth buffer value. The possible states for a ray includes creating, traversing, intersecting, intersected, shaded and out. The original GPU ray tracer proposed by Es utilizes five main kernels in the course of ray tracing. These kernels and ray state transition are depicted in Figure 3.3.

The first kernel we use is named *ray generator*. This kernel generates the ray origins and directions using the camera position and the position that the camera is looking at. The ray origins are all clipped against the bounding box of the scene. In this kernel, the depth buffer is modified so that the rays that are not hitting bounding box of the scene are set to state *out*. This is achieved by setting the corresponding depth buffer value to 0. All the other rays are considered as in *traversing* state and therefore, the corresponding depth buffer value is set to 1.

Traverser kernel is the kernel activated after ray generator. When the traverser kernel is first invoked, it checks if the ray is in an empty voxel. If it is in such a voxel, it loops until it finds a non-empty voxel. The output generated by this kernel is either used by the intersector kernel or resent to the traverser kernel for further traversal. Once the traverser

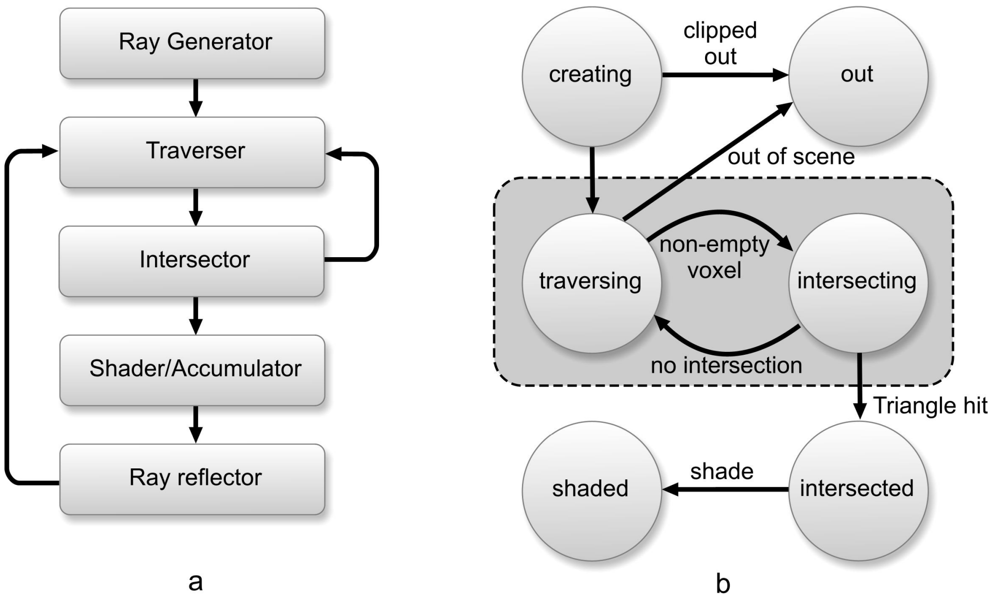


Figure 3.3: Ray tracing kernels and ray state transition (Courtesy of Alphan Es [7])

kernel finishes its execution, it sets the state of the ray to either out or intersecting. Es has proposed GPU implementation of four different traverser kernels which include 3D - Digital Differential Analyzer [1], Proximity Clouds [6], Anisotropic Chessboard Distance [57] and Extended Anisotropic Chessboard Distance [11]. We have included all of these traverser kernels in our implementation. The reader is strongly encouraged to read the GPU implementation details of these accelerated traversal kernels from Es's work [7] as they are implemented slightly different from their CPU versions.

The rays in the intersecting state are processed by the *intersector* kernel. If this kernel finds an intersection for the ray, the ray's state is set to *intersected* by modifying the depth buffer value to 0.2. Otherwise the state value is set to traversing again so that the ray can be traversed by the traverser kernel again. When a hit is found, the barycentric coordinates of the hit are written to the output texture.

After the execution of the *intersector* kernel, the pixels in *traversing* state are counted. If there is none, it means that all the rays are either in *intersected* state or out state. At this point, the intersection position and normals are calculated and forwarded to the *shader* kernel as input. The final color value for the ray is computed by using the Phong shading model [46]. The execution flow for a single ray is summarized in Algorithm 2.

The fifth kernel is the *reflector* kernel. Although we didn't include this kernel in our implementation, it is straightforward to add this kernel. Since GPUs do not support recur-

Algorithm 2 A single ray in GPU ray tracer

```
function GPU Ray Tracing(ray)
    Find origin and direction of ray
    Clip origin of ray against the bounding box of the scene
    if ray origin is out of the bounding box then
        color $\leftarrow (0, 0, 0)$
        return color
    end if
    while ray is in traversing state do
        while ray is in an empty voxel and ray is not in out state do
            traverse(ray)
        end while
        if ray state is out then
            color $\leftarrow (0, 0, 0)$
            return color
        end if
        Call intersector kernel
        if an intersection is found then
            Set ray state to intersected
        else
            Set ray state to traversing
        end if
    end while
    return color = shade(ray)
```

sion, Es proposes to use buffer stacks. The intermediate results are pushed to the stacks before secondary rays are fired and popped from stacks when primary rays are terminated and shaded.

3.2 CPU-GPU Cluster Implementation

The next step after completing the single GPU implementation was to parallelize it in a data parallel manner so that we can use it on a CPU-GPU cluster. To achieve this goal we preferred to use the so called *master-slave* paradigm with a slight modification which we will explain in the latter part. In this paradigm the *master* generates sub problems to be solved by the *slaves*. After sending the subproblem the master starts to wait for results and upon receiving the results it creates the final result. The organization for the CPU-GPU cluster is depicted in Figure 3.4

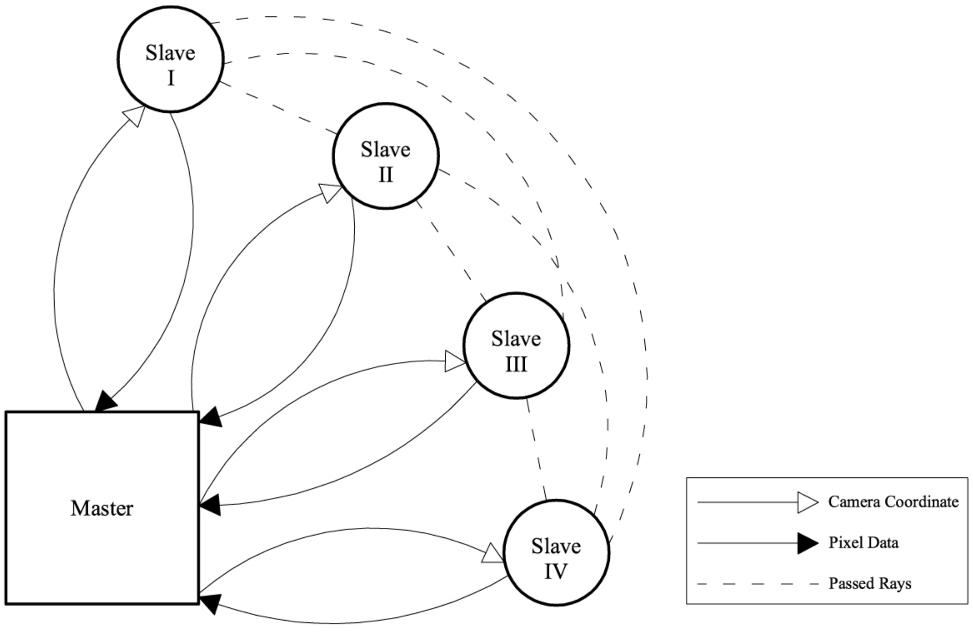
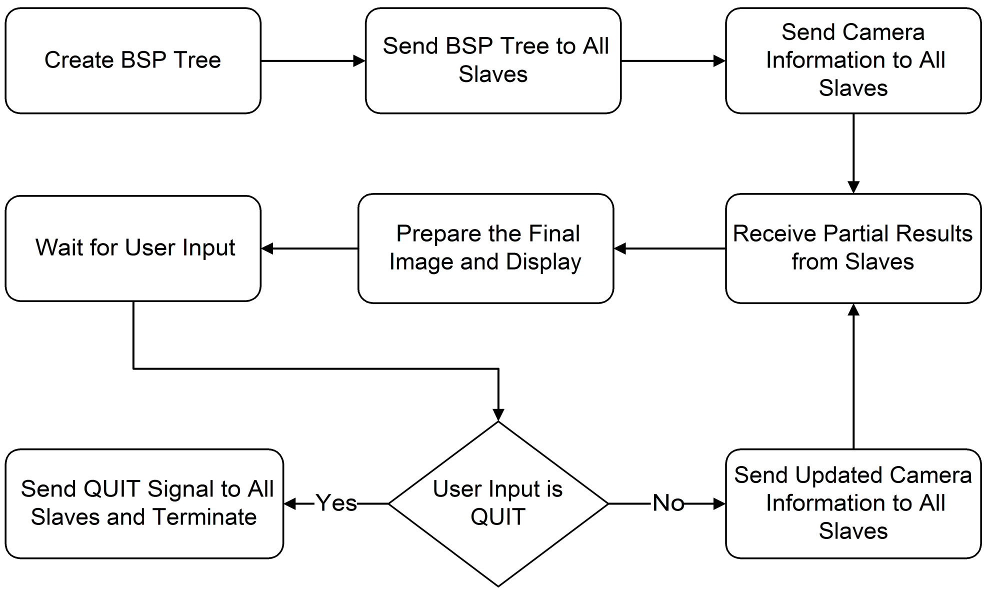


Figure 3.4: CPU-GPU Cluster implementation

In our work, the master process first builds a Binary Space Partitioning (BSP) tree. The construction of the BSP starts with choosing the splitting axis. In our case, we always choose the longest axis. The next thing we do is to sort the triangles in the given list along the splitting axis based on only one vertex. The sorting algorithm we have chosen is Combsort11 which is derived from Combsort Algorithm found by Stephen Lacey and Richard Box [29]. This sorting algorithm's average complexity is $O(nlogn)$ and we mainly choose this one over the others because of its performance and the fact that it doesn't require a stack. Once the sorting is finished, a splitting plane must be decided. If n is a positive integer and the number of leaves needed for the current node being processed is $2n$ then a plane which splits the triangles equally is chosen. If the number of leaves needed for the current node is $2n + 1$ then a plane which splits the triangles proportionally to values $n + 1$ and n is chosen. Once the splitting plane is decided, the triangle list of the children are created. Remember that the triangle list of the node was sorted based on only one vertex. Therefore, we need to make sure that we have added all the triangles whose two other vertices might fall inside the boundary of the children to the children's triangle list. This algorithm is recursively called on the children until the desired number of leaves reaches one. The creation of BSP tree is summarized in Algorithm 3.



