2013

GPU ray tracing with CUDA

Thomas A. Pitkin
Eastern Washington University

GPU Ray Tracing with CUDA

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

For the Degree

Master of Science

By

Thomas A. Pitkin III

Fall 2013

THESIS OF THOMAS PITKIN APPROVED BY

_____ DATE_____

NAME OF CHAIR, GRADUATE STUDY COMMITTEE

_____ DATE_____

NAME OF MEMBER, GRADUATE STUDY COMMITTEE

MASTER'S THESIS

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Eastern Washington University, I agree that the JFK Library shall make copies freely available for inspection. I further agree that copying of this project in whole or in part is allowable only for scholarly purposes. It is understood, however, that any copying or publication of this thesis for commercial purposes, or for financial gain, shall not be allowed without my written permission.

Signature_____

Date_____

Abstract

Ray Tracing is a rendering method that generates high quality images by simulating how light rays interact with objects in a virtual scene. The ray tracing technique can accurately portray advanced optical effects, such as reflections, refractions, and shadows, but at a greater computational cost and rendering time than other rendering methods. Fortunately, technological advances in GPU computing have provided the means to accelerate the ray tracing process to produce images in a significantly shorter time. This paper attempts to clearly illustrate the difference in rendering speed and design by developing and comparing a sequential CPU and parallel GPU implementation of a ray tracer, written in C++ and CUDA respectively. A performance analysis reveals that the optimized GPU ray tracer is capable of producing images with speedup gains up to 1852X when compared to the former CPU implementation.

Acknowledgements

This project would not have been possible without the support of many people in the EWU Computer Science Department. I would like to gratefully and sincerely thank my committee chair, Dr. Bill Clark, for his guidance, encouragement, support, and understanding. His mentorship provided an enjoyable and rewarding experience while completing my graduate studies at Eastern Washington University. I would also like to thank my committee member, Stu Steiner, for his support and suggestions. Lastly, I would like to thank the graduate students, family members, and friends who provided me with the encouragement to continue working and finish my thesis in a timely manner.

Table of Contents

Table of Figures

1 Introduction

Ray tracing is a technique for rendering images in computer graphics by simulating how light rays interact with the virtual environment. By tracing the path of a light ray through a scene and emulating the effect of the ray as it intersects with virtual objects, the ray tracing algorithm can accurately portray reflections, refractions, shadows, dispersion, scattering, and other optical effects with a high degree of realism. Although the concept and application of ray tracing existed centuries before the invention of computers, it was not until the work of Arthur Appel (Appel, 1968) that the idea of using ray tracing was introduced in the field of computer graphics. Appel demonstrated that a computer can render a scene using ray casting, a limited form of ray tracing, by sending rays from a viewpoint through each pixel in the image plane. If a ray has intersected with an object in the scene, the color of the intersected object is returned and the pixel on the image plane is updated to this color. This technique accurately projects the 3D scene onto a 2D image. Over a decade later, Turner Whitted (Whitted, 1980) improved upon Appel's research and proposed the idea of a recursive algorithm where light rays reflected if a collision with an object was detected. If a ray has intersected with an object that has a reflective or refractive surface, a secondary ray could be generated at the intersection point and traced. This technique allows computers to render complex optical effects, such as reflection and refraction, and Whitted's algorithm became known as recursive ray tracing.

Although the ray tracing algorithm is relatively simple and widely used in computer graphics, it is rarely implemented in real-time applications. Because each ray must be tested for intersection against all the virtual objects in the scene, it becomes computationally expensive to render large scenes with complex light interactions. Therefore, video games and other graphic applications

where speed is critical cannot use ray tracing effectively. However, current GPUs now provide the computational power and a parallel framework to significantly improve the performance in graphical data calculation. By redesigning the recursive ray tracing algorithm to use iterative loops, ray tracing can be parallelized on the GPU's multiple cores and this results in a considerable gain in the rendering speeds of ray tracers.

1.1 Ray Tracing

Ray tracing is a powerful rendering method consisting of a series of simple algorithms. But, before these algorithms can be discussed, it is important to understand how the virtual camera and rays interact to simulate real world physics.

1.1.1 Rays and the Camera Model

Physical cameras function by capturing particles of light, called photons, and focusing them onto a piece of photographic film or an image sensor. Most modern cameras use a lens to converge the paths of photons into an image, but it is possible to recreate this process without a lens, as demonstrated by the pinhole camera model. This simple model consists of a light-proof box and photographic film. A small hole is created at the front of the box to allow photons to strike the film. A lens is not required because the small size of the hole allows only relatively few focused photons to hit any single point of the film. If the hole is too large, photons could enter the box at varying angles and strike the same point of the film. This would cause the final image to appear blurry and out of focus. This design prevents unwanted light rays from affecting the image, but, based on the position of the film and the pinhole, results in the image being inverted and reversed.

In computer graphics, this method of capturing an image has become popular as a simple design for the interaction between photons and the camera (Glassner, 1989).

The paths of photons are represented as rays in computer graphics. A ray consists of two elements: a point and a vector. The point stores the three-dimensional origin of the ray and the vector represents the direction that the ray is traveling. The mathematical equation for a ray is:

$$R(t) = O + t\boldsymbol{D}$$

where $R(t)$ is the ray, O is the origin, $\boldsymbol{D}$ is the normalized direction vector, and t is a scalar with a range of $[0, \infty)$ that corresponds with any point along the ray (Shirley & Morley, 2003).

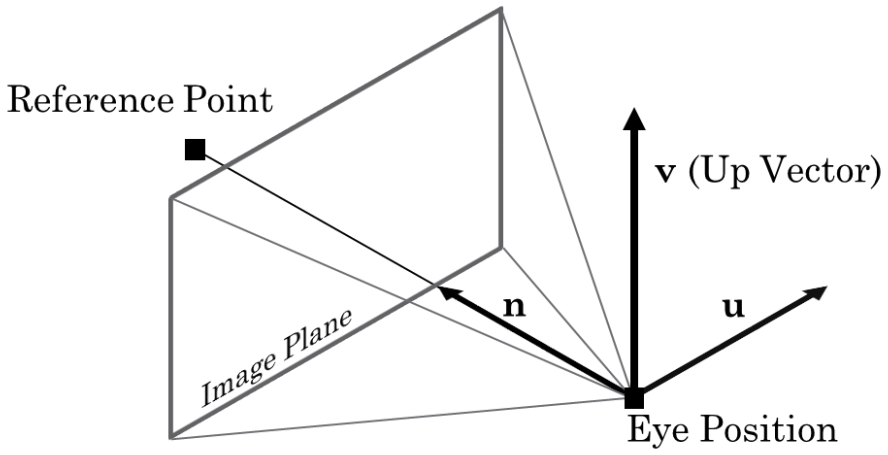

Figure 1 - Virtual Camera Model

The virtual camera model used to interact with these rays can be separated into five main parts: the eye position, the reference point, the up vector, the image plane, and the viewing frustum.

The eye position, reference point, and up vector are used to determine the camera's position and orientation. The eye position represents the camera location in 3D space. Similar to the hole of the pinhole camera, the eye position is the point where the rays converge. The reference point signifies the location at which the camera is pointing. By subtracting the eye position from

the reference point and normalizing the result, the view direction vector, also called the view plane normal vector, can be determined (Clark, 2013). The view direction vector defines the direction that the camera is facing. Perpendicular to the view direction vector is the up vector. The up vector points in the normalized direction that is directly up in the camera's orientation. The final orientation vector, the x-axis vector, can be found by calculating the cross product of the view direction vector and the up vector and normalizing the result. These three vectors act as the camera's yaw, pitch and roll, allowing the user to orient the camera in three dimensions. They are also used in the calculation of the viewing coordinate transform, which aligns the virtual camera with the world coordinate system.

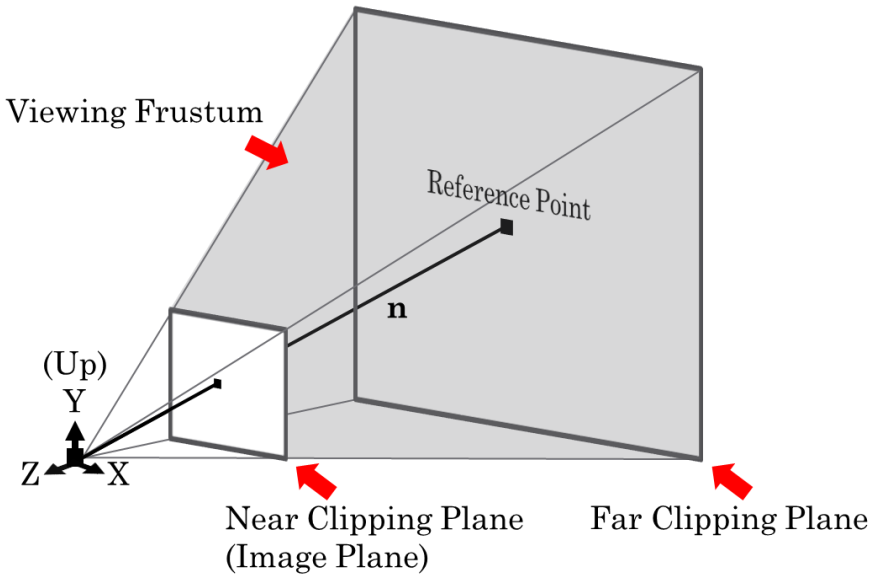

Figure 2 - Viewing Frustum

The viewing frustum simply represents the 3D volume that can be seen by the camera. It is defined as the volume created by two planes, the near and far clipping plane, located at two particular distances along the camera's view direction vector. Any objects outside of the viewing frustum and clipping planes will be "clipped" and not rendered in the final image. Every object

within the viewing frustum will be projected onto the final image, assuming no other objects obstruct the camera's view of said object.

The image plane is positioned in front of the eye position, inside the viewing frustum. The width and height of the image plane directly relates to the camera's field of view, the area that the camera can see. Even though its position differs from the pinhole camera model, the image plane still functions as the photographic film in the simulation. By excluding rays that do not collide with both the image plane and the eye position, the image plane can safely be placed in front of the eye position, before the rays converge. This results in a properly oriented image and removes an extra step in the ray tracing process (Glassner, 1989).

