# Performance of Ray Tracing on Graphical Processing Unit

Wint Pa Pa Kyaw[1], Soe Mya Mya Aye[2]

## Abstract

Ray tracing in computer graphic is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. This paper presents image generation using ray-trace method by Graphical Processing Unit (GPU). Modern GPUs have enormous amounts of arithmetic processing power. If better graphic card is used, this method can be employed to generate real-time animation. This paper presents that the required memory bandwidth is reduced using constant memory rather than global memory. GPUs for parallel programming have emerged as a low-cost. Since modern graphics processors have high memory bandwidth as well as great computational power, running ray tracing on them achieves the speed. In this paper, ray tracing is demonstrated with implementation of high performance Compute Unified Device Architecture (CUDA). The implementation achieves the performance of traced rays in constant memory rather than global memory.

Keywords: Compute Unified Device Architecture (CUDA), Graphical Processing Unit (GPU), ray tracing, constant memory, global memory

## Introduction

The focus of scientists and researchers is enhancing live system efficiency, increasing speed, and making systems real-time with minimum error. So, they have made to achieve this, and CPU process speed has improved. Advances have been made to manufacture graphic cards with multicore processors and their parallel processing techniques are done using graphic cards. Using 3D virtual simulation systems for computer games and engineering sciences, entertainment software, astronavigation result in more realistic rendered images using different lighting techniques become an important subject to generate faster algorithms. Ray tracing method is one of these algorithms.

Ray tracing allows for dramatically more lifelike shadows and reflections. The algorithm takes into account where the light hits and calculates the interaction and interplay much like the human eye would process real light, shadows, and reflections. The way light hits objects in the world also affects which colours you see. In this paper, ray tracing algorithm is implemented by the function of Compute Unified Device Architecture (CUDA) library on graphic card to compare constant memory and global memory maximum ability. Programming on the Graphics Processing Unit (GPU) is performed with CUDA 5.5. The use of the CUDA programming interface requires the use of NVIDIA graphics card. The limitation is the best limitation of reducing the amount of data transferred between the GPU and the CPU.

## Basic Concepts of Graphics Processing Unit

The Graphics Processing Unit on modern graphics cards offers the possibility of accelerating intensive tasks. By splitting the work into a large number of independent jobs, speedups are reported. Graphics cards involving GPUs perform local, dedicated, massively parallel stochastic simulation. GPUs were developed as dedicated devices to aid in real-time graphics rendering. GPUs have evolved into many-core processing units, with up to 30 multiprocessors per card, in response to commercial demand for real-time graphics rendering, independently of demand for many-core processors in the scientific computing community. As such, the architecture of GPUs is very different to that of conventional CPUs.

---

[1] Professor, Dr., Department of Computer Studies, University of Yangon

[2] Professor (Head), Dr., Department of Computer Studies, University of Yangon

The CUDA is platform for parallel computing using special GPU by NVIDIA. This platform allows software developers to highly parallel algorithms on graphic units. A CUDA has a number of different memory components that are registered, shared memory, local memory, global memory and constant memory. Figure 1 illustrates how threads in the CUDA can access the different memory components. In CUDA, threads and the host can access memory. One and two-way arrows are used to indicate read and write capability. An arrow pointing away from a memory component indicates read capability; an arrow pointing toward a memory component indicates write capability. For example, global memory and constant memory can be read or written.

Multiple kernels can be invoked on a CUDA. Each kernel is an independent grid consisting of one or more blocks. Each block has its own per-block shared memory, which is shared among the threads within that block. All threads can access different parts of memory on the device. The host is used to transfer data to and from global memory and to transfer data to and from constant memory. Once the data is in the device memory, threads can read and write different parts of memory: thread register; thread local memory; block shared memory; grid global memory; grid constant memory. Constant memory is used for data that will not change over the course of a kernel execution and is read-only. Using constant rather than global memory can reduce the required memory bandwidth, however, this performance gain can be realized when a warp of threads reads the same location.