Create BSP Tree
Send BSP Tree to All Slaves
Send Camera Information to All Slaves
Receive Partial Results from Slaves
Prepare the Final Image and Display
Wait for User Input
User Input is QUIT
Yes
Send QUIT Signal to All Slaves and Terminate
No
Send Updated Camera Information to All Slaves

Figure 3.5: Master Process

Once the BSP tree is successfully created, we would have requested number of leaves which have almost the same amount of triangles in their triangle list. Each of these leaves are assigned to slaves simply by sending the bounding box information of the leaf. After sending this information to slaves, the master sends the initial camera position to each slave and then starts to wait for the partial results. Once the master receives the partial results

Algorithm 3 BSP Tree Creation

```
function CreateTree(number of leaves requested)
    Create list of triangles
    Find bounding box for triangle list
    CreateNode(list of triangles, number of leaves requested, bounding box


function CreateNode(list of triangles, number of leaves requested, bounding box)
    n ← number of leaves requested
    if n is equal to 1 then
        return
    end if
    splittingAxis ← longest axis of the bounding box
    sort list of triangles based on one vertex
    if n is even then
        splitting plane ← the plane splitting sorted triangle list equally
        leftLeafCount ← $n/2$
        rightLeafCount ← $n/2$
    else
        splitting plane ← the plane splitting sorted triangle list proportional to $(n+1)/2$
        and $n/2$
        leftLeafCount ← $(n + 1)/2$
        rightLeafCount ← $n/2$
    end if
    boundingBoxLeft ← compute bounding box of left child
    boundingBoxRight → compute bounding box of right child
    listLeftChild ← triangles to the left of splitting plane
    listRightChild ← triangles to the right of splitting plane
    scan listRightChild and add the ones falling inside boundingBoxLeft to listLeftChild
    scan listLeftChild and add the ones falling inside boundingBoxRight to listRightChild
    CreateNode(listLeftChild, leftLeafCount, boundingBoxLeft)
    CreateNode(listRightChild, rightLeafCount, boundingBoxRight)
```

from all slaves, it combines these to create the final image. Finally, the image created is displayed on the screen. After this point the master is responsible for user interaction, sending the new camera information to slaves and receiving the results. The operations performed by the master process is summarized in Algorithm 4 and Figure 3.5.

Algorithm 4 Master Process

```
function Master Ray Trace(number of slaves)
    Create BSP Tree with number of slaves leaf count
    Send corresponding bounding box to each slave
    loop
        Send camera information to all slaves
        while There are slaves still busy with ray tracing do
            WAIT
        end while
        Receive partial result from each slave
        Display the final result
        Wait for user input
        if User input = QUIT then
            Send exit signal to all slaves
            Exit
        else
            Update camera information based on user input
        end if
    end loop
```

The slave process starts its execution by first receiving the bounding box that it is responsible for. Upon receiving its bounding box, the triangles and their information are read from the file. The next thing for the slave process is to create the necessary textures that we have discussed in the previous section. After these textures are ready, the slave starts to wait for the camera details. With the camera coordinates received, the slave starts the execution of ray tracing. Everything is the same as we have explained in section 3.1 except one part which is the creation of ray origins. As you know in the original implementation the ray origins are clipped against the bounding box of the scene. This time the rays are first clipped against the bounding box of the whole scene. The acquired clipping result is checked against the slave's bounding box. If the clipping result is inside slave's bounding box than this ray is processed in the first ray tracing pass. Otherwise, it means that the ray will be first processed by some other slave. In this case, the ray origin goes through another clipping process but this time it is against the slave's bounding box. The result is written in the origin texture but the state value for the ray is set *out*.

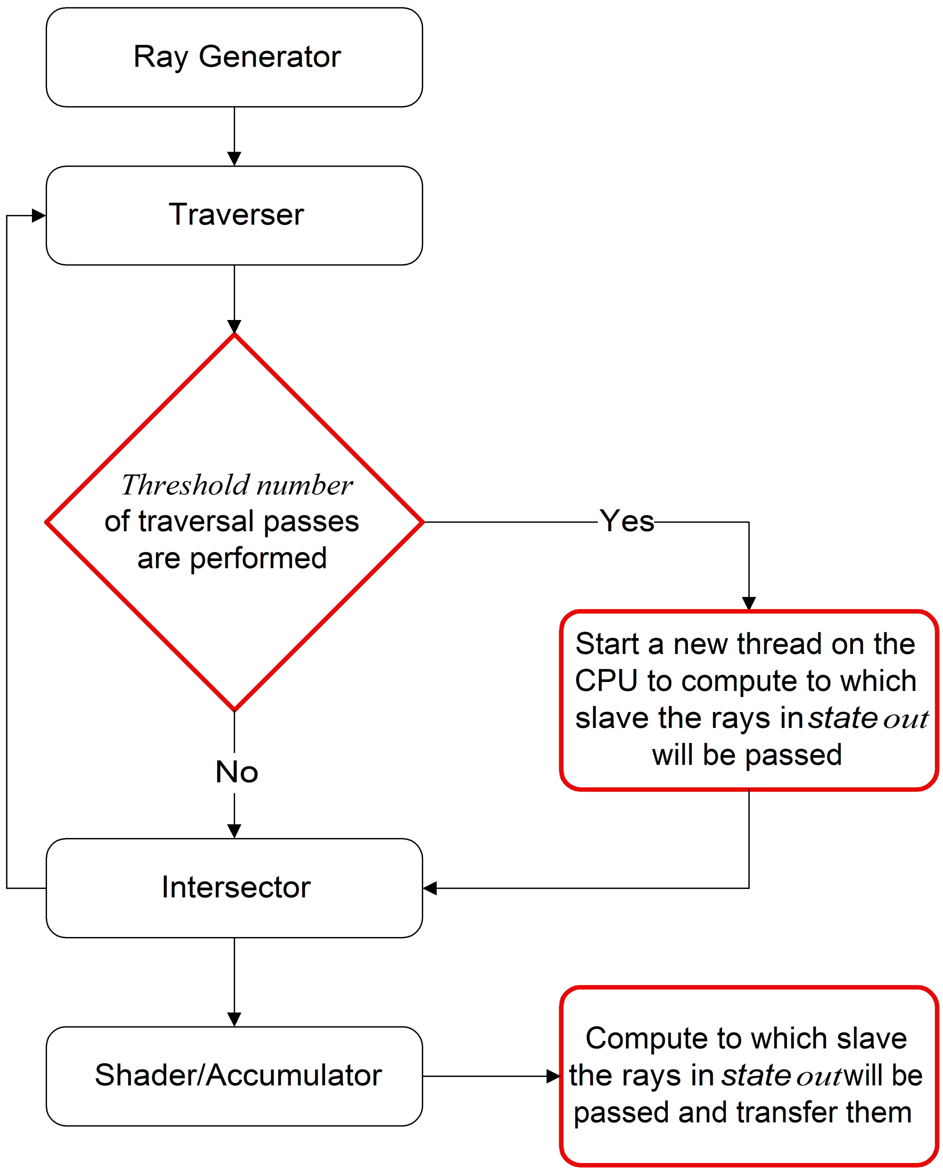

Ray Generator
Traverser
Threshold number of traversal passes are performed
Yes
No
Start a new thread on the CPU to compute to which slave the rays in state out will be passed
Intersector
Shader/Accumulator
Compute to which slave the rays in state out will be passed and transfer them

Figure 3.6: CPU-GPU Cluster Ray Caster

After the eye rays are created the GPU starts ray casting process. There are two points that must be considered at this phase. The first one is that as the number of traversal passes performed by the GPU increase, the number of rays that are in state *out* also increase. The second one is that GPU and CPU can do different things simultaneously. Based on these two facts, we start a new thread in the CPU after the GPU finishes a predetermined number of traversal passes which is specific to our CPU-GPU cluster implementation. This thread is responsible for finding to which slave the rays that are in state *out* will be passed. However, there can still be rays that are discovered to be in state *out* once GPU finishes ray casting. Therefore, the computation performed by the thread is repeated when ray casting is finished. Finally, the rays that still needs processing are sent to corresponding slaves and the shading result is left untouched. Figure 3.6 shows how the single GPU ray caster depicted in Figure 3.3 is modified to fit our needs. The red boxes denote our modifications to the original GPU ray caster.

After this initial ray tracing pass, the slave process starts to wait for further command from either the master process or another slave process. Communication between slave processes is our implementation's only difference from the classic master-slave paradigm. If a command from another slave process is received, it means that further ray tracing will be done and the steps above will be repeated for the rays that are received. However, if the slave receives a command from the master process at this point, it means that the master process is asking for the partial result to be sent. Once the transfer of the partial result is complete, the slave process starts to wait new commands from the master process which can either be new camera coordinates for a whole new ray tracing process or exit command. Algorithms 5, 6 and Figure 3.7 demonstrates the work done in the slave process.

The biggest difference between our CPU-GPU cluster implementation and a regular CPU implementation comes from the fact that in a CPU cluster implementation there is no overhead introduced by the rays that do not intersect any of the objects assigned to a slave. Since the rays are traversed on the CPU, ray tracing process and the process of determination of which rays are going to be transferred to other slaves can be done simultaneously. Also there is no CPU to GPU or GPU to CPU transfer overhead in a CPU only cluster implementation. This is due to the fact that everything is handled in the system's main memory.