1.1.2 Backward Ray Tracing

In the physical world, millions of photons can be emitted from a single light source every second (Glassner, 1989). And statistically, only an extremely small number of these photons will be seen by the camera at any single point in time. Tracing every ray from a light source would result in a costly calculation with most of the operations providing no benefit to the final calculation of the image. It would be much more efficient to trace only the rays that intersects with both the eye position and the image plane. This can be achieved with a technique called backward ray tracing (Glassner, 1989). Instead of tracing rays from each light source, backward ray tracing emits rays (called primary rays) from the camera, through the image plane, and backwards into the scene. This method removes the unnecessary rays from the simulation, leaving only the rays that contribute to the final image.

1.1.3 Ray Generation

Before a ray tracer can send primary rays through specific points on the image plane and into the scene, the image plane must first be divided into a 2D grid of pixel locations. A pixel is the smallest component of a digital image, and each pixel represents a single color at a specific point in the image. To divide the image plane into pixel locations, the physical screen that is displaying the image needs to be mapped to the image plane. Because the physical screen is measured in pixels and the image plane is measured with generic units in world coordinates, this process will map the screen's pixels into world coordinate space as pixel locations. The first step in this process is to determine what the size of a single pixel will be on the image plane. This can be done by dividing the width of the image plane by the width of the physical screen and the height of the image plane by the height of the screen.

$$Pixel\ Width = \frac{Image\ Plane\ Width}{Screen\ Width}$$

$$Pixel\ Height = \frac{Image\ Plane\ Height}{Screen\ Height}$$

Once the pixel's dimensions have been calculated, a grid of pixel locations can be determined. The first pixel location starts at the bottom left corner of the image plane. This can be calculated by adding half of the pixel's width and height to the x and y coordinates of the image plane's bottom left corner respectively.

$$Pixel\ Center\ X = Image\ Plane\ Left\ X + \frac{Pixel\ Width}{2}$$

$$Pixel\ Center\ Y = Image\ Plane\ Bottom\ Y + \frac{Pixel\ Height}{2}$$

To place more pixel locations along the grid, the pixel center simply needs to be incremented by the width and height of a pixel. Adding the pixel width to pixel center x will step one pixel to the right and adding the pixel height to pixel center y will move one pixel up.

Using this method, a single sample ray tracer can step through the image plane and fire a single ray through the center of each pixel location. The generated rays sample the scene, allowing the ray tracer to estimate a color for each pixel, based on the information returned from the associated ray (Shirley & Morley, 2003).

1.1.4 Ray-Sphere Intersection

A ray traveling through the virtual scene has two possible actions: it can intersect with object(s) in the scene or it can miss every object. The ray tracer determines this by testing the ray against each object in the scene for intersections and calculating the closest intersection to the camera. But it is important to recognize that intersection tests are costly calculations and the number of tests should be minimized as much as possible. If a scene becomes too complex with a large amount of objects, the ray tracer's speed will significantly decrease during ray intersection testing. Techniques must be used to minimize the tests and provide the ray tracer with quick and efficient ways to determine a ray's state in the scene.

The method used in this paper for minimizing intersection tests places each object inside a spherical bounding volume. The volume is calculated to be the smallest spherical space that encloses the object, providing the ray tracer with a rough estimate of each object's three-dimensional volume (Clark, 2011). This method is used for the initial test between a ray and an object and can quickly determine if more intersection testing is required. If a ray does not intersect with the object's bounding sphere, the ray tracer can conclude that the ray will not intersect with any part of the object. This gives the ray tracer the ability to avoid unnecessary

intersection tests with objects that are not within a ray's path through the scene. Another benefit to using the bounding sphere method is that calculating the intersection between a ray and a sphere is a relatively quick and simple operation.

Any point P on the surface of a three-dimensional sphere with a center of (C_x, C_y, C_z) and radius R can be represented as:

$$(P_x - C_x)^2 + \left(P_y - C_y\right)^2 + (P_z - C_z)^2 = R^2$$

This same equation can be written in vector form:

$$(\boldsymbol{P} - \boldsymbol{C}) \cdot (\boldsymbol{P} - \boldsymbol{C}) = R^2$$

To calculate intersection points between a ray and a sphere, the ray tracer needs to test if any point P along the ray is also a point on the surface of the sphere. This can be done by substituting the points $P(t) = O + t\boldsymbol{D}$ of a ray into the sphere equation and solving for the values of t which produce points on the sphere's surface.

$$(O + t\boldsymbol{D} - C) \cdot (O + t\boldsymbol{D} - C) = R^2$$

Which can also be written as:

$$(\boldsymbol{D} \cdot \boldsymbol{D})t^2 + \big(2\boldsymbol{D} \cdot (O - \boldsymbol{D})\big)t + (O - C) \cdot (O - C) - R^2 = 0$$

This equation can now be rewritten as a quadratic formula:

$$At^2 + Bt + C = 0$$

where:

$A = \boldsymbol{D} \cdot \boldsymbol{D}$

$B = 2\boldsymbol{D} \cdot (O - C)$

$C = (O - C) \cdot (O - C) - R^2$

To solve for the unknown value of t, the quadratic equation gives:

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \qquad t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

where t_0 and t_1 are the two parameter values of the intersection points between the ray and sphere. The smallest positive value of t_0 and t_1 represents the closest intersection.

There is also an optimization that can be implemented when calculating the value of t. The algorithm can end the quadratic calculation early by checking if the discriminant $B^2 - 4AC$ is negative. If the discriminant is negative, than the ray has missed and there are no intersections between the ray and the sphere (Shirley & Morley, 2003). Also, if the sphere is being used only as a bounding volume, the algorithm can finish at this point even if the discriminant is positive. This is because the intersection point of a bounding sphere is not needed. The ray tracer only needs to know if the ray has intersected with the bounding sphere (Clark, 2011).

1.1.5 Ray-Triangle Intersection

Once a ray has intersected with a bounding sphere, the ray tracer can begin calculating the closest intersection point of the ray and the object that the sphere surrounds. There is still a chance that the ray will miss the object and no intersection points will be found. This is because the bounding sphere covers a larger volume than the object that it bounds and can lead to a false positive.

An object's volume is represented by a polygonal mesh consisting of vertices, edges, and faces. Each vertex designates a single three-dimensional point in the scene. Edges are formed from connecting two vertices and create a line in 3D space. Faces are simple convex polygons composed of edges and vertices to represent part of the surface of the polygonal mesh. This

paper uses triangular meshes with each face consisting of three vertices and edges (Shirley & Morley, 2003).

To calculate intersection points between a ray and an object, the ray tracer must test the ray against every triangular face that defines the object's mesh. Moller & Trumbore's ray-triangle intersection algorithm is used in this paper to perform the required calculations (Möller & Trumbore, 1997). This algorithm uses the barycentric coordinate system to define points within a triangle corresponding to masses placed at each vertex of the triangle.

A point $T(u, v)$ on a triangle with vertices (V_0, V_1, V_2) can be written as:

$$T(u, v) = V_0 + u\boldsymbol{E_0} + v\boldsymbol{E_1}$$

where (u, v) are the barycentric coordinates and $(\boldsymbol{E_0}, \boldsymbol{E_1})$ are the edges of a triangle expressed as vectors:

$$\boldsymbol{E_0} = V_1 - V_0$$

$$\boldsymbol{E_1} = V_2 - V_0$$

The point $T(u, v)$ is within the triangle if and only if:

$$u \geq 0$$

$$v \geq 0$$

$$u + v \leq 1$$

To calculate an intersection point between a ray and a triangle, the ray tracer needs to test if any point P along the ray is also a point within the barycentric coordinates of the triangle. This can be done by substituting the points $P(t) = O + t\boldsymbol{D}$ of a ray into the triangle equation and solving for the value of t which produces a point on the triangle's surface.

$$O + t\boldsymbol{D} = V_0 + u\boldsymbol{E_0} + v\boldsymbol{E_1}$$

This equation can be rearranged to:

$$O - V_0 = u\boldsymbol{E_0} + v\boldsymbol{E_1} - t\boldsymbol{D}$$

In matrix form, the equation appears as:

$$\boldsymbol{T} = \begin{bmatrix} -\boldsymbol{D} & \boldsymbol{E_0} & \boldsymbol{E_1} \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix}$$

where:

$$\boldsymbol{T} = O - V_0$$

The solution for (t, u, v) can be obtained by using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\boldsymbol{D} \quad \boldsymbol{E_0} \quad \boldsymbol{E_1}|} \begin{bmatrix} |\boldsymbol{T} & \boldsymbol{E_0} & \boldsymbol{E_1}| \\ |-\boldsymbol{D} & \boldsymbol{T} & \boldsymbol{E_1}| \\ |-\boldsymbol{D} & \boldsymbol{E_0} & \boldsymbol{T}| \end{bmatrix}$$

In linear algebra, it is known that:

$$\det(A, B, C) = |A \quad B \quad C| = -(A \times C) \cdot B = -(C \times B) \cdot A$$

This allows the equation to be rewritten as:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\boldsymbol{D} \times \boldsymbol{E_1}) \cdot \boldsymbol{E_0}} \begin{bmatrix} (\boldsymbol{T} \times \boldsymbol{E_0}) \cdot \boldsymbol{E_1} \\ (\boldsymbol{D} \times \boldsymbol{E_1}) \cdot \boldsymbol{T} \\ (\boldsymbol{T} \times \boldsymbol{E_0}) \cdot \boldsymbol{D} \end{bmatrix}$$

$$= \frac{1}{\boldsymbol{P} \cdot \boldsymbol{E_0}} \begin{bmatrix} \boldsymbol{Q} \cdot \boldsymbol{E_1} \\ \boldsymbol{P} \cdot \boldsymbol{T} \\ \boldsymbol{Q} \cdot \boldsymbol{D} \end{bmatrix}$$

where:

$$\boldsymbol{P} = (\boldsymbol{D} \times \boldsymbol{E_1})$$

$$\boldsymbol{Q} = (\boldsymbol{T} \times \boldsymbol{E_0})$$

Finally, (t, u, v) can be solved:

$$t = \frac{\boldsymbol{Q} \cdot \boldsymbol{E_1}}{\boldsymbol{P} \cdot \boldsymbol{E_0}}$$

$$u = \frac{\boldsymbol{P} \cdot \boldsymbol{T}}{\boldsymbol{P} \cdot \boldsymbol{E_0}}$$

$$v = \frac{\boldsymbol{Q} \cdot \boldsymbol{D}}{\boldsymbol{P} \cdot \boldsymbol{E_0}}$$

The ray-triangle intersection algorithm also includes multiple conditions that allow it to finish early if the algorithm determines that the ray will not intersect with the current triangle. The first condition checks the calculated determinant $\boldsymbol{P} \cdot \boldsymbol{E_0}$. If the determinant is equal to zero, than the ray is considered to be parallel to the triangle and does not intersect. Calculating the values of t, u, and v are not required in this case because the intersection test has already failed. A determinant with a negative value indicates that a ray may be intersecting with a triangle that faces away from the virtual camera, called a back facing triangle (Clark, 2008). Depending on if the ray tracer is set to ignore back facing triangles, this could act as another early exit for the algorithm.