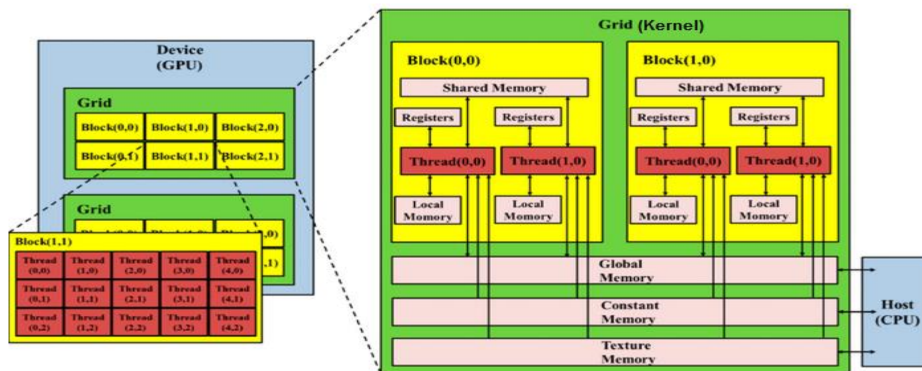


Figure 1: Schematic of CUDA Memory Hierarchy in GPU

## Ray Tracing Method

A mathematical object that contains both a point and a vector is a ray as shown in figure 2.

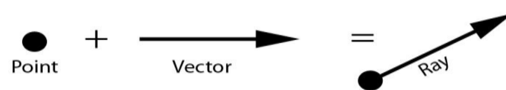


Figure 2: The Construction of a Ray

Rays are the backbone of ray-tracing. It is these rays that are cast from the eye into the world. These rays are calculated to check intersections with objects. Where an intersection exists, the ray returns colour information. Ray tracing is a method of generating realistic images, in which the paths of individual rays of light are followed from the viewer to their points of origin. The main concept of ray tracing algorithm is to find intersections of a ray with a scene consisting of a set of geometric primitives. The scene consists of a list of geometric primitives, such as polygons, spheres, cones, etc. Any kind of object can be used as a ray

tracing primitive as long as it is possible to compute an intersection between a ray and the primitive.

Each pixel of the image should be the same colour and intensity as the ray that hits that sensor. Each pixel behaves something like an eye that is looking into the scene. These rays being cast out of each pixel and into the scene are illustrated in Figure 3. The pixel would see this object and can assign its colour based on the colour of the object it sees. Most of the computation in ray tracing is the computation of these intersections of the ray with the objects in the scene.

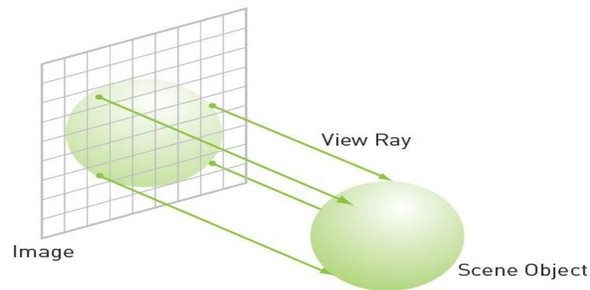


Figure 3: A Simple Ray Tracing Scheme

In-game lighting effects become more and more realistic over the years, but the benefits of ray tracing are less about the light itself and more about how it interacts with the world. With enough computational power available, it is possible to produce realistic images that are indistinguishable from real life. Ray tracing is used extensively when developing computer graphics imagery for films and TV shows.

## Implementation Ray Tracing

A GPU version of the ray tracing algorithm is implemented using NVIDIA's CUDA, which extends the standard C language with functionality to program graphics cards. Ray tracer support scenes of circles and the camera are restricted to the z-axis, facing the origin. Each circle is assigned a colour and then shades them with some pre-computed function if they are visible. The ray tracer fires a ray from each pixel and keeps track of which rays hit which circles. It tracks the depth of each of these hits. The circle closest to the camera can be seen in the case where a ray passes through multiple circles.

