An Implementation of Ray Tracing in CUDA

CSE 260 Project Report

Liang Chen Hirakendu Das Shengjun Pan

December 4, 2009

Abstract

In computer graphics, *ray tracing* is a popular technique for rendering images with computers. In this project we implemented a serial version of ray tracing in C, and three parallelized versions in CUDA. We conducted experiments to demonstrate speedup with CUDA, as well as the importance of balancing workload among threads.

1 Introduction

In computer graphics, *ray tracing* is a popular technique for rendering images with computers. It is capable of simulating a variety of light effects, e.g., reflection and refraction, generating high degree of photorealism. In consequence, its computational cost is high. In graphics literature, many techniques were proposed to accelerate the computation, including specific data structures and more efficient algorithms. In particular, we are interested in exploring the parallelism of ray tracing in this report. Current trend of processor design is turning toward the multi-core processors. Dual core or quad core already becomes popular in personal computers. Mainstream GPUs nowadays even includes hundreds of cores. How to fully utilize their highly parallel computation capabilities to improve the efficiency of ray tracing becomes an important problem. see [3] for more general information.

1.1 Algorithm Overview

Given a scene of 3D objects and light sources, the goal is to generate an image from the viewpoint of a camera or eye. A 2D image plane consisting of pixels is placed between the viewpoint and the scene. The color of a pixel is the projection of the scene onto the image plane towards the viewpoint.

Ray tracing generates the color of a pixel by sending the light ray through the pixel "backwards" away from the viewpoint. When a "backward" ray hits an object, it's recursively traced back depending on the material of the surface. In complex models more factors are considered, such as scattering, chromatic aberration, etc. For simplicity, we consider only reflections and refractions. A reflected ray for reflective surface and a refracted ray for translucent surface will be further traced.

Figure 1 demonstrates the tracing paths for a single pixel. The blue ball and the red ball are both reflective and refractive. There are three back-tracing paths:

(1) (reflection on red) → (refraction on blue) → (refraction on blue),

(2) (reflection on red) → (reflection on blue), and

(3) (refraction on red) → (refraction on red).

Since the last two paths are traced back to light sources, they are the only paths contributing to the final color of the pixel.

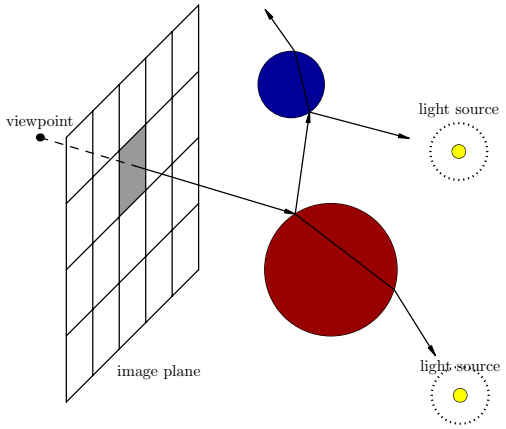


Figure 1: Ray tracing

Ray tracing has lots of advantages over the earlier rendering method. For example, the "ray casting"[1] shoots rays from the viewpoint and finds the closest object that intersects the ray. Ray casting does not trace the rays further. The advantage of ray tracing is that it traces the lights bouncing among objects, which allows a much more realistic simulation of lighting over other rendering methods. A natural capability of ray tracing is to simulate reflections, refractions, and shadows, which are difficult using other algorithms. The computation can be naturally paralellized due to the independency among pixels.

The pseudo code for our ray tracing algorithm is shown in Algorithm 1. The final combined color

Algorithm 1 `Ray Tracing`

```
1. for each pixel pixel in the image plane do
2.    Initialize:
      currentPoint ← the viewpoint
      ray ← the direction from the viewpoint to pixel
3.    return the color of pixel as COLOR(currentPoint, ray), which is calculated recursively as follows:
4.    COLOR(currentPoint, ray)
        1. if currentPoint is blocked by objects from all light sources then
        2.   return the ambient color
        3. else
        4.   Find the point hitPoint where ray first hits an object in the scene.
        5.   if hitPoint does not exists then
        6.     return the background color
        7.   else
        8.     Calculate the reflected ray reflectedRay
        9.     Calculate the refracted ray refractedRay
       10.     Set reflectedColor ← COLOR(hitPoint, reflectedRay)
       11.     Set refractedColor ← COLOR(hitPoint, refractedRay)
       12.     return combined color Combined(localColor, reflectedColor, refractedColor)
       13.   end if
       14.   end if
5. end for
```

`Combined` is a linear combination of the following three parts: the color of the object itself (`localColor`), the color from reflection (`reflectedColor`), and the color from refraction (`refractedColor`). The local color is determined by the *Blinn-Phong shading model* [2].