When solving for the value of t, the algorithm must first determine that the barycentric coordinates u and v are within the boundaries of the triangle. If back facing triangles are ignored, the intersection point exists in the triangle if and only if:

$$\boldsymbol{P} \cdot \boldsymbol{E_0} \geq u \geq 0$$

$$0 \leq v$$

$$\boldsymbol{P} \cdot \boldsymbol{E_0} \geq u + v$$

If back facing triangles are included, the intersection point is in the triangle if and only if:

$$1 \geq u \geq 0$$

$$0 \leq v$$

$$1 \geq u + v$$

If these conditions are true, the algorithm will calculate a new t parameter value for the intersection point between the ray and the triangle. And if the value of t is less than a previously calculated t value associated with the same ray, then the smaller t value should replace the previous value as the ray's closest intersection point to the camera.

1.1.6 Phong Shading Model

After the closest intersection point of each ray has been found, the ray tracer can begin calculating the surface normals. The surface normal at the intersection point is a normalized vector that indicates a direction perpendicular to the surface at a single point (Glassner, 1989). The ray tracer uses the surface normal to determine the orientation of the surface and how light will reflect from it. A shading model is used to define how the surface normals will be calculated and, as a result, how the shading of a surface is blended from polygon to polygon (Clark, 2006a). This paper uses the Phong shading model to define how objects are shaded in the scene.

Developed by Bui Tuong Phong at the University of Utah, the Phong shading model interpolates surface normals across the triangles that form an object's surface and calculates a shade value based on each normal (Phong, 1975). By interpolating between triangles, the resulting surface normal will be the same along shared edges. This will give the impression that the surface is smooth and that no hard edges exist.

To create this effect, the Phong shading model begins by calculating a single normal vector for each triangle:

$$\boldsymbol{N} = \boldsymbol{E_0} \times \boldsymbol{E_1}$$

where $\boldsymbol{N}$ is the surface normal and $\boldsymbol{E_0}$ and $\boldsymbol{E_1}$ are two vector edges of the current triangle. Once the triangle normals have been calculated, the vertex normals can be determined. Since vertices are shared between the connected edges of triangles in a surface, each vertex normal is calculated by averaging the normals from each triangle that the vertex belongs to. The equation to calculate a vertex normal is:

$$\boldsymbol{N} = \frac{1}{n}(\boldsymbol{T_1} + \boldsymbol{T_2} + \cdots + \boldsymbol{T_n})$$

where $\boldsymbol{N}$ is the vertex normal, n is the number of triangles that the vertex belongs to, and $\boldsymbol{T_1} \ldots \boldsymbol{T_n}$ are the triangle normals that are being averaged.

The Phong shading model can now calculate the surface normal at an intersection point on an object's surface by interpolating between the vertex normals of the triangle containing the point. The weights to interpolate each vertex normal by are found from the u and v coordinates that are calculated during the previous ray-triangle intersection tests. The equation to calculate the surface normal at an intersection point is:

$$\boldsymbol{N_p} = (1 - u - v)\boldsymbol{V_0} + u\boldsymbol{V_1} + v\boldsymbol{V_2}$$

where $\boldsymbol{N_p}$ is the interpolated surface normal at the intersection point, u and v are the barycentric coordinates of the point, and $\boldsymbol{V_0}$, $\boldsymbol{V_1}$, and $\boldsymbol{V_2}$ are the three vertex normals of the triangle containing the point.

1.1.7 Phong Reflection Model

Once the surface normal at an intersection point is found, the ray tracer can begin generating a return color for the current ray. If a ray does not intersect with any objects, the return color is set to a predetermined background color. If an intersection is found, the return color represents the reflection of light based on the angle of the intersected triangle and ray with respect to the positions of the light sources in the scene. This paper implements the Phong reflection model to categorize several properties of the reflected light and define how the final color is calculated.

Also developed by Bui Tuong Phong, the Phong reflection model divides light reflections into three categories: ambient, diffuse, and specular light (Phong, 1975). Ambient light is used to model the indirect light that is reflected off of other objects within a scene. Diffuse light is used model direct light that is reflected by the surface of an object in all directions. And specular light is used to model direct light that is reflected by the surface of an object in a single direction. The Phong reflection model describes how the surfaces of objects in the scene reflect and absorb light by defining each surface with an ambient, diffuse, and specular reflection coefficient. The reflection coefficient stores a value between 0 and 1 of each of the three primary wavelengths: red, green, and blue. A value of 1 indicates that the specific wavelength is completely reflected by the surface. As the value decreases towards 0, the absorption of the surface increases and less light is reflected on this wavelength. A similar property is used to define the intensity of the wavelengths of light that strike an object's surface. The equations for calculating ambient, diffuse, and specular light use these properties to vary the intensity and color of the reflection of light on individual surfaces within the scene.

Because ambient light is defined as indirect light that comes from all directions, the orientation of the object's surface with respect to the viewer does not affect the intensity of the reflected ambient light. This makes the equation to calculate ambient light straightforward:

$$I_{pa} = K_a I_a$$

where I_{pa} is the reflected intensity of the ambient light at the reflection point, K_a is the surface's ambient reflection coefficient, and I_a is the intensity of the ambient light striking the surface.

Lambert's Law explains that the intensity of a diffusely reflecting surface is directly proportional to the cosine of the angle θ between the surface normal and the reflected ray of light. The reflection of diffuse light is maximal when the angle between the surface normal and the reflected light ray is close or equal to 0. To calculate the total diffuse light on a surface at single point, the equation is written as:

$$I_{pd} = K_d \sum_{i=1}^{m} I_i \, (\boldsymbol{N} \cdot \boldsymbol{L_i})$$

where I_{pd} is the total reflected intensity of the diffuse light at the reflection point, K_d is the surface's diffuse reflection coefficient, m is the number of light sources, I_i is the intensity of the diffuse light from light source i, $\boldsymbol{N}$ is the calculated surface normal vector at the reflection point, and $\boldsymbol{L_i}$ is the negated vector of the light ray from light source i. Unlike the equation for ambient light, a summation is used to include the diffuse intensity and light ray vector from each light source.

The position of the viewer becomes important when calculating specular reflection. Since specular light does not reflect in every direction, a specular reflection can only be seen by the

viewer if he/she is in a specific location to the reflecting ray of light. The area that that specular reflection can be seen in is determined by the shininess value of a surface. If the shininess value is large, the specular reflection area will be small with a high intensity of specular light, giving the surface a glossy appearance. As the shininess value decreases to 0, the specular intensity will decrease and the reflection area will grow in size until the surface appears to be flat / matte with no visible specular reflection. The equation to calculate the total specular light on a surface at a single point is:

$$I_{ps} = \sum_{i=1}^{m} I_i \, K_s(\boldsymbol{R} \cdot \boldsymbol{V})^n$$

where I_{ps} is the total reflected intensity of the specular light at the reflection point, m is the number of light sources, I_i is the intensity of the specular light from light source i, K_s is the surface's specular reflection coefficient, $\boldsymbol{R}$ is the vector of the reflecting ray of specular light, $\boldsymbol{V}$ is the vector of the view direction of the viewer, and n is the shininess value of the surface. To calculate the reflecting light vector $\boldsymbol{R}$, the negated light ray $\boldsymbol{I}$ needs to be orthogonally projected and reflected across the surface normal:

$$\boldsymbol{R} = \boldsymbol{I} - 2(\boldsymbol{I} \cdot \boldsymbol{N})\boldsymbol{N}$$

where $\boldsymbol{R}$ is the reflected light ray, $\boldsymbol{I}$ is the incoming light ray from the light source, and $\boldsymbol{N}$ is the calculated surface normal. This equation is derived from the law of reflection, which states that the angle between the incident ray and the normal is equal to the angle created by the reflected ray and the same normal.

By joining the ambient, diffuse, and specular equations together, the combined total reflected intensity of light at the reflection point can be determined:

$$I_{total} = K_a I_a + \sum_{i=1}^{m} I_i \left(K_d(\boldsymbol{N} \cdot \boldsymbol{L_i}) + K_s(\boldsymbol{R} \cdot \boldsymbol{V})^n\right)$$

However, this calculation assumes that every light source will affect each surface. This is not true as it is entirely possible that another object could block the incoming light rays from a light source and cast a shadow on the surface being calculated. Fortunately, determining the location of shadows in a scene is a straightforward process with ray tracing.

1.1.8 Shadows

Shadow computation can be included as an intermediate step when calculating the Phong reflection at an intersection point. By sending a ray from the intersection point to each light source, the ray tracer can perform intersection tests to check if the ray intersects with any objects in its path. If an intersection is detected, then an object exists between the light and the surface and will block incoming direct light sent from the light source. To create the effect of a shadow being cast on the surface, the blocked light source can be ignored when calculating the reflected intensity of diffuse and specular light on the surface. If no intersection is found, than the direction vector of the ray can be used to define the incoming light from the source in the diffuse and specular calculations (Clark, 2006b).

1.1.9 Specular Reflection

Surfaces with a specular reflection or refraction can also be simulated with ray tracing. If a ray intersects with a surface that is reflective or refractive, a subsequent ray must be sent from the surface to compute what color the surface may be reflecting or refracting. If this new ray intersects with another specular surface, an additional ray will be sent from this surface and the cycle will continue. This technique is known as recursive ray tracing because it implements a

recursive “tree” of reflection / refraction rays to perform the computation (Jensen & Christensen, 2007).