Circles are modelled with a data structure that stores its colour of (r,b, g), its radius, and the circle’s centre coordinate of (x, y, z). If the ray does intersect the circle, the hit method computes the distance from the camera where the ray hits the circle with the following code segment.

```
__device__ float hit ( float xi, float yi, float *n )
{
    float dx = xi - x;      float dy = yi - y;
            if (dx*dx + dy*dy < radius*radius)
             {
                float dz = sqrtf( radius*radius - dx*dx - dy*dy );
                *n = dz / sqrtf( radius * radius );     return dz + z;
             }     return -INF;
}
```

The data need to allocate on the GPU but are generating it with the CPU. So both "a cudaMalloc( )" and "a malloc( )" are used to allocate memory on both the GPU and the CPU. Memory is allocated for input data, as an array of circles with the following code segment.

```
HANDLE_ERROR( cudaMalloc( (void**)&s, sizeof(Circle) * CIRCLES ) );
```

Memory on the GPU for output image is allocated with the following code segment.

```
HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );
```

The centre coordinate, colour, and radius for circles are randomly generated with the following code segment.

```
Circle *temp_s = (Circle *)malloc( sizeof(Circle) * CIRCLE );
for (int i=0; i< CIRCLE; i++)
   {
            temp_s[i].r = rnd( 1.0f ); temp_s[i].g = rnd( 1.0f );temp_s[i].b = rnd( 1.0f );
             temp_s[i].x = rnd( 1000.0f ) - 400;temp_s[i].y = rnd( 1000.0f ) - 400;
            temp_s[i].z = rnd( 1000.0f ) - 400;temp_s[i].radius = rnd( 100.0f ) + 40;
   }
```

The array of circles are copied to the GPU using cudaMemcpy() and then free the temporary buffer with the following code segment.

```
HANDLE_ERROR( cudaMemcpy( s, temp_s,sizeof(Circle) *CRCLE,cudaMemcpyHostToDevice ));
free( temp_s );
```

The following code segment is that a bitmap from the circle data is generated with the kernel.

```
dim3 grids(D/16,D/16); dim3 threads(16,16); kernel<<<grids,threads>>>( dev_bitmap );
```

Finally, the output image is copied from the GPU and displays it with the following code segment.

```
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),dev_bitmap,bitmap.image_size(),
cudaMemcpyDeviceToHost));
```

Each thread generates one pixel for the output image by computing the x- and y-coordinates as well as the linearized offset into the output buffer. Image coordinates (x,y) are shifted by D/2 so that the z-axis runs through the center of the image with the following code segment.

```
int x = threadIdx.x + blockIdx.x * blockDim.x;int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x; float xi = (x - D/2);float yi = (y - D/2);
```

Since each ray needs to check each circle for intersection, through the array of circles are iterated, checking each for a hit. The following code segment is to check each circle for intersection.

```
for(int i=0; i<CIRCLE; i++)
{
    float n;     float t = s[i].hit( xi, yi, &n );
            if (t > maxz)
            {
              float fscale = n;    r = s[i].r * fscale; g = s[i].g * fscale; b = s[i].b * fscale;
            }
}
```

Through each of the input circles are iterated and called its hit () method to determine whether the ray from the pixel sees the circle. If the ray hits the current circle, whether the hit is closer to the camera than the last circle one hit are determined. If it is closer, this depth is stored as new closest circle. The colours associated with this circle are stored so that when the loop has terminated, the thread knows the colour of the circle that is closest to the camera. Since this is the colour that the ray from the pixel sees, this is the colour of the pixel and stores this value in the output image buffer. After every circle has been checked for intersection, the current colour is stored into the output image.