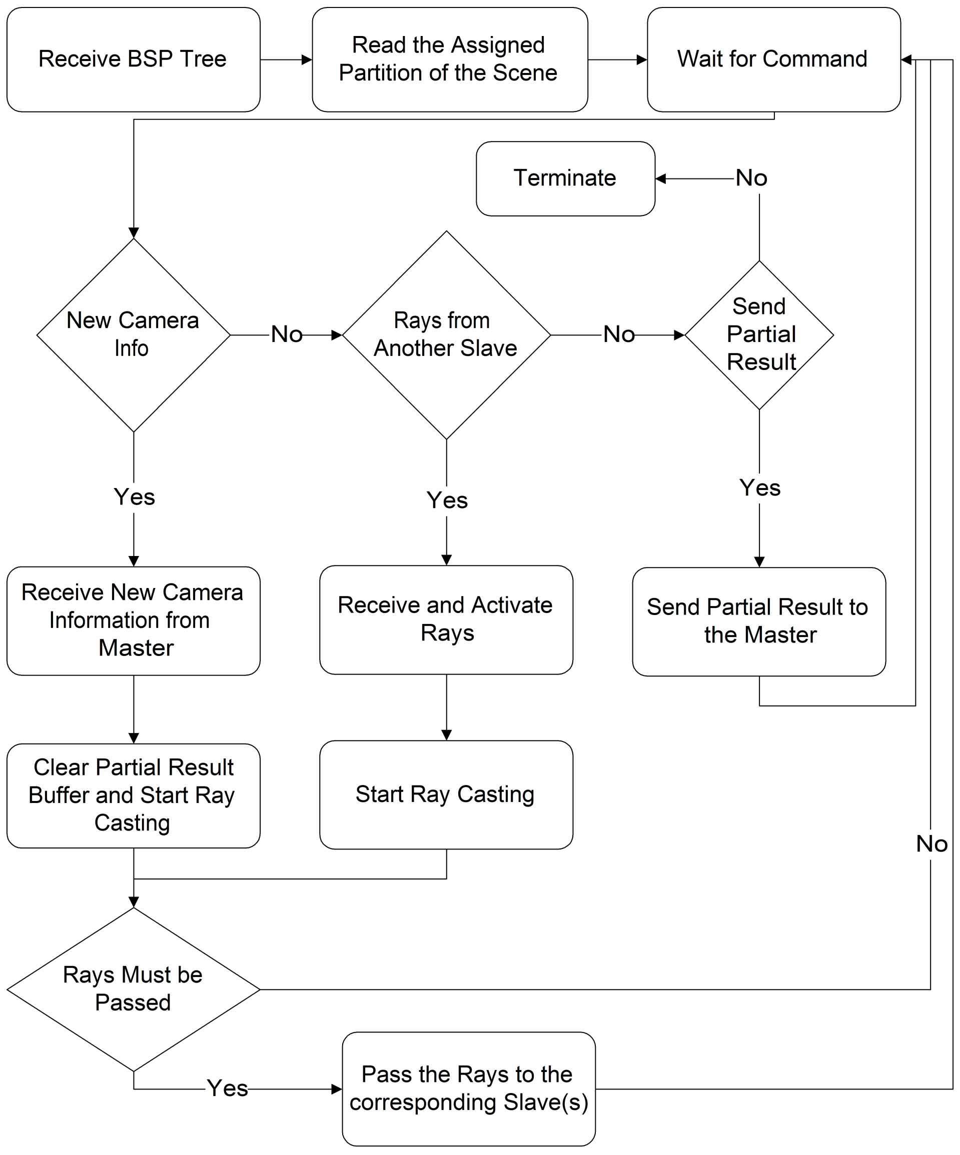

Receive BSP Tree
Read the Assigned Partition of the Scene
Wait for Command
Terminate
No
New Camera Info
No
Rays from Another Slave
No
Send Partial Result
Yes
Yes
Yes
Receive New Camera Information from Master
Receive and Activate Rays
Send Partial Result to the Master
Clear Partial Result Buffer and Start Ray Casting
Start Ray Casting
No
Rays Must be Passed
Yes
Pass the Rays to the corresponding Slave(s)

Figure 3.7: Slave Process

Algorithm 5 Slave Process

```
function Slave Ray Trace()
    Receive assigned bounding box from the master process
    Read the scene information based on the bounding box received
    Create and prepare the static textures needed by the GPU Ray tracer
    loop
        Wait for a command from any source
        if Command = QUIT then
            Exit
        else if Command = New camera information then
            Receive new camera information from the master
            Raytrace the rays that first hits the slave's bounding box
            if traversal step count is 4 then
                Read the depth buffer information from the GPU
                Create a thread and compute the next slave information for the missed
                rays specified in the depth buffer
            end if
            When intersection positions in barycentric coordinates is ready find out
            which rays still needs processing
            if There are rays still needing processing then
                for each ray that still needs processing do
                    if Next slave information is not computed by the CPU thread then
                        Compute the next slave information for the ray
                    end if
                end for
                Send the rays in need of more processing to the responsible slaves
            end if
            Finish ray tracing
            Keep the shading result
        else if Command = Process more rays then
            ProcessRaysSentFromASlave()
        else if Command = Send partial result then
            Send the partial result to the master process
        end if
    end loop
```

Algorithm 6 Process rays sent from a slave

```
function ProcessRaysSentFromASlave()
    Receive the rays from the sending slave process
    Set the received rays' state to *traversing*
    Start ray tracing
    if traversal step count is equal to *thresholdvalue* then
        Read the depth buffer information from the GPU
        Create a thread and compute the next slave information for the missed rays
        specified in the depth buffer
    end if
    When intersection positions in barycentric coordinates is ready find out which rays
    still needs processing
    if There are rays still needing processing then
        for each ray that still needs processing do
            if Next slave information is not computed by the CPU thread then
                Compute the next slave information for the ray
            end if
        end for
        Send the rays in need of more processing to the responsible slave
    end if
    Finish ray tracing
    Keep the shading result
```

CHAPTER 4

EXPERIMENTS

In this chapter, we will first explain our testing methodology. After that, we are going to share the performance results we have obtained from our CPU-GPU ray tracer explained in the previous chapter. We will also provide our interpretations of these results.

4.1 Experimental Setup

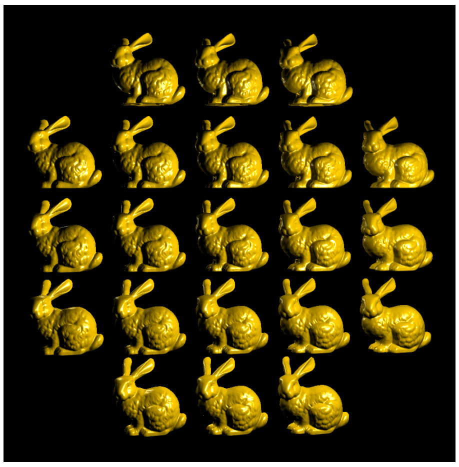
We have performed our experiments on a cluster including five computers. Four of these computers are identical and they are used by the slave processes. The specifications of these four computers are as follows:

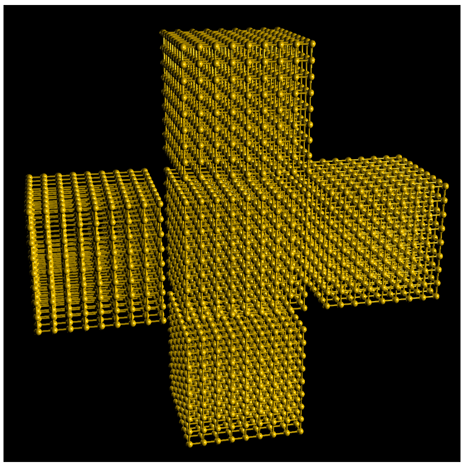
- Intel Pentium 4 3.20 GHz CPU
- 1GB DDR2 RAM
- NVIDIA GeForce 7800 GTX 256MB Graphics Card

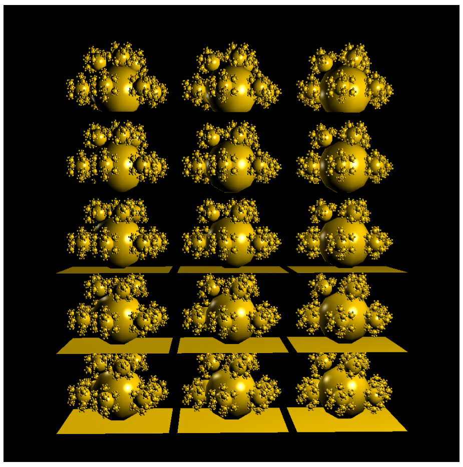
The fifth computer is used by the master process and its specifications are as follows:

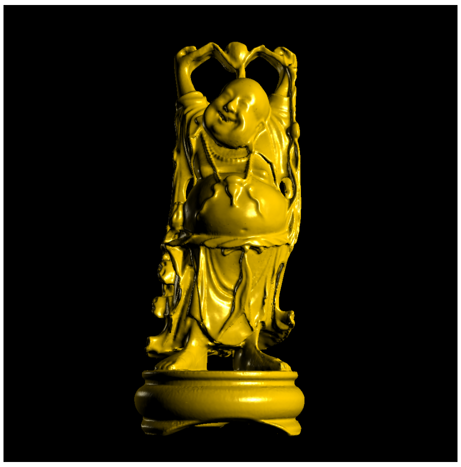
- Intel E4300 1.8Ghz CPU
- 2GB DDR2 RAM
- NVIDIA GeForce 7900 GS 256MB Graphics Card

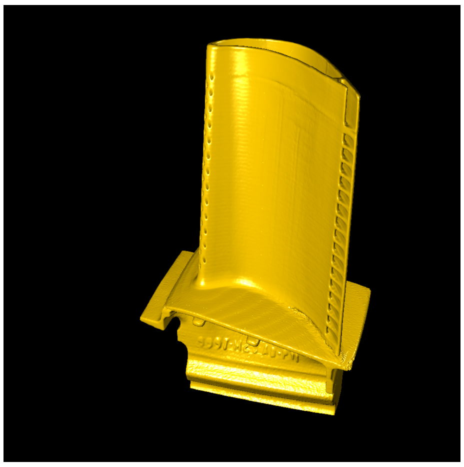
The computers are interconnected using a gigabit ethernet switch. Development is done using C++ and Cg 2.0. Intra-cluster communication library we have chosen is MPICH2 1.0.6 [35]. We have decided to use six different models which include the well known stanford bunny (1458471 triangles), happy buddha (1087716 triangles), lattice (626940 triangles), sphere (1328430 triangles), turbine blade (1765388 triangles) and manuscript (4305679 triangles). The final results achieved after ray casting these models can be seen in Figure 4.1.

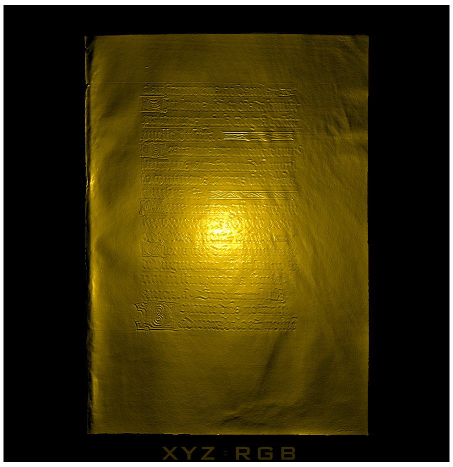

(a) Bunny


(b) Lattice


(c) Sphere


(d) Buddha


(e) Turbine Blade



(f) Manuscript

Figure 4.1: Ray Casted Models

In order to have a better understanding of the results, the camera is rotated around the model used with intervals of 10 degrees for a total of 36 different angles. This camera rotation process is repeated 10 times to make sure that the best result for each angle is stored.