2 Parallelization

The parallelization of ray tracing comes from the fact that each pixel has no interaction with the other pixels and its color computation is totally independent. The back tracing of lights can be done in parallel for all the pixels. CUDA is potentially suitable for this job: GPUs have hundreds of cores which can trace the lights of the pixels simultaneously. Therefore, a naive approach to parallelize the problem in GPU is creating a thread for every pixel to render the color in parallel.

3 Load Balancing

Ideally, if every pixel requires the same computation workload and GPU has N cores, the speedup of the parallelized ray tracing would be N, compared with serial implementation in a single core processor (assume their clock rate is around the same). However, this assumption is not true: some pixels require more computation than the others. Imagine that the light from a pixel has no intersection with objects, the color of the pixel is just the background color; the computation workload of this pixel is very light. On the other hand, if the light intersects with an object, then the color of the pixel should be the color of the object, which may include the color of other objects that have reflections on it. The computation workload for such light is heavier.

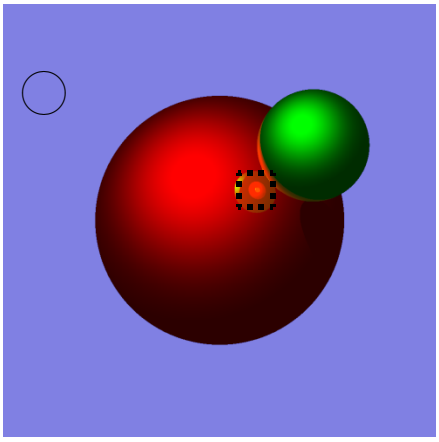


Figure 2: Comparison of workload for different pixels

Figure 2 demonstrates the variety of workloads among pixels. Workload for pixels in the circled region are minimum in the sense that the tracing will return after one step, as the light only "intersects" with the background. On the other hand, workload for pixels in the boxed region is high. The red ball and the blue ball reflect onto each other multiple times. When we trace the light starting from this region, the number of steps is much more than the circled region.

Unbalanced workload hurts the performance. The total execution time of the algorithm is bound by the pixels whose computing time is the longest. Thus, the benefit gained through parallelism is greatly limited. More importantly, adding more cores doesn't solve the problem as the "busiest" pixel are still computed by a single core. This means that future processors with more cores cannot not improve the execution time, but only reduces the parallel efficiency, which is a very bad news for a parallel implementation.

We study the workload balancing problem of ray tracing. In general, two types of strategies can be applied to solve the problem: static scheduling and dynamic scheduling. The static balancing estimates the workload beforehand, and assigns the workload to computing units accordingly. The dynamic balancing adjusts the workload for different computing units on the fly. In the following, we investigate different scheduling strategies to achieve high throughput and parallel efficiency.

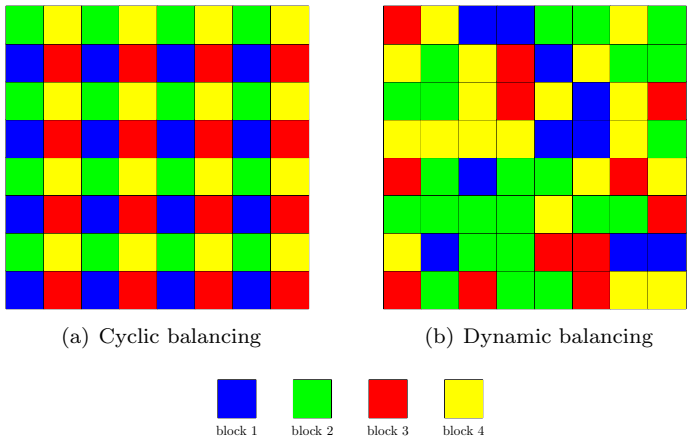
3.1 Adaptive Partitioning

Adaptive partitioning is widely used in parallel computation to partition the job so that each partition has around the same workload. For the problem of ray tracing, ideally we would like to estimate, in reasonable time, the workload for the pixels so that we may partition the pixels into regions with various sizes that have the same workload. However, this is prohibitively difficult. The estimation can't be done without intensive computation, due to the complexity of light traveling among 3D objects.