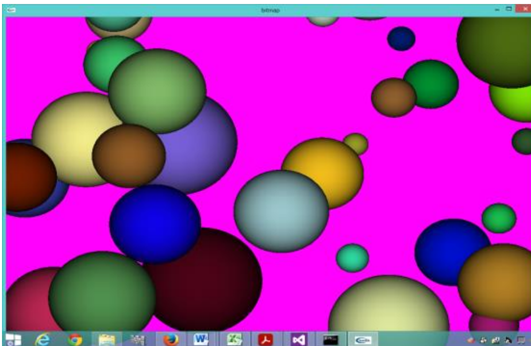

Figure 4(a) A Screenshot from the Ray Tracing with 40 Circles

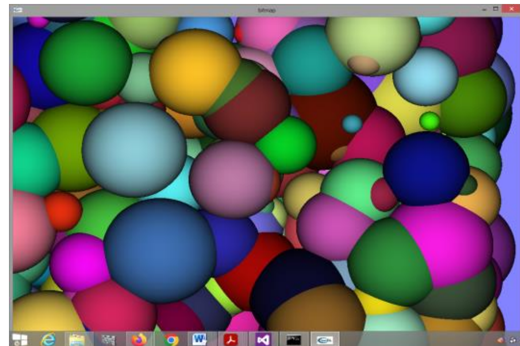

Figure 4(b) A Screenshot from the Ray Tracing with 900 Circles

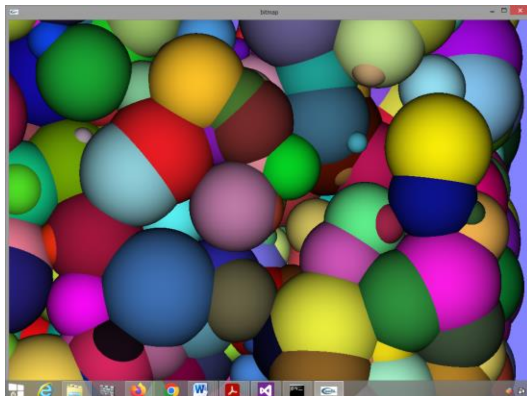

Figure 4 (c) A Screenshot from the Ray Tracing with 1500 Circles

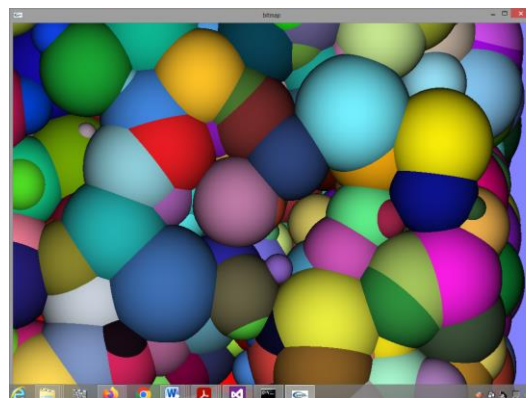

Figure 4(d) A Screenshot from the Ray Tracing with 2000 Circles

Figure 4 (a, b, c, d) shows result of creates an image by ray trace technique when rendered with 40, 900, 1500 and 2000 circles. The output image should be pretty obvious image. So the output image is to be improved using the constant memory on GPU.

The following code segment is the mechanism for declaring memory constant.

```
__constant__ Circle s[CIRCLES];
```

The following code segment is to copy the array of circles from host memory to constant memory on the GPU. The versions use constant memory.

```
HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s, sizeof(Circle) * CIRCLES ) );
```

The following code segment is to copy the array of circles to the GPU using cudaMemcpy(). The versions use without constant memory.

```
HANDLE_ERROR(cudaMemcpy( s,temp_s,sizeof(Circle)*CIRCLES,cudaMemcpyHostToDevice));
```

The differences between cudaMemcpyToSymbol() and cudaMemcpy() using cudaMemcpyHostToDevice are that cudaMemcpyToSymbol() copies to constant memory and cudaMemcpy() copies to global memory.

## Results and Discussion

Multiple tests for CUDA C program and ray tracing are performed. The results are observed for run with various numbers of objects as shown in Table 1, 2, 3, 4, 5, 6. These tables represent to the timed data for different GPUs (NVIDIA GeForce 920M and NVIDIA GeForce GT 750M). All units of time are measured in milliseconds.

Table 1: List of GPUs and render times using constant memory for number of object 900

| **Graphics Card** | **GT 750M(ms)** | **920M(ms)** |
|---|---|---|
| Fram 1 | 303.7 | 286.9 |
| Fram 2 | 295.1 | 287.8 |
| Fram 3 | 299.9 | 287.1 |
| Fram 4 | 292.4 | 287.5 |
| Fram 5 | 298.2 | 287 |
| Fram 6 | 297 | 288 |
| Fram 7 | 296.7 | 287.1 |
| Fram 8 | 299.7 | 288.5 |
| Fram 9 | 303.5 | 287.6 |
| Fram 10 | 296.3 | 287.2 |
| Total | 2982.5 | 2874.7 |
| **Average** | **298.25** | **287.47** |

Table 1 presents the timed data using constant memory for number of object 900. When the program is executed with a corei7 multiprocessor and NVIDIA's GPU (GeForce GT 750M), the render times are obtained at 298.25ms. When the program is executed with a corei3 multiprocessor and NVIDIA's GPU (GeForce 920M), the render times are obtained at

287.47ms. It can be observed that GPU (GeForce 920M) time is faster than GPU (GeForce GT 750M) time using constant memory for number of object 900.