4.2 Results

In our first group of tests we tried to determine the performance of our CPU-GPU cluster implementation using different camera angles and compared the results achieved with the different number of slaves. The only test scene that can be rendered using a single GPU is the Buddha scene Figure 4.1(d). Bunny Figure 4.1(a), lattice Figure 4.1(b) and sphere Figure 4.1(c) test scenes require at least two slaves while the turbine blade Figure 4.1(e) test scene requires at least three slaves. The manuscript scene Figure 4.1(f) is the biggest scene among the test scenes we have chosen and it requires four slaves to be used which is the maximum number of slaves in our test setup.

Figure 4.2 shows the angle vs time results achieved on the bunny scene. These results prove that our implementation's performance is highly dependent on the viewing angle. This is due to the fact that our implementation do not consider load balancing. Our implementation partitions the scene to balance the memory load on each slave. Still, there is only one case (Figure 4.2(c)) where there is a slowdown using a greater number of slaves.

In lattice Figure 4.3, sphere Figure 4.4 and turbine blade Figure 4.6 scenes the load imbalance problem becomes more apparent. While in the bunny scene there was only one case where there was a slowdown using a higher number of slaves, in these three test scenes there are multiple examples of this slowdown problem.

The buddha model is a highly detailed model which makes it a complex scene to ray cast in reasonable time periods. As it can also be seen in Figure 4.5, our CPU-GPU cluster implementation is able to improve the performance of the single GPU implementation significantly. However, the results are again dependent on the viewing angle. Another important point is that as the resolution used is increased the communication overhead gets more significant. As there is no communication overhead introduced when single GPU implementation is used, the performance gap is decreased significantly when the resolution is increased.

Figure 4.8, Figure 4.9, Figure 4.10 and Figure 4.11 shows the speedups achieved using bunny, lattice, sphere and buddha models based on the viewing angle. In these figures, the effect of using a higher resolution can be better seen. The resolution increase affects the communication overhead in a negative way especially when there is higher number of slaves. The speedup factors achieved using four slaves are decreased significantly when the highest resolution 1024x1024 is used. Another point is that in the bunny scene there seems to be a super linear speed-up. Unfortunately, this is again due to the poor load imbalance when two slaves are used in this scene.

Using the results we have achieved so far, we can say that it is not guaranteed that increasing the number of slaves will also increase the performance. The performance is dependent on the viewing angle unless there is some sort of load balancing mechanism added.

In the second group of tests, we tried to identify the major overheads introduced by our implementation. Figure 4.12 shows the overhead analysis results we have achieved using the bunny test scene. There are six major overheads. One of the major overheads comes from the nature of data parallel approach, load imbalance. The GPUs must be processing same number of rays at the same time in the ideal case. Unfortunately, with a data parallel approach this is very hard to achieve and depends on how you partition the scene. The idle times for the slaves in Figure 4.12 shows the load imbalance problem.

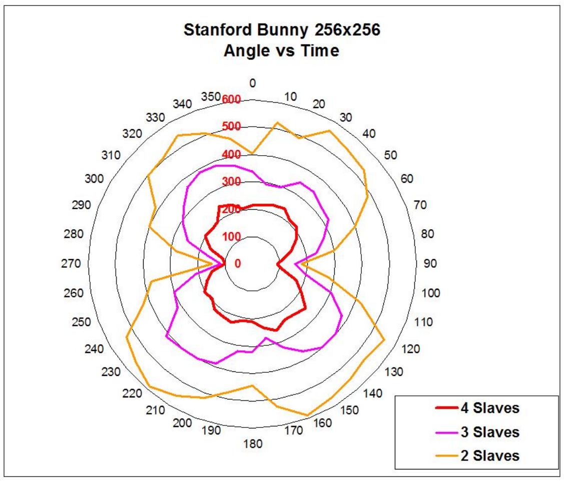
Another major overhead is that the whole depth buffer must be controlled once the ray traversal is finished in order to find out the rays that can still be processed by other slaves. In Figure 4.12 this is shown as the CPU traverse time overhead. The higher the resolution is, the higher the number of rays is. Therefore, CPU traverse time becomes more significant when the resolution is increased.

When a slave receives rays that are passed from another slave, it must transfer these rays to the GPU. As there is no way of transferring rays directly from the network to a slave's GPU memory, some time between CPU and GPU must be spent to transfer the passed rays. CPU to GPU transfer time in Figure 4.12 depicts this overhead.

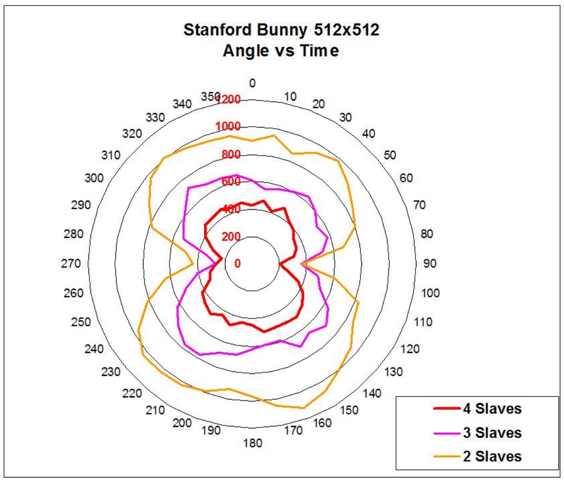
The time spent to transfer rays between slaves is insignificant compared to the overheads we have mentioned so far. This is due to the fact that we are not transferring the origins and directions of the rays. As all these information is already present on each slave from the beginning, our implementation transfers only the ID numbers of the rays which are integers.

The final overhead is the time spent on combining the partial results received from the slaves. This overhead is only linearly dependent on the resolution. Figure 4.12 shows that the time spent to combine results is 5.43 milliseconds when the resolution is 256x256. When the resolution is increased four times so that the resolution become 512x512, the time spent to combine the results is increased to 22.5 milliseconds which is almost four times of the previous timing.

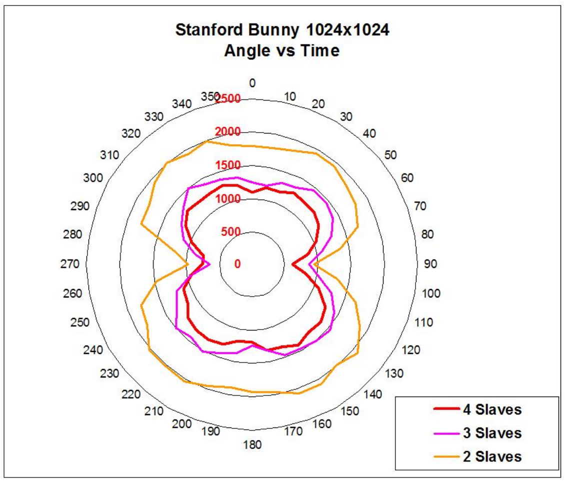
In the final group of tests, we focused on trying to find the performance the difference between our CPU-GPU and GPU cluster implementations. The results we achieved with this group of tests can be found in Figure 4.13 and Figure 4.14. It is clear that our thread modification to our GPU cluster implementation had a positive effect in all of our test scenes.



(a) 256x256 Resolution

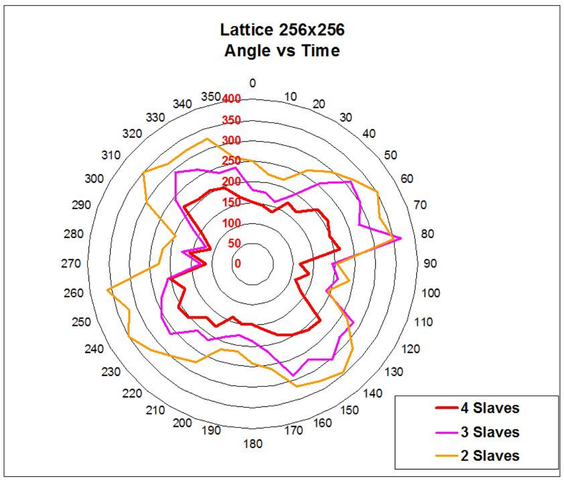


(b) 512x512 Resolution

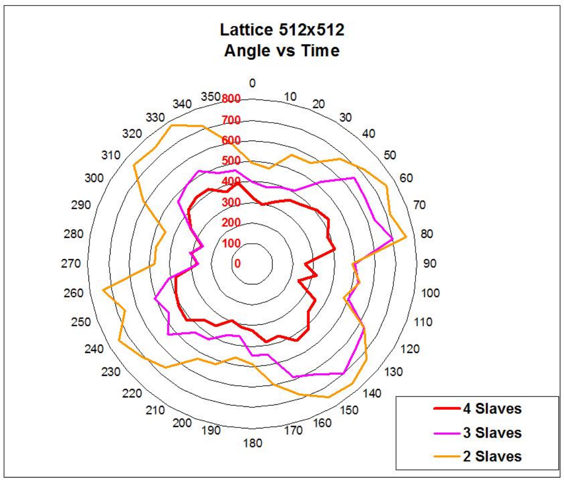


(c) 1024x1024 Resolution

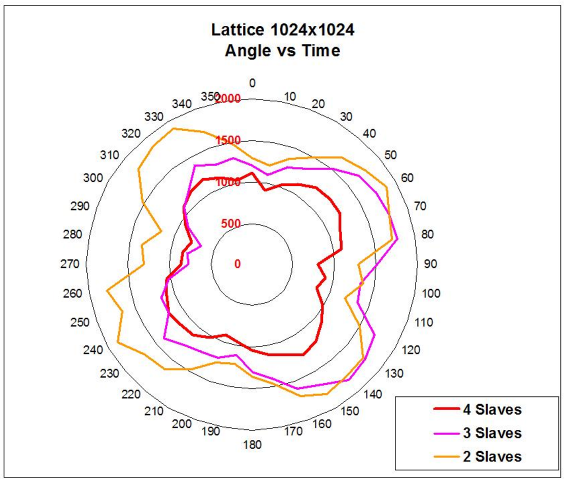
Figure 4.2: Bunny: Angle vs Time results