3.2 Cyclic Balancing

Cyclic balancing iterates through computing units and assigns atomic workloads to them one by one. For ray tracing, the key observation is that the color in an image most likely changes gradually from pixel to pixel. For example, in Figure 2, the color changes gradually except for the boundaries of balls and shadows. Although workloads vary significantly overall, the difference between workloads of adjacent pixels tends to be the very small. Therefore, the workloads assigned to every computing unit in each iteration are around the same. The total workloads of every unit (the sum of all the iterations) are also similar.

Taking advantage of this observation, we partition the pixels into chunks, and assign the workload cyclically to a computing unit, which is a *computing block* of threads in our CUDA implementation. Figure 3(a) is a visual demonstration of our cyclic method.



(a) Cyclic balancing (b) Dynamic balancing

Figure 3: Cyclic balancing and Dynamic balancing
(each square represents a block of threads)

3.3 Dynamic balancing

One problem with static balancing is that most of the time the cost model is very complex and the workload estimation is not accurate. Dynamic balancing aims to schedule workloads dynamically so that real workload is evenly distributed.

In our scenario, we partition the pixels into *chunks*, each of which has the same as the number of threads in a computing block. The chunks are dynamically assigned to the thread blocks as the computation goes. The basic idea is to keep all the computing blocks busy all the time. For example, when a computing block is assigned with a chunk of pixels with the background color, the computation for this chunk will be done immediately and the computing block will continue to fetch another chunk to compute. Figure 3(b) is a visual demonstration of the dynamic method.

3.4 Fully Dynamic Balancing

Until now, the load balancing is at the thread block level, i.e., one pixel is fully computed by a single thread. In some scenarios, dynamic scheduling within a single thread block is also desirable.

In previous dynamic implementation, the number of the threads in a block equals to the number of pixels in the chunk. One thread computes exactly one pixel. However, even within a chunk the workloads may vary for different pixels. In some settings where the thread block size is smaller than the chunk size, some threads may be able to compute more pixels than others.

Furthermore, the lengths of the tracing paths may vary dramatically for pixels within a chunk. Break tracing paths into multiple pieces and assigning them dynamically to the threads improve the parallel efficiency, as shown in Figure 4.

Unfortunately, the above scenarios cannot be implemented in CUDA engine. The key problem is the synchronization between threads within a thread block. In CUDA, instructions are scheduling in a 32-thread warp. 32 threads execute the same instruction at the same time. However, dynamic scheduling within a thread block requires the threads in a block acquire the lock at the same time in order to obtain a job to compute. This behavior inevitably leads to deadlock status.

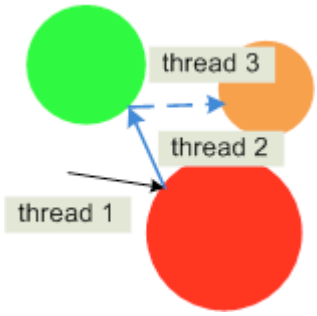


Figure 4: Attempt to break a tracing path

4 Implementation

We implemented the following versions of Algorithm 1 in C and CUDA.

Serial We implemented the serial code in pure C from scratch. The input scene is maintained as a `struct`, including objects (3D spheres and a supporting table), light sources, a view point, and a view window. The program generates a 2D image of the objects observed through the view window. The colors are computed pixel by pixel serially.

Naive This is the first attempt to parallelize the computation. The idea is simple: since computing the colors is independent among pixels, one CUDA thread is created from each pixel. The implementation is straightforward: create CUDA grid `dimGrid` and block `dimBlock` so that the total number of threads `(dimGrid.x*dimGrid.y) * (dimBlock.x*dimBlock.y)` is exactly the number of pixels in the image, and then invoke the kernel function with these specifications. All threads start simultaneously. When there are more threads than available thread processors, the decision on scheduling will be left to the compiler and the GPU.

Shared Memory For the sake of simplification, for all the parallelized methods, including the naive implementation, and the following cyclic and dynamic implementations, we load objects in the scene into the shared memory to speed up the search process. The object list is frequently searched to determine which object a ray is reflected or refracted onto, and which object may be blocking the current ray from

the light sources. In our ray tracing model, the objects (spheres or planes) have simple parameters, which allow us to load all objects into the shared memory. Since we are focusing on load balancing, we currently don't consider the scalability over scene size.