Table 2: List of GPUs and render times using Global memory for the number of object 900

| Graphics Card | GT 750M **(ms)** | 920M**(ms)** |
|---|---|---|
| Fram 1 | 367.5 | 341.4 |
| Fram 2 | 375.5 | 340.9 |
| Fram 3 | 381.9 | 341.1 |
| Fram 4 | 381.5 | 341.5 |
| Fram 5 | 379.6 | 341.2 |
| Fram 6 | 372.8 | 341.1 |
| Fram 7 | 376.9 | 341.6 |
| Fram 8 | 375.8 | 340.9 |
| Fram 9 | 365.9 | 341 |
| Fram 10 | 367.2 | 341.9 |
| Total | 3744.6 | 3412.6 |
| Average | 374.46 | 341.26 |

Table 2 shows the timed data using Global memory for the number of object 900. When the program is executed with a corei7 multiprocessor and NVIDIA's GPU (GeForce GT 750M), the render times are obtained at 374.46ms. When the program is executed with a corei3 multiprocessor and NVIDIA's GPU (GeForce 920M), the render times are obtained at 341.26ms. It can be observed that the average GeForce 920M render time is faster than the average GeForce GT 750M render time using Global memory for number of object 900.

Table 3: List of GPUs and render times using constant memory for number of object 1500

| **Graphics Card** | **GT 750M(ms)** | **920M(ms)** |
|---|---|---|
| Fram 1 | 479 | 480.8 |
| Fram 2 | 480.8 | 481.1 |
| Fram 3 | 477.6 | 481.6 |
| Fram 4 | 470.1 | 480.7 |
| Fram 5 | 481.6 | 481.4 |
| Fram 6 | 482.6 | 481.1 |
| Fram 7 | 485.7 | 481.2 |
| Fram 8 | 470.6 | 480.7 |
| Fram 9 | 479.3 | 481.2 |
| Fram 10 | 484.8 | 482.1 |
| Total | 4792.1 | 4811.9 |
| **Average** | **479.21** | **481.19** |

Table 3 presents the timed data using constant memory for the number of object 1500. When the program is executed with a corei7 multiprocessor and NVIDIA's GPU (GeForce GT

750M), the render times are obtained at 479.21ms. When the program is executed with a corei3 multiprocessor and NVIDIA's GPU (GeForce 920M), the render times are obtained at 481.19ms. It can be observed that GPU (GeForce GT 750M) time is faster than GPU (GeForce 920M) time using constant memory for number of object 1500.

Table 4 shows the timed data using Global memory for the number of object 1500. When the program is executed with a corei7 multiprocessor and NVIDIA's GPU (GeForce GT 750M), the render times are obtained at 607.17ms. When the program is executed with a corei3 multiprocessor and NVIDIA's GPU (GeForce 920M), the render times are obtained at 564.33ms. It can be observed that the average GeForce 920M render time is faster than the average GeForce GT 750M render time using Global memory for the number of object 1500.

Table 4: List of GPUs and render times using Global memory for the number of object 1500

| **Graphics Card** | **GT 750M(ms)** | **920M(ms)** |
|---|---|---|
| Fram 1 | 600.9 | 565.6 |
| Fram 2 | 616.1 | 564.2 |
| Fram 3 | 604.3 | 563.7 |
| Fram 4 | 614.4 | 563.6 |
| Fram 5 | 602.3 | 565.2 |
| Fram 6 | 616.9 | 563.5 |
| Fram 7 | 607.9 | 563.7 |
| Fram 8 | 599 | 564.8 |
| Fram 9 | 602.4 | 564.6 |
| Fram 10 | 607.5 | 564.4 |
| Total | 6071.7 | 5643.3 |
| **Average** | **607.17** | **564.33** |