(a) 256x256 Resolution

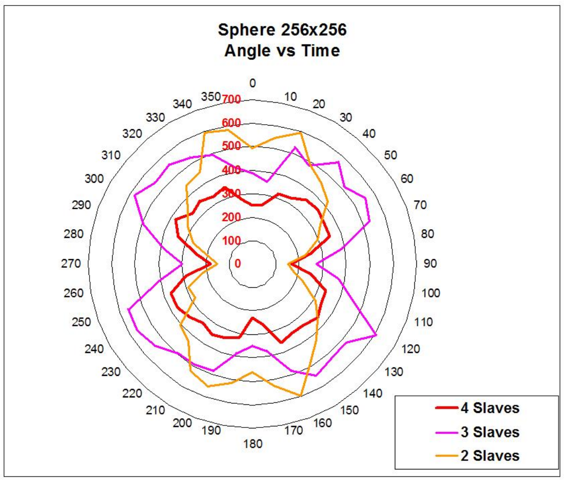


(b) 512x512 Resolution

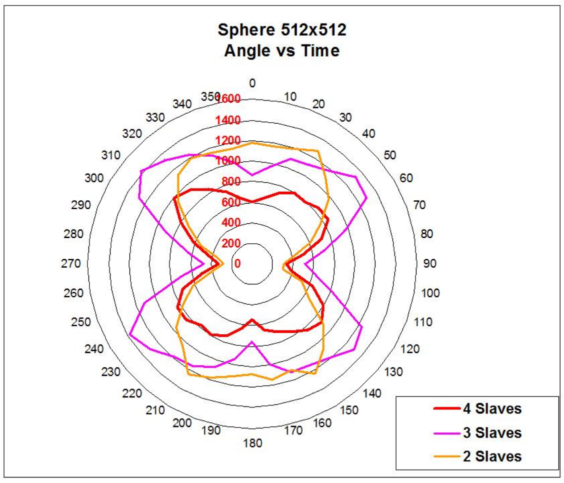


(c) 1024x1024 Resolution

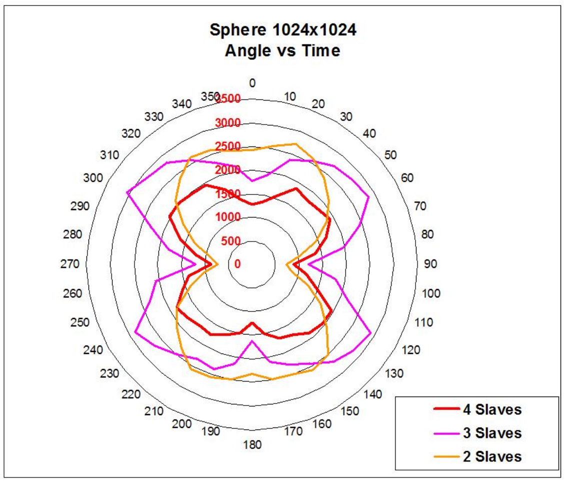
Figure 4.3: Lattice: Angle vs Time results



(a) 256x256 Resolution

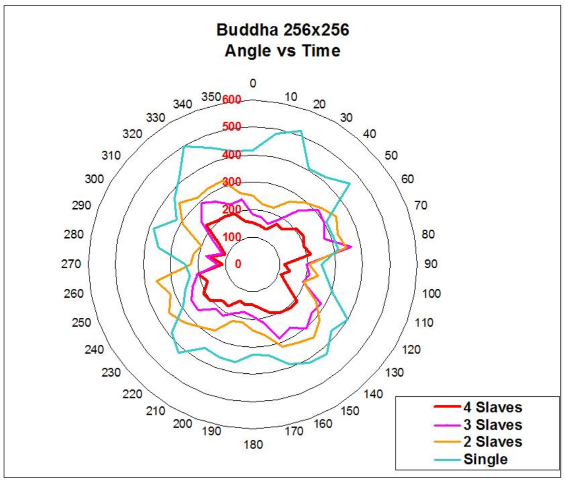


(b) 512x512 Resolution

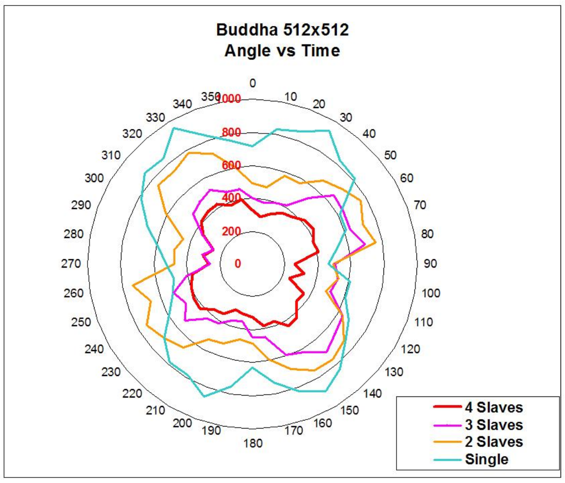


(c) 1024x1024 Resolution

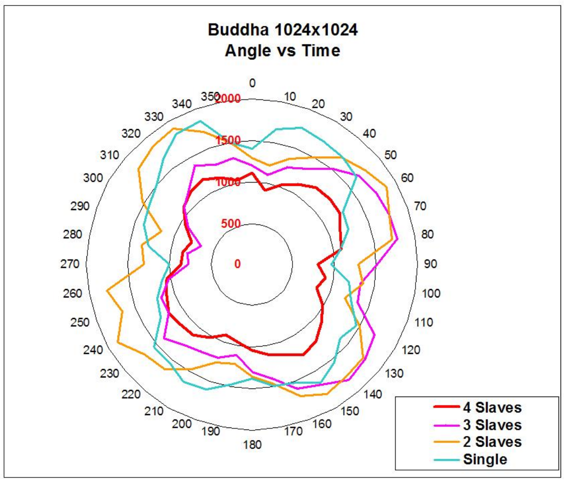
Figure 4.4: Sphere: Angle vs Time results



(a) 256x256 Resolution

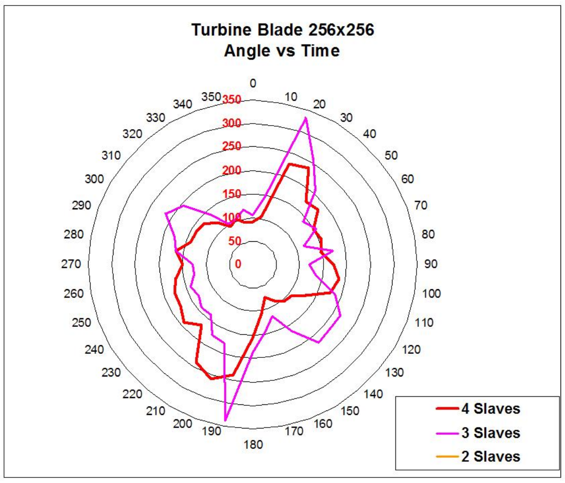


(b) 512x512 Resolution

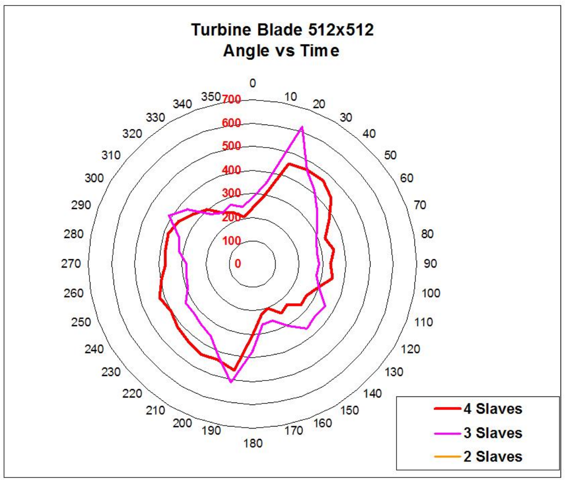


(c) 1024x1024 Resolution

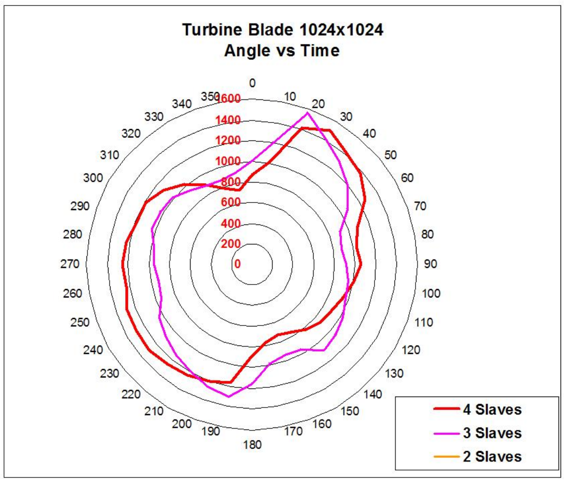
Figure 4.5: Buddha: Angle vs Time results



(a) 256x256 Resolution

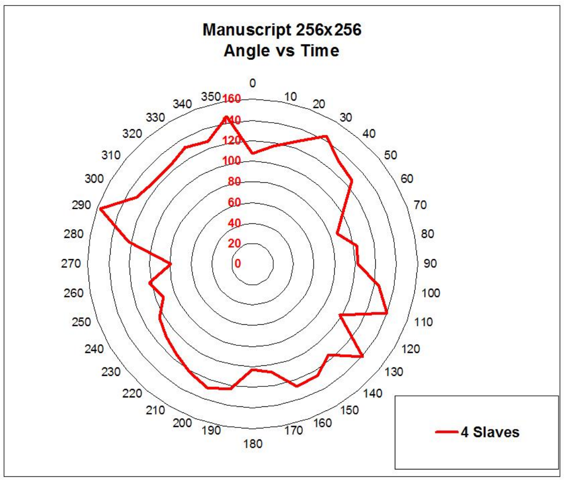


(b) 512x512 Resolution

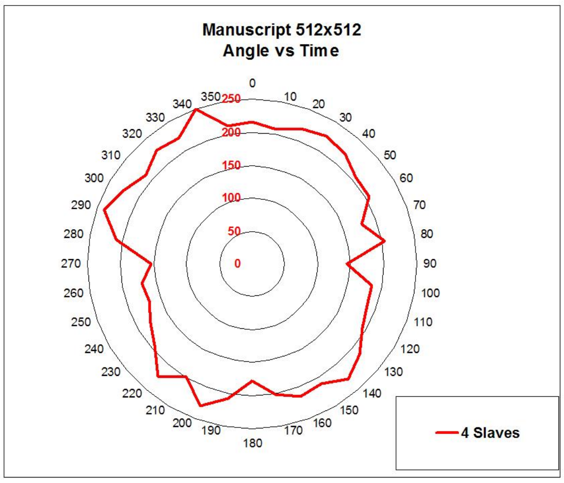


(c) 1024x1024 Resolution

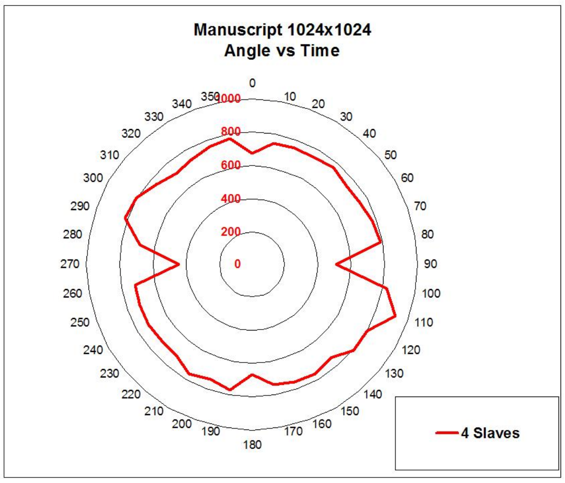
Figure 4.6: Turbine Blade: Angle vs Time results