Cyclic As previously discussed, we observe that the color in an image most likely changes gradually from pixel to pixel. In the cyclic implementation, we partition the pixels into chunks, and assign pixels from alternating chunks to a block of threads.

Note that the cyclic implementation is a method of static load balancing. Each thread computes the colors of a predetermined group of pixels. The pixels are chosen in an interleaving fashion with the hope to spread the workout among pixels.

Dynamic The dynamic implementation is similar to the cyclic implementation, in the sense that the pixels are also divided into chunks. Unlike the cyclic method, which computes for pixels in predetermined chunks, the dynamic approach allows the blocks of threads to fetch chunks on the fly.

A natural problem arises as for scheduling the assignment of chunks. To avoid race conditions as well as overlapped computation, we keep a counter in the global memory. The counter is the index of the next chunk to be computed, and it also servers as the purpose of a lock. When a block finishes its computations for all pixels in the current chunk, it reads and increments the counter in one non-interrupted transaction using `atomicAdd()`.

The first thread in each block is responsible for reading and incrementing the counter. The value of the counter, which is in the global memory, is read into a variable in the shared memory. It should be necessary to use `__threadfence_block()` right after the reading. This ensures that all threads within the same block becomes aware of the newly obtained chunk.

In addition, `__synchthreads()` is used to guarantee that the threads with the same block are synchronized.

5 Experiments

5.1 Settings

The input for our experiments is a scene of 3D objects represented as `struct`, including 60 balls with random colors, a support table, and two light sources. These specific parameters are chosen so that it's sufficient to demonstrate the effect of load balancing. The program outputs an image of size 2000×2000. Figure 5 shows a 2D image generated by our code. The image is scaled so that it can properly fit in the document.

We conducted several experiments to analyze the performance of our implementation. Table 1 shows some of the results. The corresponding speedups are shown in Figure 6. All results are obtained from the server Lincoln.

For the parallel implementations, the geometries, i.e., the block dimensions, number of threads etc., are chosen so that both relatively high throughput parallel efficiency can be achieved at the same time. Notice that unlike previous homework where geometries may influence the memory activities, in our experiments, it only influences the workload balancing: how many computing units, and how many work pieces. The reason is that the input data is greatly simplified, as we only consider a set of spheres and one plane. The input data can easily fit into the shard memory.

In the experiments, three metrics are used to evaluate the performance: execution time, Gflops and parallel efficiency. Gflops are measures through our own program, i.e., we manually count the float operations throughout the execution of every thread. Parallel efficiency is computed as

$$\text{parallel efficiency} = \frac{\text{average number of flops per thread}}{\text{maximum number of flops of a thread}}$$

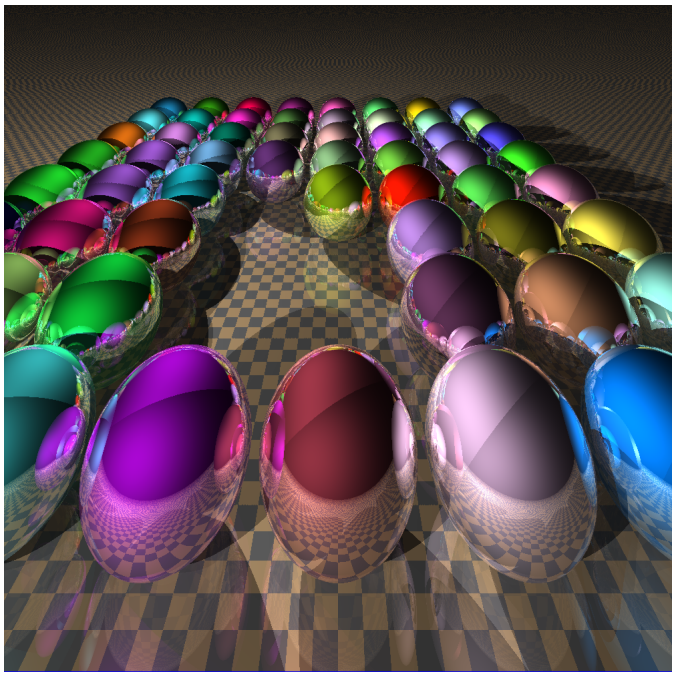


Figure 5: 2D image (scaled) generated by ray tracing

Implementation	Geometry		throughput (seconds)	Execution time (Gflops)	Prallel efficiency $\frac{\texttt{average throughput}}{\texttt{max throughput}}$
	block size	# threads			
serial	-	-	90.56	-	-
Naive	16×10	2000×2000	1.75	10.91	11.20%
Cyclic	8×8	200×200	0.43	44.10	77.44%
Dynamic	8×8	80×80	0.44	43.37	85.36%

Table 1: Experiment results

5.2 Results

Serial A typical run of the serial C code on Lincoln takes 90.56 seconds. This will be the baseline in evaluating the performance of the parallel implementations.