Table 5: List of GPUs and render times using constant memory for the number of object 2000

| **Graphics Card** | **GT 750M(ms)** | **920M(ms)** |
|---|---|---|
| Fram 1 | 627.4 | 639.3 |
| Fram 2 | 621 | 638.1 |
| Fram 3 | 620.8 | 638 |
| Fram 4 | 620.4 | 637.9 |
| Fram 5 | 621.1 | 637.3 |
| Fram 6 | 621.2 | 636.7 |
| Fram 7 | 624.4 | 637.9 |
| Fram 8 | 622.4 | 638.1 |
| Fram 9 | 621.8 | 638.4 |
| Fram 10 | 621.7 | 639.6 |
| Total | 6222.2 | 6381.3 |
| **Average** | **622.2** | **638.13** |

Table 5 presents the timed data using constant memory for the number of object 2000. When the program is executed with a corei7 multiprocessor and NVIDIA's GPU (GeForce GT 750M), the render times are obtained at 626.72ms. When the program is executed with a corei3 multiprocessor and NVIDIA's GPU (GeForce 920M), the render times are obtained at 638.13ms. It can be observed that GPU (GeForce GT 750M) time is faster than GPU (GeForce 920M) time using constant memory for the number of object 2000.

Table 6 shows the timed data using Global memory for the number of object 2000. When the program is executed with a corei7 multiprocessor and NVIDIA's GPU (GeForce GT 750M), the render times are obtained at 801.53ms. When the program is executed with a corei3 multiprocessor and NVIDIA's GPU (GeForce 920M), the render times are obtained at 752.79ms. It can be observed that the average GeForce 920M render time is faster than the average GeForce GT 750M render time using Global memory for the number of object 2000.

Table 6: List of GPUs and render times using Global memory for the number of object 2000

| **Graphics Card** | **GT 750M(ms)** | **920M(ms)** |
|---|---|---|
| Fram 1 | 794.3 | 752.8 |
| Fram 2 | 797.6 | 753.7 |
| Fram 3 | 793.4 | 752.3 |
| Fram 4 | 806.2 | 752.5 |
| Fram 5 | 804 | 752.1 |
| Fram 6 | 809.1 | 753 |
| Fram 7 | 795.1 | 753.7 |
| Fram 8 | 809.5 | 753 |
| Fram 9 | 804.7 | 752.2 |
| Fram 10 | 801.4 | 752.6 |
| Total | 8015.3 | 7527.9 |
| **Average** | **801.53** | **752.79** |

Table 7: GPU render times for scenes with increasing object counts

| Memory | 900 Objects (ms) | 1500 Objects (ms) | 2000 Objects (ms) |
|---|---|---|---|
| 920M(constant) | 287.47 | 481.19 | 638.13 |
| 750M(constant) | 298.25 | 479.21 | 626.72 |
| 920M( global ) | 341.26 | 564.33 | 752.79 |
| 750M (global) | 374.46 | 607.17 | 801.53 |

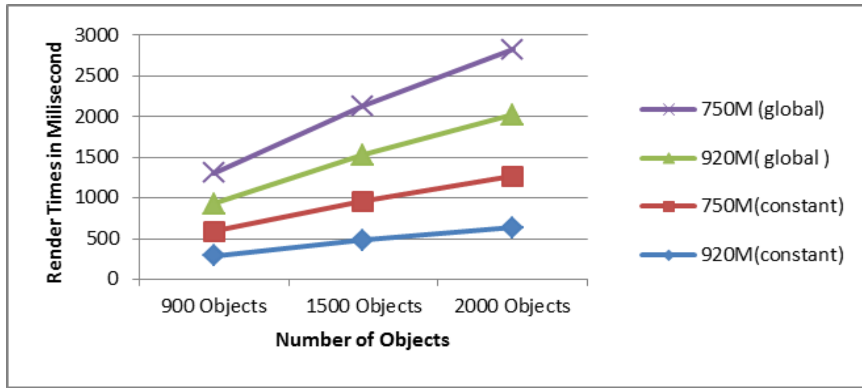


Figure 5: NVIDIA GeForce 920M render times versus GT 750M render times for scenes with increasing object counts