If a ray intersects with a surface that has a specular reflection, a reflection ray can to be generated using the law of reflection:

$$\boldsymbol{R} = \boldsymbol{I} - 2(\boldsymbol{I} \cdot \boldsymbol{N})\boldsymbol{N}$$

where $\boldsymbol{R}$ is the reflected light ray, $\boldsymbol{I}$ is the incoming light ray from the light source, and $\boldsymbol{N}$ is the calculated surface normal. The ray tracer can then recursively trace the reflection ray in the same way the primary rays are processed. Once a reflective ray in a set of recursive rays has intersected with a non-reflective surface, missed every object, or reached a maximum recursive limit, the ray tracer can begin stepping back through the recursive tree combining the calculated Phong reflection color returned by each ray. By combining the returned reflected light with the total reflected light of previous reflective surfaces, the surfaces will appear to reflect the surrounding objects and environment.

Since most objects will not have a 100% reflective surface, some light will be absorbed when reflecting on the surface. The light contributing from the reflection rays should be less than the incident amount of light on the reflective surface. To model this absorption, the ray tracer can use the reflectivity of a surface, between 0 and 1, to scale the reflected light and reduce the intensity of the reflection on the surface.

1.1.10 Specular Refraction

Surfaces with a specular refraction are computed in the same recursive manner as reflective surfaces. If a primary ray intersects with a refractive surface, a refractive ray will be created and

transmitted through the object. To generate a refractive ray, the ray tracer uses an equation derived from the law of refraction or Snell's law. Snell's law tells us that:

$$n \sin\theta = n_t \sin\theta'$$

where n is the index of refraction of the material of the incident ray, n_t is the index of refraction of the transmitting material, and θ and θ' are the angles of incidence and refraction.

And since:

$$\sin^2\theta + \cos^2\theta = 1$$

it can now be shown that:

$$\cos^2\theta' = 1 - \frac{n^2(1 - \cos^2\theta)}{n_t^2}$$

If $\boldsymbol{N}$ is the surface normal vector and $\boldsymbol{B}$ is a vector that is parallel to the surface, than these to unit vectors form a basis for the plane of refraction. A new equation can be written using these vectors:

$$\boldsymbol{T} = \sin\theta' \boldsymbol{B} - \cos\theta' \boldsymbol{N}$$

where $\boldsymbol{T}$ is the refracted light ray.

Similarly, the incident ray $\boldsymbol{D}$ and surface vector $\boldsymbol{B}$ can be found using the same basis:

$$\boldsymbol{D} = \sin\theta \boldsymbol{B} - \cos\theta \boldsymbol{N}$$

$$\boldsymbol{B} = \frac{\boldsymbol{D} + \boldsymbol{N}\cos\theta}{\sin\theta}$$

And finally, using substitution to solve for the refracted ray $\boldsymbol{T}$:

$$\boldsymbol{T} = \frac{n(\boldsymbol{D} + \boldsymbol{N}\cos\theta)}{n_t} - \boldsymbol{N}\cos\theta'$$

$$= \frac{n(\boldsymbol{D} - \boldsymbol{N}(\boldsymbol{D} \cdot \boldsymbol{N}))}{n_t} - \boldsymbol{N}\sqrt{1 - \frac{n^2(1 - (\boldsymbol{D} \cdot \boldsymbol{N})^2)}{n_t^2}}$$

1.2 Objective

The intent of this paper is to clearly illustrate the difference between CPU and GPU ray tracers in both speed and quality of the ray tracing algorithm. To meet this objective, the author will develop two ray tracers: a ray tracer which runs exclusively on the CPU and a GPU ray tracer based on the original CPU implementation. The CPU ray tracer will be written in C++ and implement a recursive ray tracing algorithm running on a single thread. The GPU ray tracer will utilize the Compute Unified Device Architecture (CUDA) by NVIDIA and replace the recursive ray tracing algorithm with a parallel algorithm that is iterative. Rendering tests of simple and complex scenes will be executed on both the CPU and GPU ray tracer and the performance of each will be measured.

1.3 Related Work

There has been a large amount of research in general-purpose computing on graphics processing units (GPGPU computing) and numerous papers can be found that are directed towards the improving the speed of ray tracing with GPGPU computing and parallelization. Some research, similar to this one, approaches the idea of ray tracing on GPU hardware as a study in parallelizing the ray tracing process and how it compares to sequential ray tracing (Britton, 2010; Budge et al., 2008; Segovia et al., 2009). Other papers target specific algorithms or data structures that can be used in ray tracing and implement new optimized structures suitable for GPGPU computing (Carr, Hoberock, Crane, & Hart, 2006; Santos, 2009). The results

from this research have shown that GPGPU computing is a powerful technology that can give a considerable performance gain to parallelizable applications, such as ray tracing.

1.4 CUDA

NVIDIA's CUDA was designed to run code in parallel by taking advantage of the streaming multiprocessors that are primarily used for graphics processing on GPUs. In 2007, the CUDA development environment became available for public use on NVIDIA's G80 graphics cards. Since then, CUDA has been made available for all G8x series GPUs and higher. This includes the GeForce, Quadro, and Tesla line of NVIDIA graphics cards. Each CUDA capable card has a specific compute capability, which represents the version of CUDA that is supported. Supported features, such as recursion and C++ functionality, and technical specifications can be determined from the compute capability of a card. Newer versions of CUDA provide developers with powerful features, such as function pointers, recursion, and C++ class functionality. When developing a CUDA accelerated application, it is important to understand how the programming model and GPU architecture work together to perform parallel computations.

1.4.1 Kernel Functions

In the CUDA programming model, a kernel function specifies code that should be executed in parallel. When a kernel function is executed, the kernel's code is run multiple times simultaneously with different sets of data. This parallel programming technique is called SPMD (single program, multiple data) (Kirk & Hwu, 2010) and is the primary method of parallelization in CUDA. CUDA-specific keywords *__global__*, *__device__*, and *__host__* are placed before a function declaration to indicate whether a function is a kernel, a device (GPU) function, or a host (CPU) function. It is important to recognize that kernels are not

functions and cannot return a value. They are subroutines that are callable from the host computer that execute on the CUDA device (Farber, 2012).

1.4.2 Threads

When a kernel is launched, a grid of GPU threads is generated to run the kernel's code in parallel. Each thread is responsible for its own set of data that will be computed with the executed code. The grid that is created by the kernel is typically comprised of thousands to millions of threads (Kirk & Hwu, 2010). At the top level, a grid can be organized into a one, two, or three-dimensional hierarchy of thread blocks. All blocks have a unique identifier (*blockIdx.x, blockIdx.y, blockIdx.z*) and contain the same number of threads. Each block is then organized into a three-dimensional array of threads where each thread has its own unique identifier (*threadIdx.x, threadIdx.y, threadIdx.z*). This organization of blocks and threads assists in defining what data set is associated with a thread and which streaming multiprocessor the thread will be executed on.

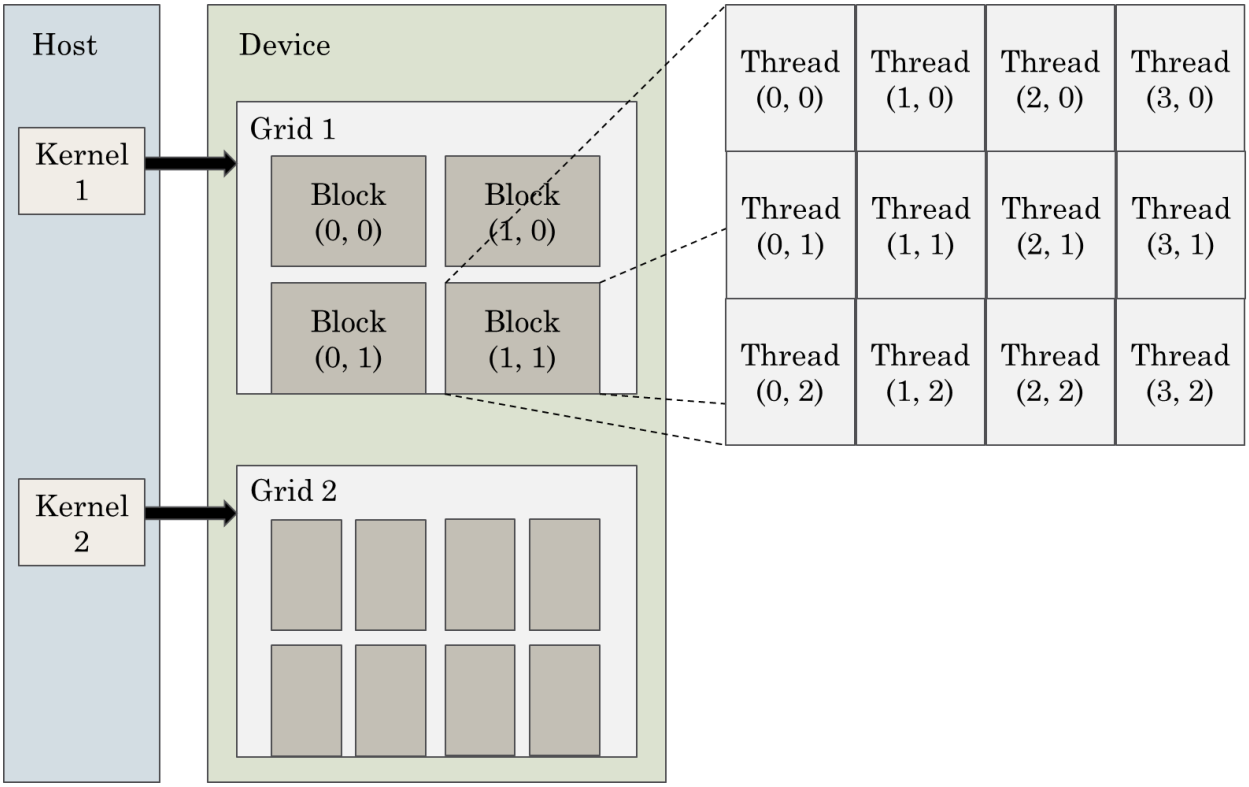

Figure 3 - CUDA Thread Organization

1.4.3 Memory Model

CUDA threads execute on a physically separate GPU device from the host computer that is running the application. This means that the GPU device needs to have its own multiprocessors, on-chip memory, and on-board memory that are separate from the host. Before a kernel launches, the data that each thread will be calculating needs to reside in the on-board global memory of the device. This can be done by allocating the required global memory space on the device and copying the data from the host memory, the RAM, to the newly allocated area before executing the kernel.