Naive Each block of threads has dimensions 16×10. The total number of threads is 2000×2000 (hence the number of blocks is $\frac{2000}{16} \times \frac{2000}{10} = 125 \times 200$). Each thread is assigned with exactly one pixel.

Compared to the serial code, the naive implementation already achieves a speedup of 52 times.

The number of flops per thread is also estimated. Figure 7(a) plots the distribution of workload (in Gflops). As we can see, the workload is unevenly distributed. The next two experiments show how the performance can be improved by balancing the workload.

Cyclic Each block of threads has dimensions 8×8. There are 25×25 blocks, namely the total number of threads is 200×200. The pixels are divided into 10×10 grids, and a block is assigned with 80×80 pixels (8×8 from each grid).

The cyclic implementation achieves a speedup of 211 compared to the serial code. More importantly, compared to the naive implementation, we gain a speedup of 4 times.

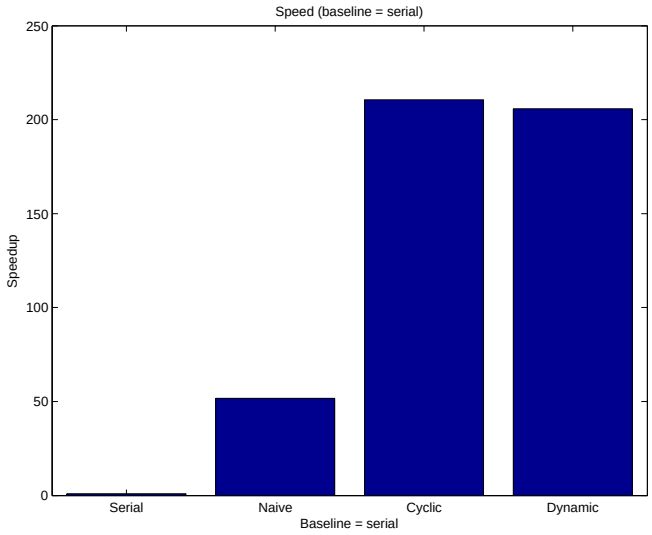


Figure 6: Comparison of speedups

The estimated number of flops per thread is plotted in Figure 7(b). Clearly the workload is more evenly distributed compared to the naive implementation. This is consistent with our gain in the speedup. All show that we have effectively achieved our goal of balancing the workload among pixels.

Dynamic Each block of threads has dimensions 8×8. Each block will handle 8×8 pixels at a time, one thread per pixel. The total number of blocks is 10×10, namely there are 80×80 available threads. The pixels are divided into chunks of size 8×8, which are dynamically assigned to the next available block.

Compared to the cyclic implementation, the dynamic implementation is able to achieve higher throughput, while gaining around the same speedup. The slight drop in speedup is due to overhead in scheduling blocks.

Figure 7(c) show the spread out of workload. Although a small fraction of threads get assigned small relatively amount of workload, the variance is significantly reduced. Visually, The plot for dynamic implementation is less "peaky" than the plot for cyclic implementation.

For a more detailed performance analysis and various profiler outputs, please refer to Appendix A.

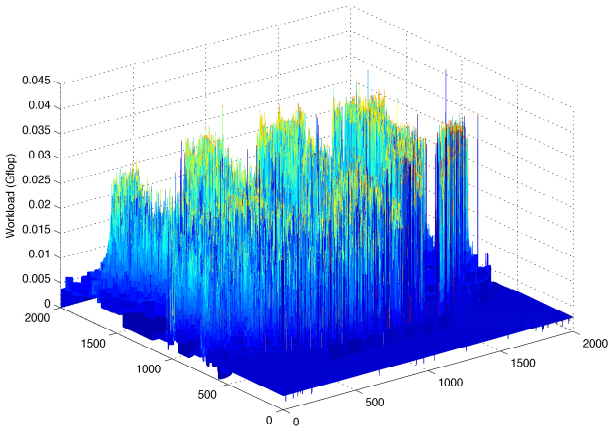
6 Conclusion

In conclusion, besides the serial code, we implemented parallelized implementations of ray tracing in CUDA. We studied the workload balancing problem of ray tracing in CUDA. We exploited and implemented: cyclic and dynamic scheduling. We conducted experiments to demonstrate the importance of load balancing. We not only achieved the speedup as compared to the serial code, but also effectively solved the load balancing problem that exists in the naive implementation. The cyclic and dynamic implementations achieve high speedup, throughput, and parallel efficiency with careful design of geometries on the input.