The render times of the GeForce 920M and GT 750M out-performing all ray renders of similar scenes with the 900, 1500, 2000 object counts is illustrated in Figure 5. While the GPU outperforms ray in all three scenes (900 –1500 – 2000), the degree by which the GPU outperforms decreases as the number of objects increases. From Table 7 and Figure.5, it can be observed that the NVIDIA GeForce 920M (constant memory) render time is faster than the GeForce 920M (global memory) render time and the NVIDIA GeForce GT 750M (constant memory) render time faster than the GeForce GT 750M (global memory) render time. So the experiments show that the constant memory ray tracer is faster than the version that uses global memory.

Table 8: Output data of Number of objects and Speed

| No. of objects | 900 | 1500 | 2000 |
|---|---|---|---|
| speed | 1.097287 | 1.075913 | 1.064746 |

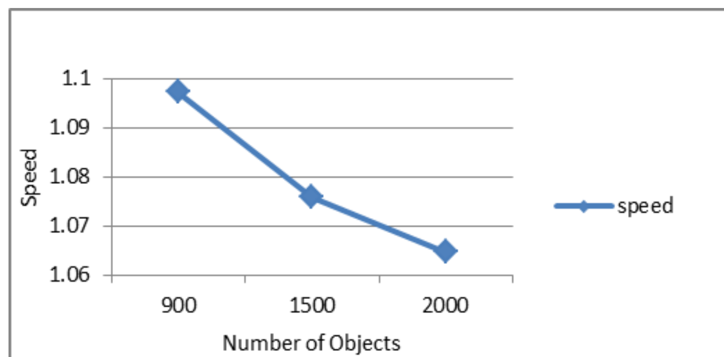


Figure 6: Chart of declining GPU renderer performance with increasing object Counts

The comparison of GPU (NVIDIA GeForce 920M) speed to GPU (NVIDIA GeForce GT 750M) is presented in Figures 6 and Table 8. Upon inspection of figure 6, the GPU (920M) can render one frame almost 1.1 times before GPU (750) can render one frame in 900 objects. As the number of objects in the scene increases, the comparative GPU rendering performance decreases. Moreover the results show that the speed of the NVIDIA GeForce 920M is faster than the speed of the NVIDIA GeForce GT 750M.

## Conclusion

This research describes the utility of graphics cards. When two versions of the ray tracer are run, the time it takes to complete the GPU work is compared. Performance is improved by using constant memory in this research. The experiments on a GeForce GT 750M

show the constant memory ray tracer performing faster than the version that uses global memory. Moreover, the experiments on a GeForce 920M show the constant memory ray tracer performing up faster than the version that uses global memory. This research has investigated the usages of CUDA events to request the runtime to record time stamps at specific points during GPU execution. The CPU is synchronized with the GPU on one of these events and then the time elapsed between two events are computed. This research focuses on applying the calculation efficiency of the GPU to the rendering processes of ray tracing. With the use of a GPU, ray tracing can be performed in less time.

## Acknowledgements

I would like to express my sincere grateful very much to Professor Dr Soe Mya Mya Aye, Head of Department of Computer Studies, University of Yangon, for her kind permission to carry out this research. I am greatly indebted to Dr Pho Kaung (Retired Rector, and the University of Yangon) for his excellent help and creative ideas that have assisted me in broadening my research skills and his continuous guidance, devotion and perpetual encouragement.

## References

Britton, Andrew D.,(2010), Full CUDA implementation of GPGPU recursive ray-tracing.PhD diss., Purdue University.

Edelman,A.,(2004),Applied Parallel Computing, Massachusetts Institute of Technology press.

Grama,A., Gupta,A., Karypis,G. and Kumar,V., (2003), Introduction to Parallel Computing, 2nd Edition, Pearson Education.

Sasikumar,M., Shikhare,D. and Prakash,P.,R.,(2000), Introduction to Parallel Processing, Prentice-Hall,New Delhi, India, ISBN-81-203-1619-3.

T. A. Pitkin, (2013), GPU ray tracing with CUDA, Eastern Washington University.