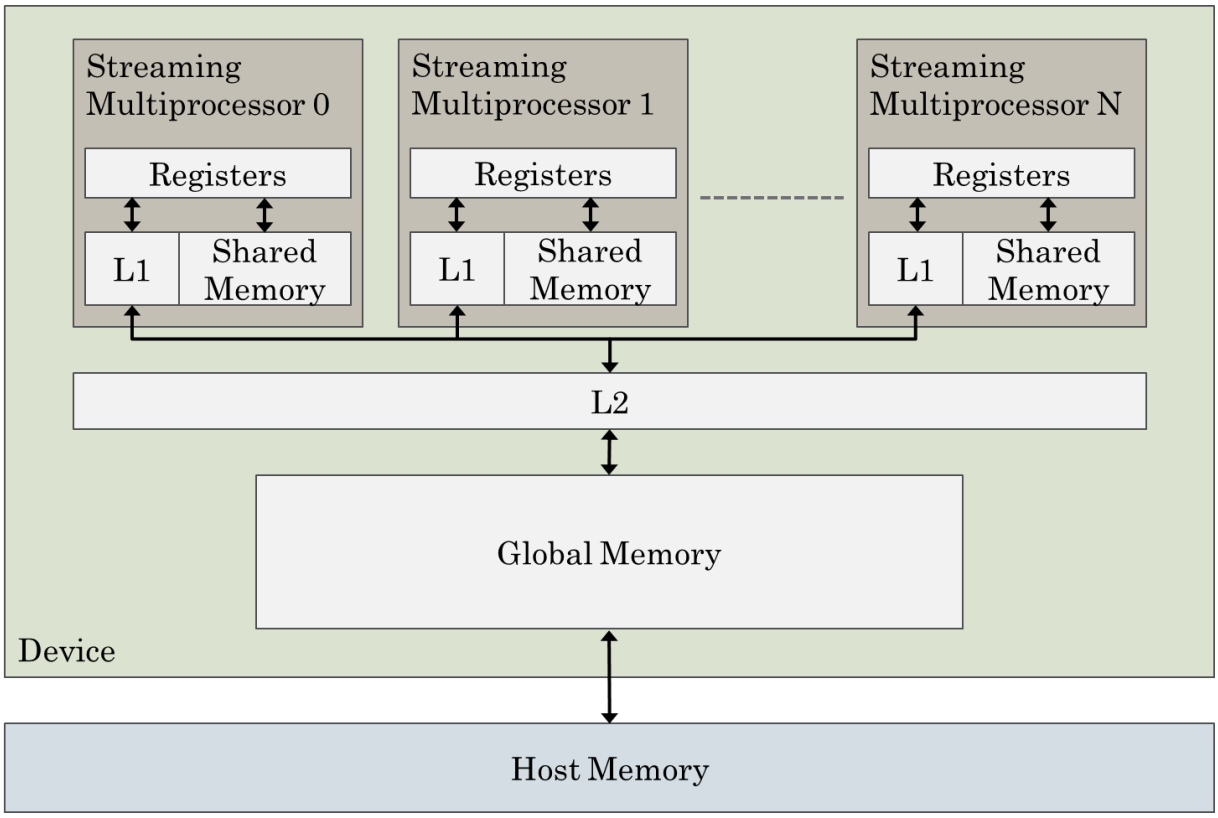

Figure 4 - CUDA Memory Hierarchy

Global memory is an on-board, shared memory system that is accessible by all streaming multiprocessors that are on the device (Farber, 2012). It is the most commonly used memory, providing the space to store large amounts of data on the GPU between kernel calls. Global memory is measured in gigabytes and is the largest but slowest type of memory available on a GPU. Because of its limited bandwidth, global memory cannot provide the streaming multiprocessors with enough data to keep them fully loaded (without stalls). To achieve peak performance, the data being processed must be moved to the streaming multiprocessor's on-chip memory.

The limited on-chip memory of a streaming multiprocessor is measured in kilobytes and it contains the fastest type of memory on the GPU. It is significantly faster than off-chip memory,

but on-chip memory cannot store the large amounts of data that can be placed in global memory. This makes it necessary to reuse data within the streaming multiprocessor to achieve optimal performance. Fortunately, there are multiple types of off-chip and on-chip memory and caches that provide various opportunities for data reuse.

On devices with a compute capability of 2.0 or higher, L1 and L2 caches are included to store certain types of memory and improve memory access requests and data reuse. The unified L2 cache sits between the off-chip and on-chip memory and caches all data loads and stores in a least recently used (LRU) fashion (Farber, 2012). This includes the initial memory copy from the host to the device. The L1 cache is designed for data reuse with spatial locality, not temporal locality like the L2 cache. Each streaming processor has its own L1 cache that is used to store local, on-chip data.

There are six CUDA memory types that will be discussed in this section: global, constant, texture, shared, local, and register memory.

Constant and texture memory, are bound to global memory in the same location. These two types of memory have extra characteristics and a small on-chip cache on each streaming multiprocessor that allow for better data reuse and quicker memory access than generic global memory. Constant memory is a read-only, off-chip memory type that can only be written to by the host. Once written to, constant memory is statically allocated within a file that resides in global memory. Constant memory provides an efficient way to broadcast read-only data to all threads on the GPU and store the data in the uniform cache on each streaming multiprocessor for reuse. Off-chip texture memory is generally used in visualization and is optimized for 2D spatially locality. If a series of threads request memory in a non-linear access pattern, it is

possible for texture memory to outperform global memory by finding nearby data in the 2D texture coordinate space and caching it in the texture cache on the streaming multiprocessor.

Register memory is the fastest memory on the GPU. This on-chip memory provides the threads on a streaming multiprocessor with enough bandwidth and low enough latency to reach peak performance (Farber, 2012). It is important to monitor how many registers each thread requires as there is a maximum number of registers per streaming multiprocessor. For instance, if 2048 threads are running on a streaming multiprocessor and there is a maximum of 32K 32-bit registers, then there can only be 16 registers allocated to each thread.

When a kernel uses more registers than are available on a streaming multiprocessor, the extra data is spilled onto the on-chip local memory. Local memory is bound to a section of the L1 cache. Spilled register memory and structures / arrays that are too large for registers are stored in local memory to minimize performance impact and reduce requests to global memory. The L1 cache also contains a section of shared memory. Shared memory is designed to allow threads within the same block on a streaming multiprocessor to share the designated memory. This provides threads with a method to communicate and collaborate on computations (Sanders & Kandrot, 2011). Shared memory also has the ability to multicast, allowing a single shared memory fetch to broadcast data to multiple requesting threads.

It is important to understand the pros and cons of each memory type when designing a CUDA application. By using the CUDA memory types to take advantage of data reuse, the threads on a streaming multiprocessor can experience minimal latency during computation and avoid stalling.

1.5 Parallelization with CUDA

The first step in converting a sequential program to a parallel program is to decide how a kernel will divide the task among thousands of GPU threads running in parallel. Thankfully, a ray tracer's design and data structure can be easily mapped to a parallel solution.

1.5.1 Thread Organization

Since rays can be computed independently from other rays in the ray tracer, it makes sense to assign each ray to a thread on the GPU. This allows each thread to compute a ray's reflection color without interfering with other threads and volatile memory.

A kernel's grid of blocks can also simplify how a thread's unique ID maps to a ray's position on the two-dimensional image plane. By defining a grid and block's hierarchy to be organized in two dimensions, a thread ID's x and y value, offset by the IDs and dimensions of the block and grid, match a ray's generic x and y positions. That is, a ray with $x = 0$ and $y = 0$ at the bottom left pixel position of the image plane would have a thread ID of $x = 0$ and $y = 0$ as well. This provides a kernel with a simple way to fetch the appropriate ray data for each thread based only on the thread, block, and grid information. This method also works if the rays are stored in a linear array. Here is the equation to calculate the x, y, and linear offset to the appropriate ray data associated with a thread:

$$x = threadIdx.x + blockIdx.x * blockDim.x$$

$$y = threadIdx.y + blockIdx.y * blockDim.y$$

$$linear\ offset = x + y * blockDim.x * gridDim.x$$

1.5.2 Kernel Complexity and Size

Since each thread can independently perform all necessary calculations for a ray, it is possible to write a single kernel for the entire ray tracer. But large, complex kernels, such as this one, can introduce issues that will not be noticeable until runtime.

The first issue is based on the performance and communication between the kernel and the host. When a kernel is launched, the host loses contact with the kernel as it runs on the GPU device. Unless specified to run asynchronously, the host waits for the kernel to finish before continuing to execute commands. The issue in this case comes from a kernel taking too long to complete its task on the GPU device. If a host loses contact with a running kernel for more than a few seconds, the host interrupts the device and cancels the kernel's operation. This is a safety measure that is implemented into all Nvidia drivers to prevent processes from running indefinitely on the GPU.

Another issue with using a single, complex kernel is a greater risk of register spilling and expensive conditional operations. If a single thread requires a large number of registers to run a complicated kernel program, then the execution of thousands of threads in parallel will cause the registers to spill to the L1 cache and a decrease in performance will occur. A complicated kernel program is also more likely to contain a large number of conditional statements. Unlike conditional statements in sequential programing, parallel conditional statements do not allow threads on a streaming multiprocessor to skip over branches of code. Each branch of each conditional must be evaluated, resulting in slower thread execution (Farber, 2012).

The solution to both of these problems is to divide a complex kernel into multiple smaller kernels or to call the kernel multiple time with a smaller set of data. Both solutions allow the kernel to complete faster and communicate with the host that it has finished in time.

The method used in this paper divides the parallel ray tracer into multiple, small kernels based on the different algorithms of a ray tracer (ray generation, triangle intersection, shading, etc.). Any kernels that execute complex algorithms, such as the triangle intersection kernel, are launched multiple times with smaller data sets.