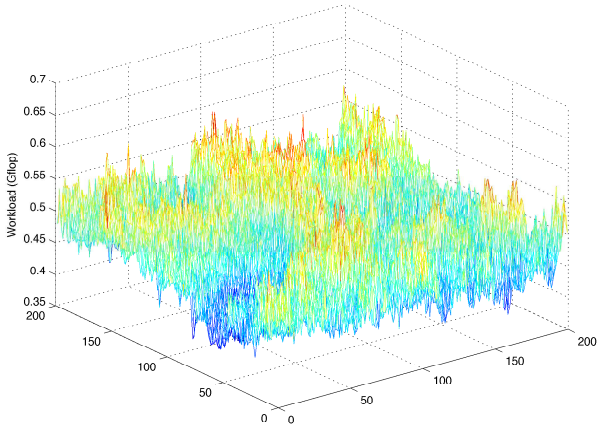
In additional to the promising results, we also made an attempt to fully balance the workload for ray tracing. Further analysis shows that CUDA is incapable of supporting fully dynamic scheduling strategy.

A Additional Performance Analysis

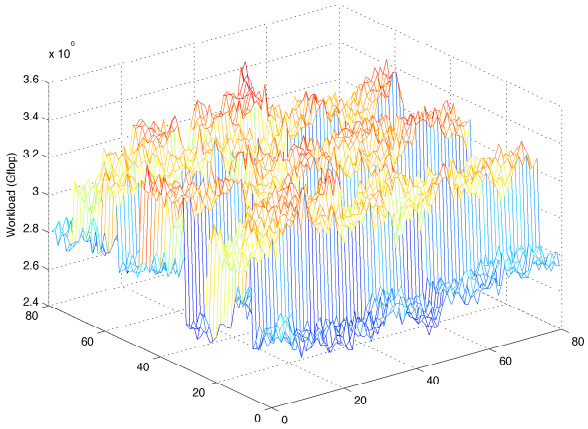
A brief visualisation of various functions performing floating point operations (FLOPs), and their dependency tree with root as the kernel `Render()` is shown in Figure 8. This may help to better understand the profiler



(a) Naive implementation



(b) Cyclic implementation



(c) Dynamic implementation

Figure 7: Workload distribution over threads

outputs. The kernel `Render()` is called each time to determine the color of a image pixel. The basic functions on coordinate triples, described in `Triples.cu` and the number of FLOPs performed by them are shown in Table 2. Likewise, the secondary functions in `Objects.cu` and their FLOPs are shown in Table 3

The actual number of FLOPs performed by the program is calculated by passing a `flop_counter` array (one counter for each thread) to the functions whenever they cannot be calculated before runtime. The individual functions then increment the `flop_counter` depending on the conditional paths taken (and in turn pass the `flop_counter` array to the functions they call if necessary).

Parts of the `gprof` profiler output for the single-threaded CPU implementation of the program are shown in Figure 9. As is evident, most of the time is spent in the function `Intersect_Sphere()` which is called frequently and is computationally intensive (17 or 21 FLOPs). Since `Intersect_Sphere()` in turn calls `Norm2()`, it also shows up high in the profiler statistics. Other functions like `DotProduct()` and `Sub()` which are called frequently and almost by every function show up high in the list too.