(a) 256x256 Resolution

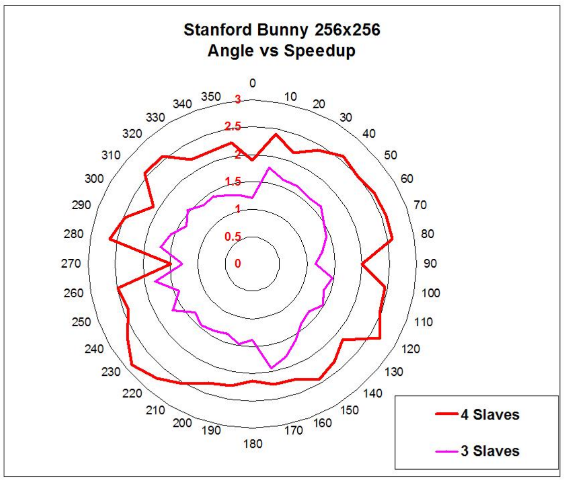


(b) 512x512 Resolution

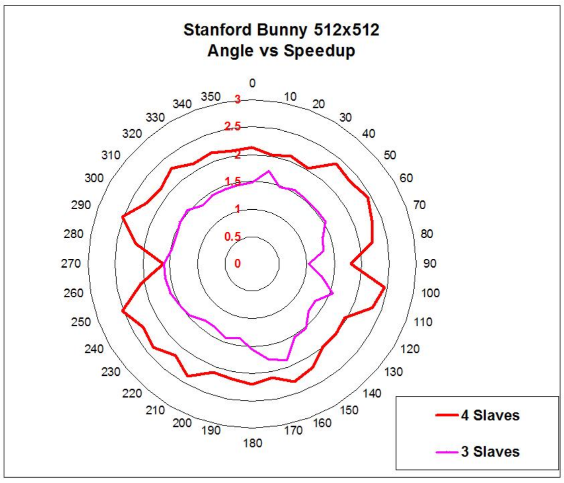


(c) 1024x1024 Resolution

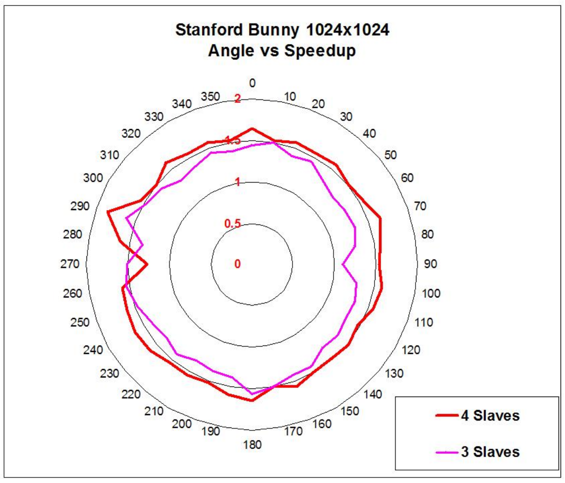
Figure 4.7: Manuscript: Angle vs Time results



(a) 256x256 Resolution

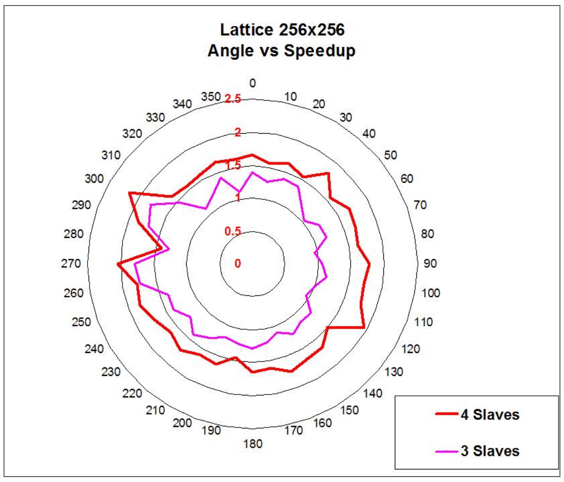


(b) 512x512 Resolution

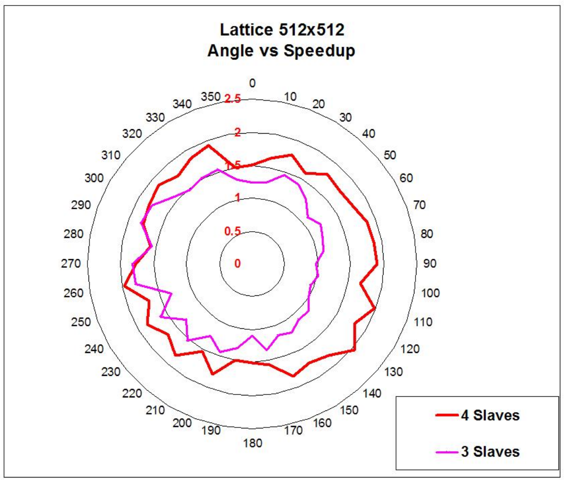


(c) 1024x1024 Resolution

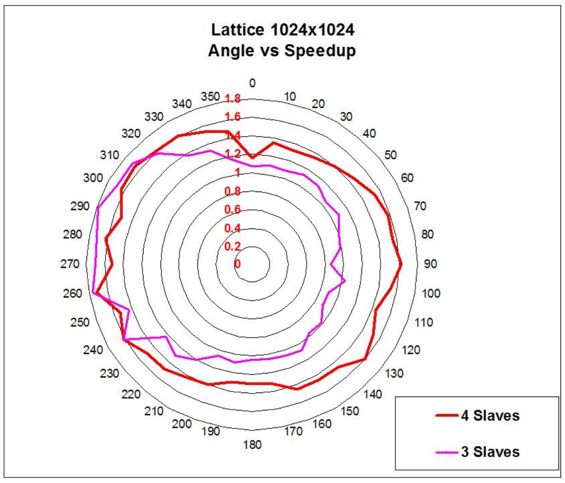
Figure 4.8: Bunny: Speedup against 2 Slaves



(a) 256x256 Resolution

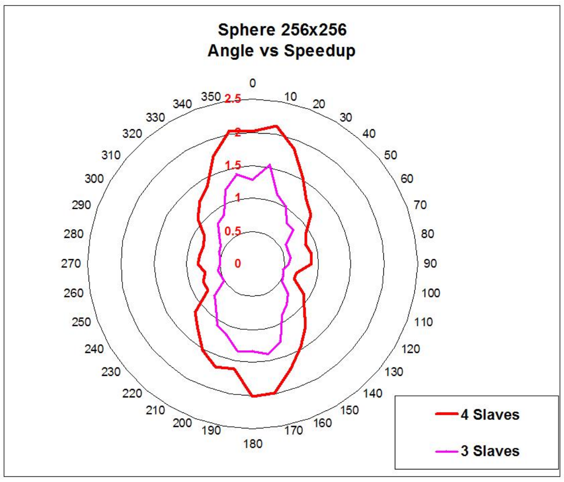


(b) 512x512 Resolution

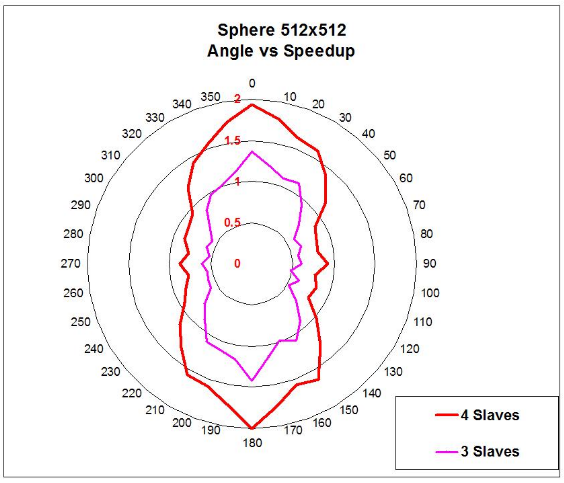


(c) 1024x1024 Resolution

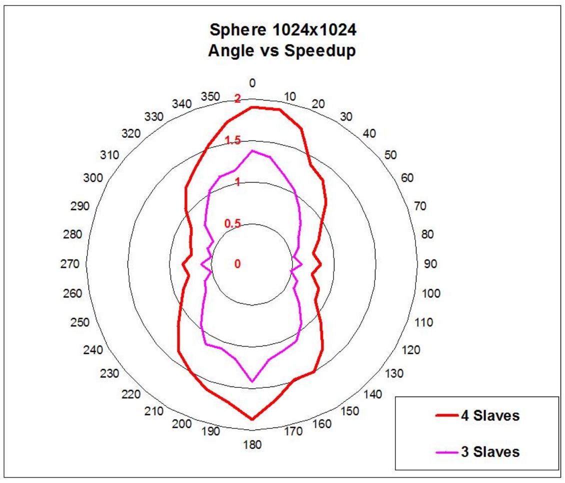
Figure 4.9: Lattice: Speedup against 2 Slaves