1.5.3 Replacing Recursion

Recursion is a necessary process in classic ray tracing. Advanced effects, such as reflections and refractions, use recursion to temporarily store intermediate color values before calculating a final convex combination of colors. Although recursion is available on newer CUDA devices, the complexity of the recursive stack during ray tracing can cause unnecessary pressure on register and local memory. This can result in a large decrease in performance as threads are forced to wait for memory requests. Instead of relying on recursion, a parallel ray tracing algorithm should be written as an iterative loop, giving the developer complete control over the design and use of the stack.

This paper uses an iterative ray tracing algorithm outlined by Alejandro Segovia, Xiaoming Li, and Guang Gao (Segovia et al., 2009) to replace the original, recursive ray tracing algorithm. By viewing each recursive ray bounce as an iteration, a “for loop” is implemented into the parallel ray tracer to remove the necessity of recursion without losing functionality. A series of linear arrays in global memory are then used to store the intermediate colors and surface reflectivity values that are returned from the rays in each iteration. Once the iterative loop has completed, a convex combination generates a final color value for each index of the linear array series.

1.5.4 Caching Surface Data

Shared memory is a section of memory that is shared between all threads in a block on a streaming multiprocessor. It can be used to store data for all threads to access or to allow threads to communicate with other running threads in the same block. This work uses shared memory to optimize triangle intersection testing by caching data and reducing the number of global memory fetches.

During triangle intersection testing, each ray that has intersected with a bounding sphere of an object must perform intersection tests with every triangle that forms the surface of the object. This means that if multiple rays intersect the same bounding sphere of an object, the surface data of the object will be requested multiple times from global memory for each ray. To improve the speed of fetching a large amount of data from global memory, fetched surface data can be reused by caching the data in fast, on-chip shared memory.

To cache the object's surface data, each thread is responsible for fetching one set of triangle data from global memory and storing it into shared memory. Once the data has been stored, every thread in the block now has access to a subset of the object's surface data and can perform intersection tests without the need to reload the data for every ray. Because there is a limited amount of shared memory, this process repeats until all surface data has been loaded and tested for intersections. The decrease in requests to global memory results in an increase in speed during triangle intersection testing.

1.5.5 Optimizing Mesh Data

Memory size and speed is an important factor in GPU computing. For a parallel program to run efficiently when computing large sets of data, it is necessary for the data to be optimized for

quick access and computation. The host can assist in reducing the amount of calculations and global memory requests on the GPU by pre-processing large sets of data before they are copied to the GPU's memory. This paper uses a pre-processing optimization from Möller & Trumbore to reduce the necessary calculations required when testing for intersections between a ray and a triangle (Möller & Trumbore, 1997).

Möller & Trumbore explain that two edges of a triangle need to be calculated before the triangle can be tested against a ray for an intersection. This results in a reoccurring edge calculation for every ray that is tested against the same triangle. But an optimization can be made to prepare the triangle data for intersection tests and remove the reoccurring edge calculation. By pre-processing the edge calculation on the host, each triangle can be represented as a vertex and two edges in the data set that is copied to the GPU. This process removes a large number of operations during ray tracing and increases the speed and performance of the triangle intersection algorithm on the GPU.

1.6 Test Environment

The ray tracing benchmarks are performed on a desktop machine with the following specifications:

1.6.1 Operating System

Linux Distribution	Ubuntu Gnome Remix
Version	13.04

1.6.2 Software

Graphics API	OpenGL 4.1
Shading Language	GLSL 3.3
Application Framework	Qt 5.0.1
Application Programming Language	C/C++11
Application Compiler	gcc 4.4.7 g++ 4.7.3
CUDA Compiler	nvcc 5.5.0

1.6.3 CPU

Brand	Intel
Series	Core i7
Model	BX80601920
Socket Type	LGA 1366
Core	Bloomfield
Multi-Core	Quad-Core
Name	Core i7-920
Operating Frequency	2.66 GHz
QPI	4.8 GT/s
L2 Cache	4 x 256 KB
L3 Cache	8 MB
Manufacturing Tech	54 nm

1.6.4 GPU

Brand	Asus
Model	ENGTX570 DCII/2DIS/1280MD5
Interface	PCI Express 2.0 x16
Chipset Manufacturer	Nvidia
Name	GeForce GTX 570
Architecture	Fermi
Core Clock	742 MHz
CUDA Cores	480
Memory Clock	3800 MHz
Memory Size	1280 MB
Memory Interface	320-bit
Memory Type	GDDR5

2 Results

Figure 5 - A Reflective Render of Al, the Crocodile, and the Teapot Using the GPU Ray Tracer

The results of the ray tracing benchmarks have been divided into noteworthy iterations from the development process. Each benchmark requires the ray tracer to use one pass of rays to generate a 640 by 480 pixel image of three different objects. To test the scalability of the ray tracer, each object consists of a varying number of surfaces and triangles. It is important to note that each benchmark is displayed in a logarithmic scale and that, because of the iterative design, each multi-kernel benchmark includes all previous multi-kernel optimizations.

2.1 Single Kernel

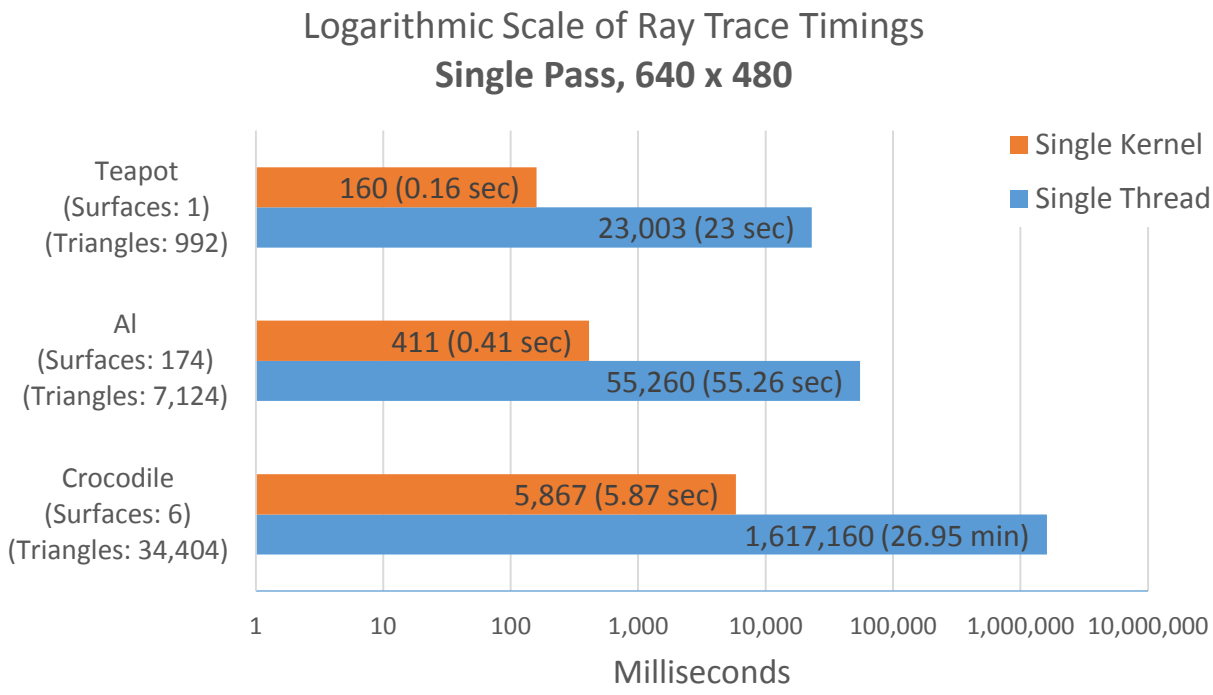

Figure 6 - Comparison of Single Thread and Single Kernel Timings

The initial iteration of the CUDA ray tracer was designed to run as a single kernel on the GPU. Although it showed a significant improvement in performance over the sequential ray tracer, the single kernel appeared unresponsive to the driver and was interrupted after a short period of time. Disabling this safety feature resulted in an unresponsive operating system while the benchmark was running.

2.2 Multi-Kernel

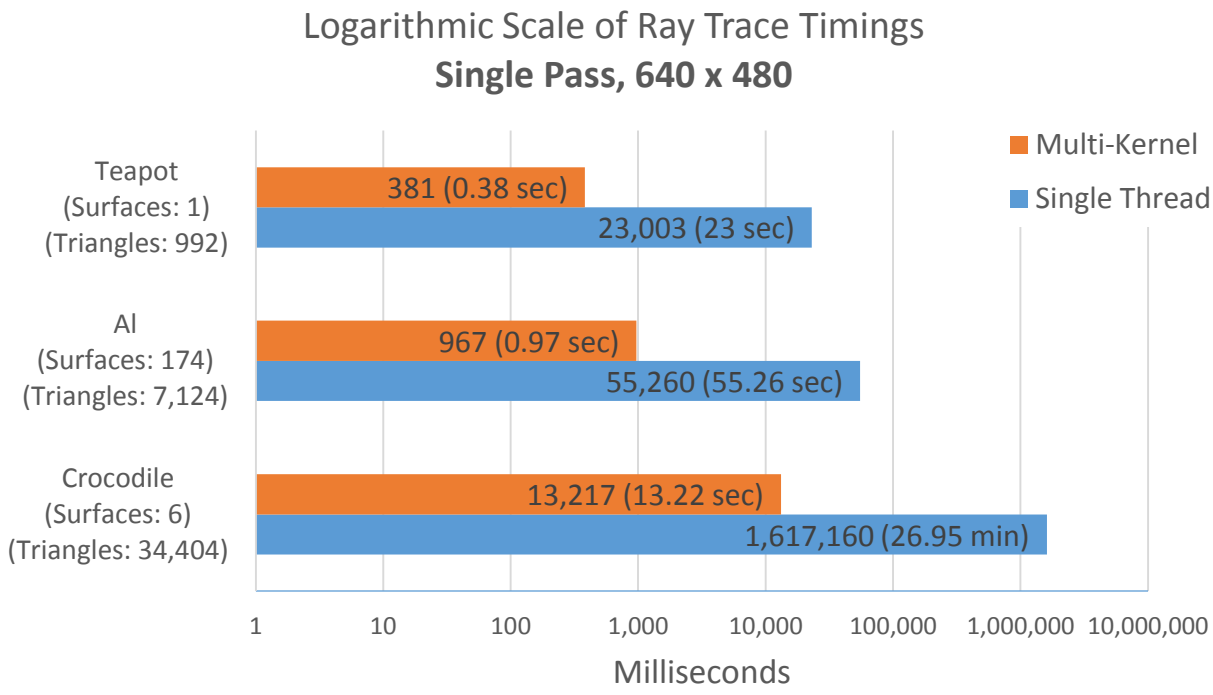

Figure 7 - Comparison of Single Thread and Multi-Kernel Timings

The first iteration of the multi-kernel benchmark required slightly more than twice the time to complete when compared to the single kernel benchmark. But, the multi-kernel ray tracer was still capable of ray tracing a scene significantly faster than the sequential ray tracer. By communicating with the host after short intervals, the multi-kernel ray tracer could run for long periods of time without interruption from the driver and allowed the user to communicate with the operating system without the system becoming unresponsive.