The CUDA profiler (`cudaprof`) output for the *cyclic* implementation is shown in Figure 10. We observe that the program is computationally intensive with very little time spent in `cudaMemcpy` and most time spent in the `Render2()` kernel. The occupancy rate is 16.7% and the instruction throughput is about 52.2%.

```
Render()
    Sub()
    UnitVector()
    TraceSingle()
        GetLocalColor()
            GetNormVec()
                GetNormVec_Sphere()
                    Sub()
                    UnitVector()
                GetNormVec_Table()
            Sub()
            UnitVector()
            Intersect()
                Intersect_Sphere()
                    Sub()
                    DotProduct()
                    Norm2()
                Intersect_Table()
            DotProduct()
            Scalar()
        GetReflectionDir()
            GetReflectionDir_Sphere()
                GetNormVec_Sphere()
                    Sub()
                    UnitVector()
                DotProduct()
                Scalar()
                Add()
        GetRefractionDir()
            GetRefractionDir_Sphere()
                GetNormVec_Sphere()
                    Sub()
                    UnitVector()
                Scalar()
                Add()
        GetInterObj()
            Intersect()
                Intersect_Sphere()
                    Sub()
                    DotProduct()
                    Norm2()
            Scalar()
            Add()
        Scalar()
        Add()
```

Figure 8: A tree representation of the various functions called by the main kernel `Render()` which draws each pixel of the image. The functions are broken down upto the basic function calls shown in Table 2. Only the functions that perform float-point operations are shown. The functions may perform additional flops (like floating point addition, multiplication and special operations like `log()` and `sqrt()`) other than that due to the functions mentioned under it. Also, the functions mentioned under a function may be called multiple times or may not be called at all dependending on loops, conditionals and number of objects.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 30.67      5.90     5.90 396193264     0.01     0.04  Intersect_Sphere
 17.60      9.29     3.39 406387207     0.01     0.01  DotProduct
 17.18     12.60     3.31 409424570     0.01     0.01  Sub
 10.21     14.56     1.97 396193264     0.00     0.00  Norm2
  6.81     15.88     1.31 401594723     0.00     0.04  Intersect
  4.52     16.75     0.87  2361734     0.37     3.95  GetLocalColor
  3.59     17.44     0.69  3361734     0.21     2.60  GetInterObj
  2.24     17.87     0.43 13231306     0.03     0.03  UnitVector
  1.56     18.17     0.30                             GetRefractionDir
  1.51     18.46     0.29                             GetRefractionDir_Sphere
  1.09     18.67     0.21  8488049     0.02     0.02  Scalar
 (truncated)
-----------------------------------------------
index % time    self  children    called     name
                                                 <spontaneous>
[1]     96.9    0.03   18.64                 main [1]
                0.04   18.60 1000000/1000000     Render [2]
                0.00    0.00      15/15          printLittle [24]
                0.00    0.00       1/1           CreateScene [25]
                0.00    0.00       1/1           PrintSceneOnHost [26]
-----------------------------------------------
                0.04   18.60 1000000/1000000     main [1]
[2]     96.8    0.04   18.60 1000000         Render [2]
                0.16   18.40 3361734/3361734     TraceSingle [3]
                0.03    0.00 1000000/13231306     UnitVector [11]
                0.01    0.00 1000000/409424570     Sub [9]
-----------------------------------------------
                0.16   18.40 3361734/3361734     Render [2]
[3]     96.4    0.16   18.40 3361734         TraceSingle [3]
                0.87    8.46 2361734/2361734     GetLocalColor [6]
                0.69    8.05 3361734/3361734     GetInterObj [7]
                0.02    0.19 2361734/2361734     GetReflectionDir [15]
                0.06    0.00 2361734/6126315     Add [17]
                0.06    0.00 2361734/8488049     Scalar [14]
-----------------------------------------------
                0.65    7.20 199890683/401594723     GetLocalColor [6]
                0.66    7.27 201704040/401594723     GetInterObj [7]
[4]     81.9    1.31   14.47 401594723         Intersect [4]
                5.90    8.47 396193264/396193264     Intersect_Sphere [5]
                0.10    0.00 5401459/5401459     Intersect_Table [19]
-----------------------------------------------
                5.90    8.47 396193264/396193264     Intersect [4]
[5]     74.6    5.90    8.47 396193264         Intersect_Sphere [5]
                3.30    0.00 396193264/406387207     DotProduct [8]
                3.20    0.00 396193264/409424570     Sub [9]
                1.97    0.00 396193264/396193264     Norm2 [10]
-----------------------------------------------
                0.87    8.46 2361734/2361734     TraceSingle [3]
[6]     48.5    0.87    8.46 2361734         GetLocalColor [6]
                0.65    7.20 199890683/401594723     Intersect [4]
                0.31    0.00 9446936/13231306     UnitVector [11]
                0.02    0.06 2361734/2361734     GetNormVec [20]
                0.08    0.00 9446936/409424570     Sub [9]
                0.07    0.00 8801758/406387207     DotProduct [8]
                0.06    0.00 2361734/8488049     Scalar [14]
                0.01    0.00 2361734/2361734     GetColor [21]
-----------------------------------------------
                0.69    8.05 3361734/3361734     TraceSingle [3]
[7]     45.4    0.69    8.05 3361734         GetInterObj [7]
                0.66    7.27 201704040/401594723     Intersect [4]
                0.06    0.00 2372396/6126315     Add [17]
                0.06    0.00 2372396/8488049     Scalar [14]
-----------------------------------------------
                0.01    0.00 1392185/406387207     GetReflectionDir_Sphere [16]
                0.07    0.00 8801758/406387207     GetLocalColor [6]
                3.30    0.00 396193264/406387207     Intersect_Sphere [5]
[8]     17.6    3.39    0.00 406387207         DotProduct [8]
-----------------------------------------------
                0.01    0.00 1000000/409424570     Render [2]
                0.02    0.00 2784370/409424570     GetNormVec_Sphere [18]
                0.08    0.00 9446936/409424570     GetLocalColor [6]
                3.20    0.00 396193264/409424570     Intersect_Sphere [5]
[9]     17.2    3.31    0.00 409424570         Sub [9]
-----------------------------------------------
                1.97    0.00 396193264/396193264     Intersect_Sphere [5]
[10]    10.2    1.97    0.00 396193264         Norm2 [10]
-----------------------------------------------
                0.03    0.00 1000000/13231306     Render [2]
                0.09    0.00 2784370/13231306     GetNormVec_Sphere [18]
                0.31    0.00 9446936/13231306     GetLocalColor [6]
[11]     2.2    0.43    0.00 13231306         UnitVector [11]
-----------------------------------------------
                                                 <spontaneous>
[12]     1.6    0.30    0.00                 GetRefractionDir [12]
-----------------------------------------------
                                                 <spontaneous>
[13]     1.5    0.29    0.00                 GetRefractionDir_Sphere [13]
-----------------------------------------------
                0.03    0.00 1392185/8488049     GetReflectionDir_Sphere [16]
                0.06    0.00 2361734/8488049     GetLocalColor [6]
                0.06    0.00 2361734/8488049     TraceSingle [3]
                0.06    0.00 2372396/8488049     GetInterObj [7]
[14]     1.1    0.21    0.00 8488049         Scalar [14]
-----------------------------------------------
                0.02    0.19 2361734/2361734     TraceSingle [3]
[15]     1.1    0.02    0.19 2361734         GetReflectionDir [15]
                0.03    0.15 1392185/1392185     GetReflectionDir_Sphere [16]
                0.01    0.00  969549/969549      GetReflectionDir_Table [22]
-----------------------------------------------
  (truncated)