(a) 256x256 Resolution

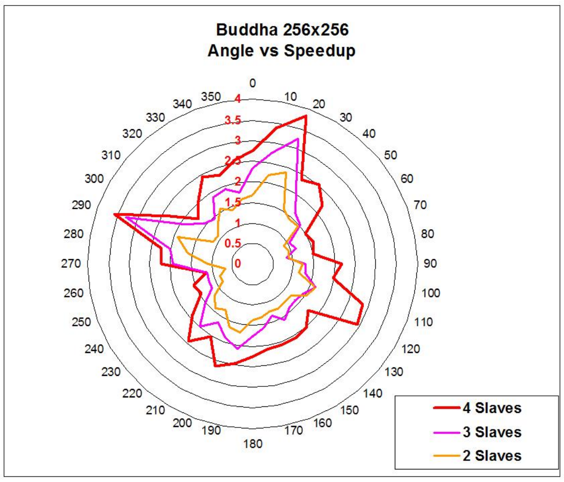


(b) 512x512 Resolution

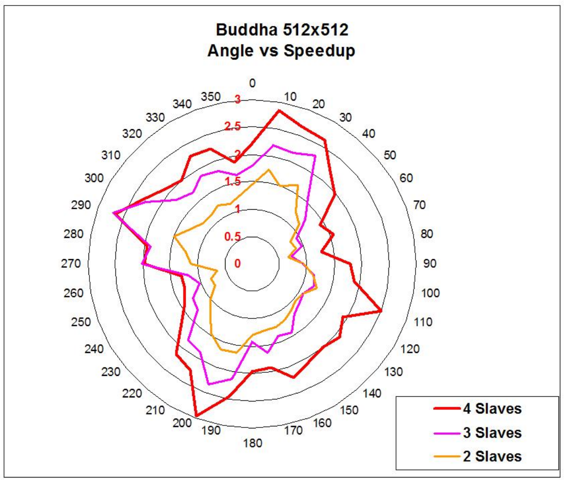


(c) 1024x1024 Resolution

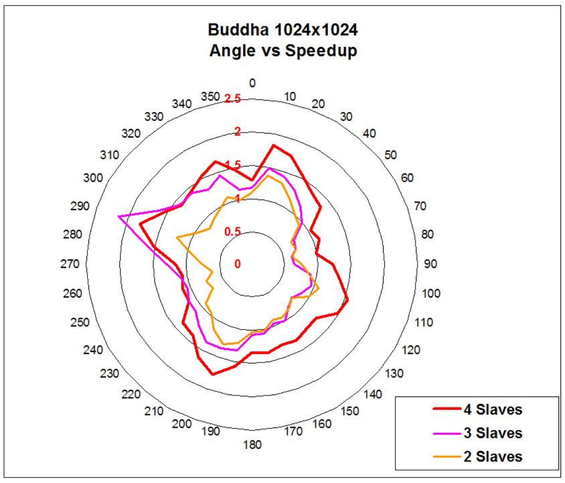
Figure 4.10: Sphere: Speedup against 2 Slaves



(a) 256x256 Resolution



(b) 512x512 Resolution



(c) 1024x1024 Resolution

Figure 4.11: Buddha: Speedup against single GPU implementation

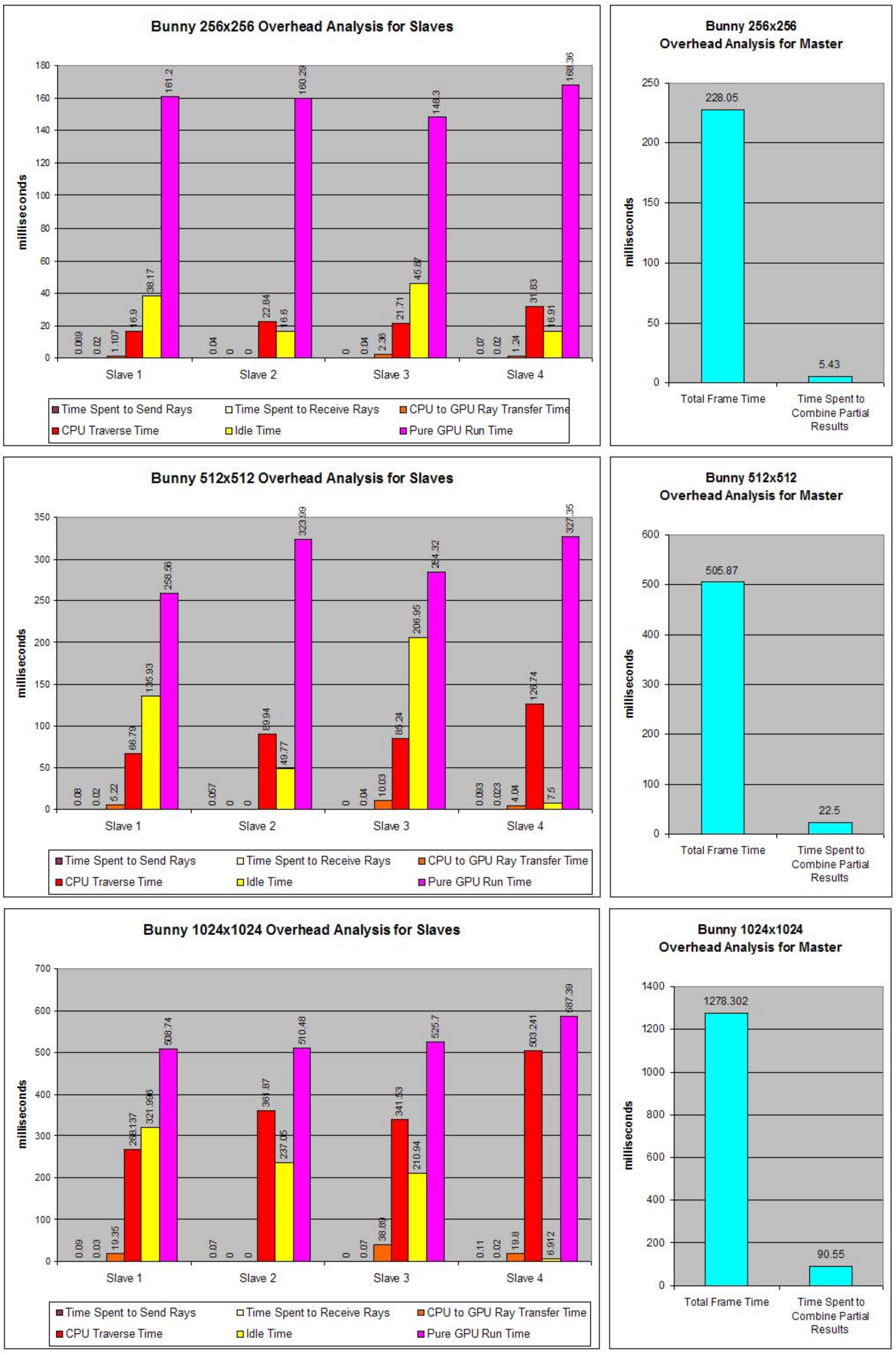
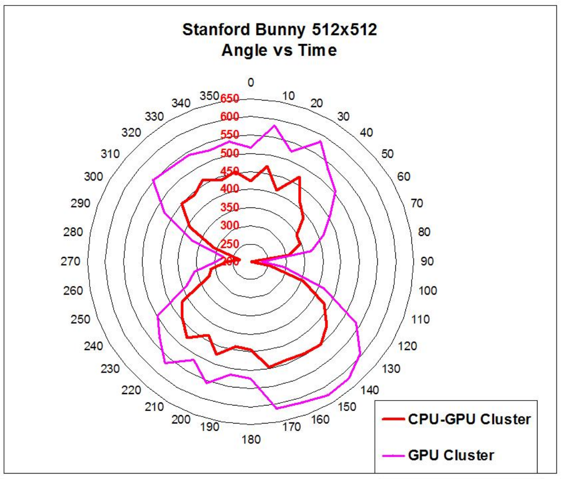
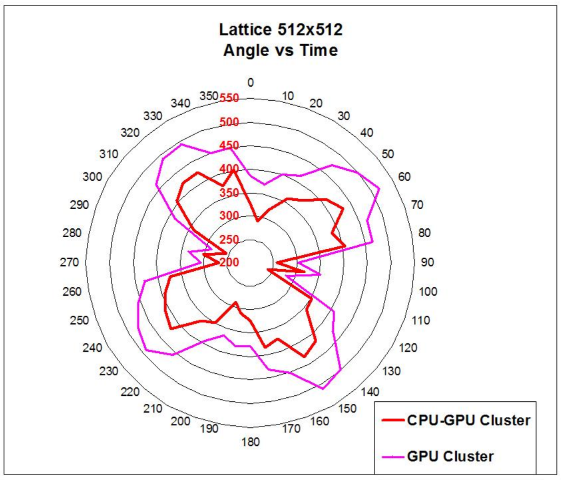


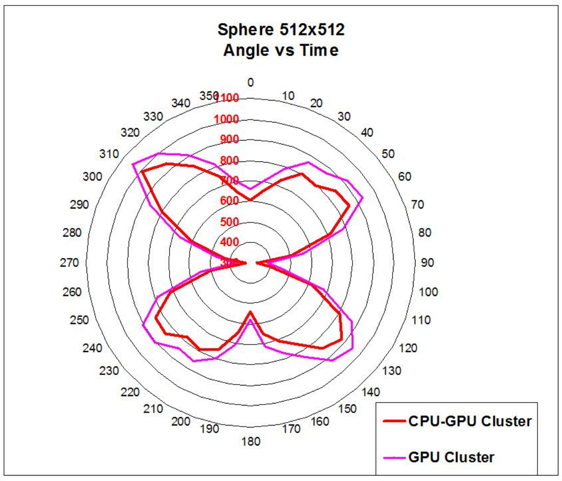
Figure 4.12: Overhead Analysis for Bunny Scene



(a) Bunny

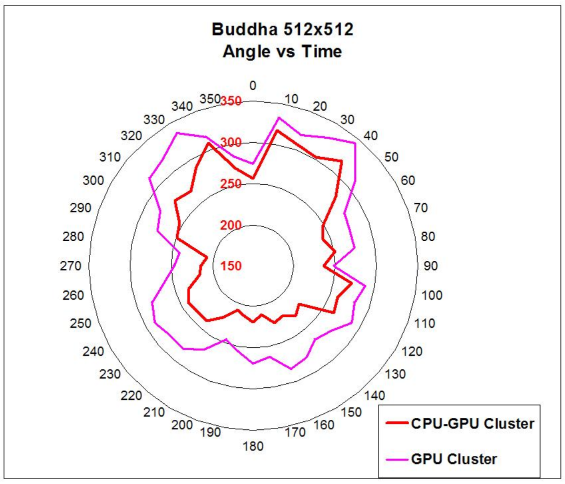


(b) Lattice

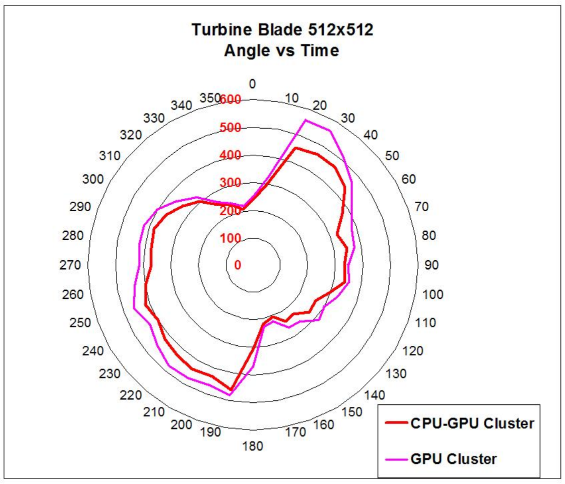


(c) Sphere

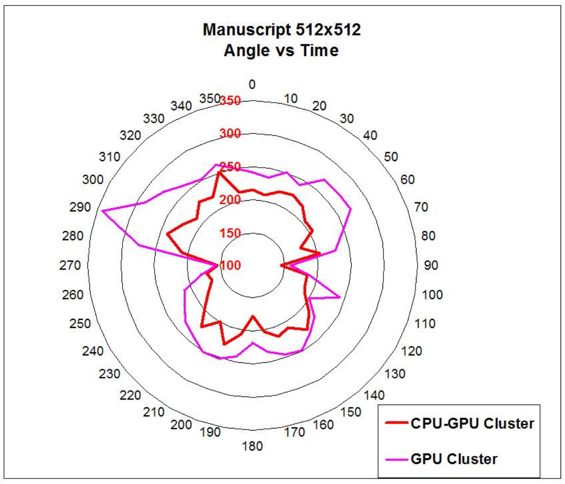
Figure 4.13: CPU-GPU Cluster vs GPU Cluster Part I