2.3 Multi-Kernel with Single-Precision Floating Point Arithmetic

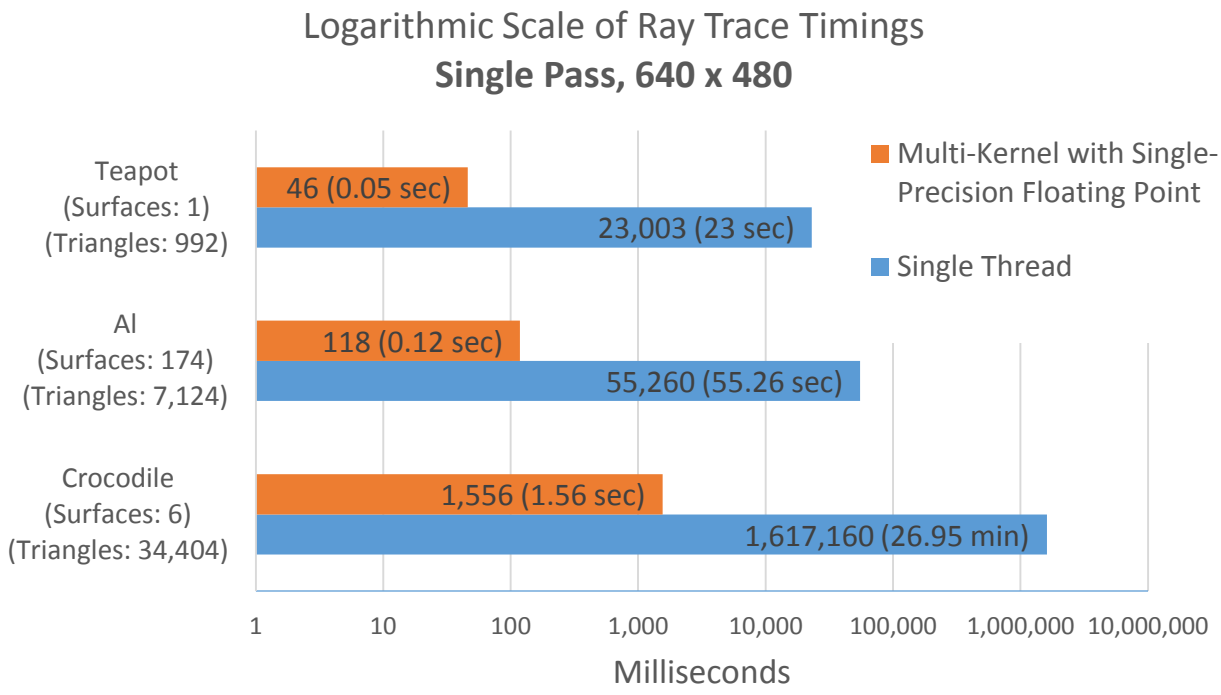

Figure 8 - Comparison of Single Thread and Multi-Kernel with Single-Precision Floating Point Timings

Originally, the ray tracer used double-precision floating point values to maintain a high level of precision during calculations. But, because of the large size of double-precision floating points, this resulted in an excessive amount of pressure on register and local memory on the streaming multiprocessors. It was eventually decided that double-precision computing was unnecessary and could be removed. By replacing the double-precision floating point values with single-precision floating points, the register and local memory could store twice as many values resulting in less memory requests and a large performance increase during calculations.

2.4 Multi-Kernel with Surface Caching

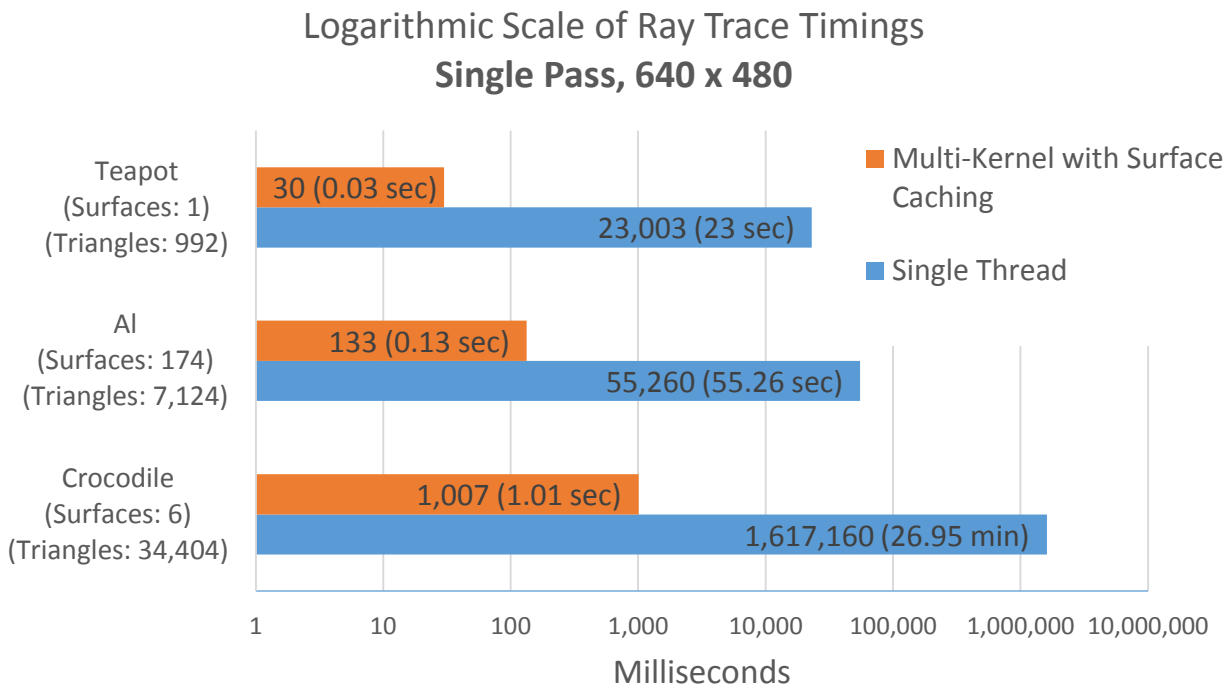

Figure 9 - Comparison of Single Thread and Multi-Kernel with Surface Caching Timings

This iteration of the multi-kernel implemented the surface caching technique that was discussed in section 1.5.4. By using the on-chip shared memory to store surface data for reuse with other threads, a slight increase in performance could be observed in the results of the benchmark. It should be noted that the "AI" object was the only item that did not benefit from surface caching. This is because an object that consists of many surface, each with a limited amount of triangles, requires the cache to update frequently and forces threads to wait for tasks, causing a decrease in performance.

2.5 Multi-Kernel with Intersection Optimization

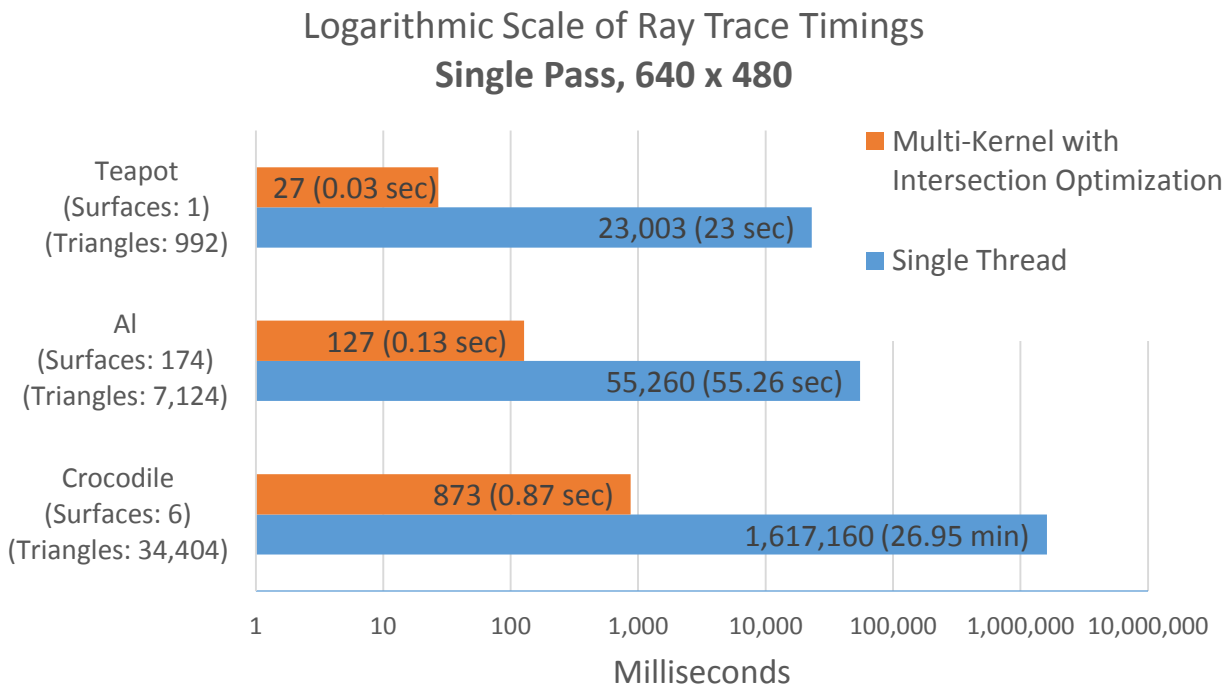

Figure 10 - Comparison of Single Thread and Multi-Kernel with Intersection Optimization Timings

The final iteration of the multi-kernel ray tracer optimizes the triangle mesh data for intersection, defined in section 1.5.5, to avoid costly triangle edge calculations during ray tracing. Based on the results of the benchmark, this optimization primarily targets objects consisting of a large number of triangles, such as the crocodile object.

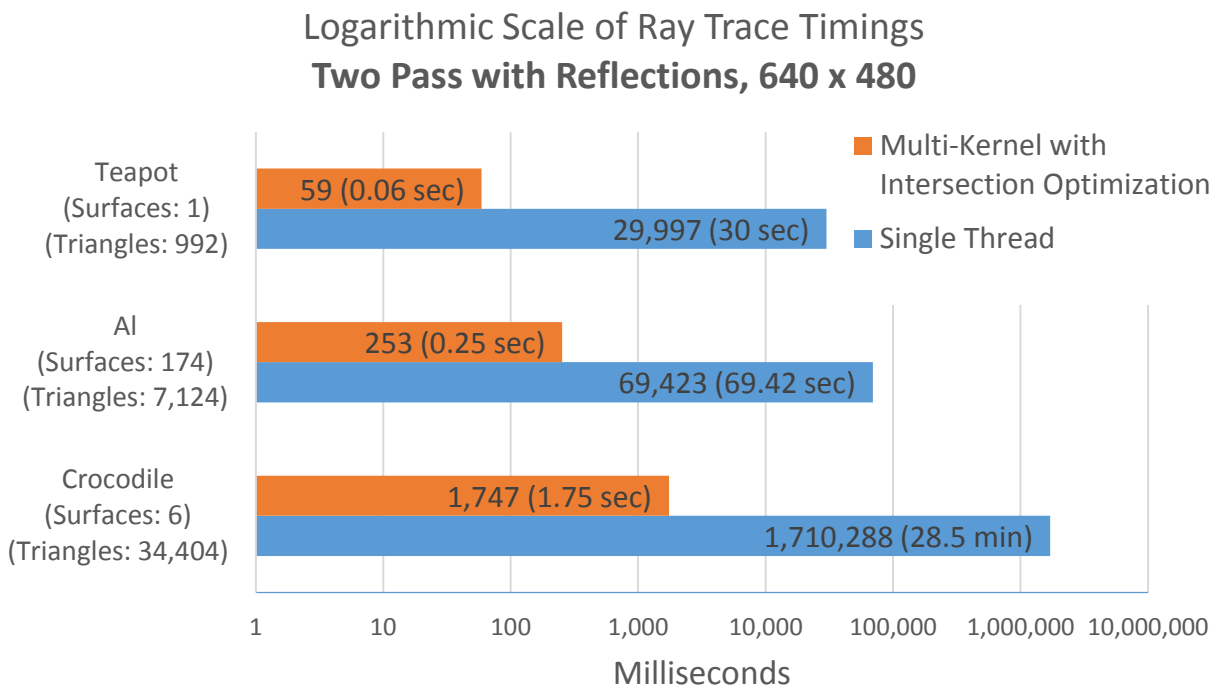

Figure 11 - Comparison of Single Thread and Multi-Kernel with Intersection Optimization Timings and Reflections Enabled

A second benchmark was also performed with the final version of the parallel ray tracer. This benchmark measured how much time the ray tracer required to generate a 640 by 480 pixel image of three different objects using two passes of rays with reflections enabled.

3 Conclusions and Future Work

By designing and analyzing a CPU and GPU based ray tracer, this paper has successfully demonstrated that there is a substantial improvement in performance by using GPGPU computing and CUDA for ray tracing. The benchmark results have shown that the parallel ray tracer on the GPU is capable of producing images up to 185,241% faster, a speedup of 1852X, when compared to the sequential ray tracer on the CPU.

There are, however, many optimizations that can still be implemented to further improve the performance and stability of the ray tracer.

Spatial partitioning is a technique used to divide Euclidean space into small, non-overlapping regions. Regions are then stored into a hierarchical data structure that can be quickly traversed for fast lookup of a region's contents. Common spatial partitioning data structures include BSP trees, quadtrees, octrees, and k-D trees. Currently, this paper uses a brute force technique for ray-triangle intersection testing with only a simple bounding volume to avoid unnecessary testing with other objects. If a full spatial partitioning structure was implemented, the triangular meshes of objects could be divided into small regions, each containing only a few triangles. This would result in each ray only being required to test against the minimal number of triangles that exist in the regions that the ray passes through, resulting in a dramatic improvement in the performance of ray-triangle intersection testing. One problem that must be overcome is the recursive nature of a spatial partitioning data structure. A parallel version of the data structure would need to be designed before a practical spatial partitioning system could be implemented. Fortunately, these data structure are already being researched and designed (Choi et al., 2010).

To increase stability and performance, it is important to design the ray tracer to dynamically adapt to the hardware in the system that it is running on. It is not uncommon for high performance desktops and servers to operate with multiple GPUs. By including the functionality to detect and use all available GPUs in a system, the ray tracer could easily scale to larger, more complex systems while taking full advantage of the system's hardware. Also, the ray tracer should adapt to the GPU that it is running on. Nvidia GPUs can differ in compute compatibility, number of processors, size of memory, features, etc. A GPU ray tracer should detect these differences and optimize its algorithms to run efficiently on the available hardware. The GPU ray tracer in this paper lacks this feature and was only optimized for the available GPU on the test

system. When running the ray tracer on a newer and faster GPU, a decrease in performance was observed because of the inefficient use of the increased on-chip memory and number of CUDA cores.

4 References

Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30--May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)* (p. 37). New York, New York, USA: ACM Press. doi:10.1145/1468075.1468082

Britton, A. (2010). *Full CUDA implementation of GPGPU recursive ray-tracing*. Purdue University. Retrieved from http://docs.lib.purdue.edu/techmasters/24/

Budge, B. C., Anderson, J. C., Garth, C., & Joy, K. I. (2008). A straightforward CUDA implementation for interactive ray-tracing. *2008 IEEE Symposium on Interactive Ray Tracing*, 178–178. doi:10.1109/RT.2008.4634641

Carr, N. A., Hoberock, J., Crane, K., & Hart, J. C. (2006). Fast GPU ray tracing of dynamic meshes using geometry images, 203–209. Retrieved from http://dl.acm.org/citation.cfm?id=1143079.1143113

Choi, B., Komuravelli, R., Lu, V., Sung, H., Bocchino, R. L., Adve, S. V, & Hart, J. C. (2010). Parallel SAH k-D tree construction. In *Proceedings of the Conference on High Performance Graphics* (pp. 77–86). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. Retrieved from http://dl.acm.org/citation.cfm?id=1921479.1921492

Clark, B. (2006a). Lighting and Shading. Retrieved from http://penguin.ewu.edu/cscd570/2011/PDFNotes/LightingShadingReview.pdf

Clark, B. (2006b). Additional Ray Tracing Features. Retrieved from http://penguin.ewu.edu/cscd570/2011/PDFNotes/RayTracingAdditionalFeatures.pdf

Clark, B. (2008). Polygon Intersection Techniques in Ray Tracing. Retrieved from http://penguin.ewu.edu/cscd570/2011/PDFNotes/Polygon-TriangleRayIntersection.pdf

Clark, B. (2011). Introduction to Ray Tracing - Ray Tracing Simple Spheres. Retrieved from http://penguin.ewu.edu/cscd570/2011/PDFNotes/IntroRayTracing-SphereIntersection.pdf

Clark, B. (2013). 3-D Viewing - The Camera Transform. Retrieved from http://penguin.ewu.edu/cscd470/Fall_13/PDFNotes/10-3DViewing.pdf

Farber, R. (2012). *CUDA application design and development*. Waltham, MA :: Morgan Kaufmann,. Retrieved from http://ewu.worldcat.org/title/cuda-application-design-and-development/oclc/760157354&referer=brief_results

Glassner, A. (1989). *An Introduction to ray tracing*. London: Academic Press. Retrieved from http://ewu.worldcat.org/title/introduction-to-ray-tracing/oclc/59104048&referer=brief_results

Jensen, H. W., & Christensen, P. (2007). High quality rendering using ray tracing and photon mapping. In *ACM SIGGRAPH 2007 courses on - SIGGRAPH '07* (p. 1). New York, New York, USA: ACM Press. doi:10.1145/1281500.1281593

Kirk, D. B., & Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1841511

Möller, T., & Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, (1), 1–7. Retrieved from http://www.tandfonline.com/doi/abs/10.1080/10867651.1997.10487468

Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, *18*(6), 311–317. doi:10.1145/360825.360839

Sanders, J., & Kandrot, E. (2011). *CUDA by example : an introduction to general-purpose GPU programming*. Upper Saddle River NJ: Addison-Wesley. Retrieved from http://ewu.worldcat.org/title/cuda-by-example-an-introduction-to-general-purpose-gpu-programming/oclc/535495666&referer=brief_results

Santos, A. dos. (2009). KD-Tree traversal implementations for ray tracing on massive multiprocessors: a comparative Study. *… , 2009. SBAC-PAD' …*, 41–48. doi:10.1109/SBAC-PAD.2009.25

Segovia, A., Li, X., & Gao, G. (2009). Iterative layer-based raytracing on CUDA. In *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International* (pp. 248–255). doi:10.1109/PCCC.2009.5403843

Shirley, P., & Morley, R. K. (2003). *Realistic Ray Tracing*. A. K. Peters, Ltd. Retrieved from http://dl.acm.org/citation.cfm?id=940410

Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM*, *23*(6), 343–349. doi:10.1145/358876.358882

VITA

Author: Thomas A. Pitkin III

Place of Birth: Richland, Washington

Undergraduate Schools Attended: Columbia Basin College,
Eastern Washington University

Degrees Awarded: Bachelor of Arts, Computer Science, 2011, Eastern Washington University

Honors and Awards: Graduated Magna Cum Laude, Eastern Washington University, 2011

Graduate Assistantship, Computer Science Department, 2011-2013,
Eastern Washington University