(a) Buddha



(b) Turbine Blade



(c) Manuscript

Figure 4.14: CPU-GPU Cluster vs GPU Cluster Part II

CHAPTER 5

CONCLUSION

Ray tracing on a GPU is a relatively new research area compared to ray tracing itself. Definitely, there is still room for more progress. In this thesis, we have shown that large scenes can be ray casted with the use of a CPU-GPU cluster. We have based our GPU implementation on Es's work [7]. We have utilized a data parallel approach to alleviate the memory constraint. We have also introduced the idea of using threads on CPU while GPU is working on ray traversal and intersection in order to improve the performance.

One of our goals at the beginning was to render scenes that are impossible to render on a single GPU. We have shown that we have achieved this goal by rendering stanford bunny scene with 1458471 triangles, lattice scene with 626940 triangles, sphere scene with 1328430 triangles, turbine blade scene with 1765388 triangles and manuscript scene with 4305679 triangles.

We have performed extensive experiments on our test scenes to compare the performance of our CPU-GPU implementation with using different number of slaves. Although we were able to balance the memory load on each of the computers by utilizing BSP tree partitioning of the scene, the performances achieved were dependent on the viewing angle. This was due to the load imbalance introduced by the nature of data parallel approach.

The biggest disadvantage introduced by using a CPU-GPU cluster is the fact that one can never be sure if a ray is going to cross the borders of the bounding box or not until the ray traversal is finished. To alleviate this disadvantage, we have made use of the fact that some of the rays that will be out of the bounding box is identified in the first few traversal passes. Based on this observation, we chose a threshold value for starting the status check of rays in a CPU thread. If a ray that is in state *out* is discovered, the slave that the ray will be passed is computed by this thread. Naturally, this threshold value is highly dependent on the scene being processed. With this approach, we were able to utilize both the CPU and GPU at the same time. Consequently, we observed a notable performance improvement compared to the non-threaded approach where CPU is waiting for the GPU to finish ray tracing process. Unfortunately, there can be rays in *out* state in the end of the GPU traversal which were still in *traversing* state when this thread was launched. The same process that took place in the thread must be repeated for these rays too. Therefore, there is also the overhead introduced by these rays.

Another problem we faced with using a GPU is the fact that the transfers between the GPU and main memory must be minimized due to high performance penalty of such transfers. Unfortunately, we had to utilize this type of communication for:

- Transferring rays to GPU memory that were sent by another slave.
- Checking rays' status by reading the depth buffer content.
- Reading the partial result from GPU memory.

The final and the most obvious overhead is the communication overhead. This overhead impacts the overall performance especially when the results from the slaves are gathered and a high resolution is used.

We believe that our implementation's performance can be further improved by carefully considering load balancing issues. Unfortunately, data migration approaches are not feasible for a CPU-GPU cluster implementation as they require the static scene textures in the GPU to be rebuilt. To overcome this limitation and provide more flexibility, the CPU can be used for ray traversal while GPU is used only for intersection tests.

REFERENCES

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[2] AMD. AMD "Close to Metal" Technology Unleashes the Power of Stream Computing. *http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_544~11 4147,00.html*, Last visited April 2008.

[3] Christian-A. Bohn. Kohonen feature mapping through graphics hardware. In Paul P. Wang, editor, *Proc. JCIS'98*, volume II, pages 64–67. Association for Intelligent Machinery, Inc, 1998.

[4] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. *Graphics Hardware (2002)*, pages 1–10, 2002.

[5] John G. Cleary, Brian M. Wyvill, Graham M. Birtwistle, and Reddy Vatti. Multiprocessor ray tracing. 5(1):3–12, March 1986.

[6] Daniel Cohen and Zvi Shefer. Proximity clouds: An acceleration technique for 3d grid traversal. *The Visual Computer*, 11:27–38, 1994.

[7] Ş. Alphan Es. *Accelerated Ray Tracing Using Programmable Graphics Pipelines.* Ph.d. thesis, Middle East Technical University, January 2008.

[8] D. Demarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed interactive ray tracing for large volume visualization, 2003.

[9] J. N. England. A system for interactive modeling of physical curved surface objects. *SIGGRAPH Comput. Graph.*, 12(3):336–340, 1978.

[10] Manfred Ernst, Christian Vogelgsang, and Günther Greiner. Stack implementation on programmable graphics hardware. In *VMV*, pages 255–262, 2004.

[11] Alphan Es and Veysi İşler. Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor. *J. Parallel Distrib. Comput.*, 67(11):1201–1217, 2007.

[12] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. *IEEE Computer Graphics and Applications*, 7(5):14–20, 2005.

[14] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[15] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 213–222, July 1984.

[16] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Comput. Graph. Appl.*, 9(6):12–26, 1989.

[17] Eric Haines. Bsp plane cost function revisited. *http://tog.acm.org/resources/RT News/html/*, Last visited May 2004.

[18] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[19] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, 1984.

[20] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 133–142, August 1986.

[21] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):133–142, August 1986.

[22] RapidMind Inc. Sh: A high-level metaprogramming language for modern GPUs. *http://libsh.org/*, Last visited April 2008.

[23] Veysi Isler, Cevdet Aykanat, and Bulent Özguç. An efficient parallel spatial subdivision algorithm for parallel ray tracing complex scenes. In *First Bilkent Computer Graphics Conference, ATARV-93*, Ankara, Turkey, 1993.

[24] Henrik Wann Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop*, pages 21–30, 1996.

[25] James T. Kajiya. The rendering equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150, August 1986.

[26] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *ACM Computer Graphics*, 20(4):269–278, 1998.

[27] III Kenneth E. Hoff, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[28] Stanford University Graphics Lab. BrookGPU. *http://graphics.stanford.edu/projects/brookgpu/index.html*, Last visited April 2008.

[29] Stephen Lacey and Richard Box. A fast, easy sort. *BYTE*, 16(4):315–ff., 1991.

[30] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics*, 24(4):327–335, 1990.

[31] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 1990.

[32] Kurt Menzel. Parallel Rendering Techniques for Multiprocessor Systems. In *Proceedings of the Spring School on Computer Graphics (SSCG '94)*, pages 91–103, Bratislava, Slovakia, 1994. Comenius University Press.

[33] Microsoft. DirectX. *http://www.gamesforwindows.com/en-US/AboutGFW/Pages/-DirectX10.aspx*, Last visited April 2008.

[34] Microsoft. HLSL Workshop. *http://msdn2.microsoft.com/en-us/library/bb173495 (VS.85).aspx*, Last visited April 2008.

[35] MPICH2. Message Passing Interface. *http://www.mcs.anl.gov/research/projects/mpich2/*, Last visited April 2008.

[36] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–, 1965.

[37] nVidia. CUDA. *http://developer.nvidia.com/object/cuda.html*, Last visited April 2008.

[38] nVidia. Cg. *http://developer.nvidia.com/page/cg_ main.html*, Last visited May 2008.

[39] OpenGL. OpenGL - The Industry Standard for High Performance Graphics. *http://www.opengl.org/*, Last visited April 2008.

[40] OpenGL. OpenGL Shading Language. *http://www.opengl.org/documentation/glsl/*, Last visited April 2008.

[41] OpenGL. WGL_ARB_pbuffer. *http://www.opengl.org/registry/specs/ARB/wgl_ p-buffer.txt*, Last visited December 2007.

[42] D. E. Orcutt. Implementation of ray tracing on the hypercube. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 1207–1210, New York, NY, USA, 1988. ACM.

[43] John Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA, 2005. ACM.

[44] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[45] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[46] Bui Tuong Phong. *Illumination for computer-generated images.* PhD thesis, 1973.

[47] Michael Potmesil and Eric M. Hoffert. The pixel machine: a parallel image computer. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 69–78, New York, NY, USA, 1989. ACM.

[48] Michael Potmesil and Eric M. Hoffert. The pixel machine: a parallel image computer. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 69–78, New York, NY, USA, 1989. ACM.

[49] POV-Ray. POV-Ray - The Persistence of Vision Raytracer. *http://www.povray.org/*, Last visited May 2008.

[50] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA, 2004. Adviser-Patrick M. Hanrahan.

[51] Erik Reinhard. Scheduling and data management for parallel ray tracing. Technical Report CS-EXT-1999-223, 1, 1999.

[52] Erik Reinhard and Frederik W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7):873–885, 1997.

[53] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann, and Amitabh Varshney. Real-time procedural textures. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 95–100, New York, NY, USA, 1992. ACM.

[54] J. Salmon and J. Goldsmith. A hypercube ray-tracer. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 1194–1206, New York, NY, USA, 1988. ACM.

[55] SGI. EXT_framebuffer_object. *http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt*, Last visited October 2007.

[56] Simon Green. The OpenGL Framebuffer Object Extension. *http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf*, Last visited October 2007.

[57] Milos Sramek and Arie Kaufman. Fast ray-tracing of rectilinear volume data using distance transforms, 2000.

[58] Niels Thrane and Lars Ole Simonsen. *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis, University of Aarhus, August 2005.

[59] Caglar Unlu. *Task Parallelism for Ray Tracing on a GPU Cluster.* Master's thesis, Middle East Technical University, February 2008.

[60] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[61] WikiPedia. GeForce 8 Series. *http://en.wikipedia.org/wiki/GeForce_8_Series*, Last visited May 2008.