Index by function name
  [17] Add                    [15] GetReflectionDir       [26] PrintSceneOnHost
  [25] CreateScene            [16] GetReflectionDir_Sphere [2] Render
   [8] DotProduct             [22] GetReflectionDir_Table [14] Scalar
  [21] GetColor               [12] GetRefractionDir        [9] Sub
   [7] GetInterObj            [13] GetRefractionDir_Sphere [3] TraceSingle
   [6] GetLocalColor           [4] Intersect              [11] UnitVector
  [20] GetNormVec              [5] Intersect_Sphere        [1] main
  [18] GetNormVec_Sphere      [19] Intersect_Table        [24] printLittle
  [23] GetNormVec_Table       [10] Norm2
```

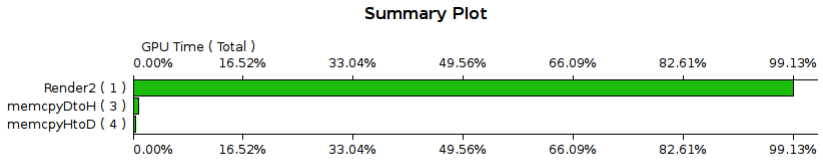
Figure 9: The `gprof` profiler output (only the salient parts) for the CPU implementation.

Function	# of FLOPs
`Add()`	3
`Sub()`	3
`Scalar()`	3
`DotProduct()`	5
`Norm2()`	5
`UnitVector()`	9

Table 2: Basic functions on coordinate triples in `Triples.cu` and their FLOPs.

Function	# of FLOPs
`GetNormVec_Sphere()`	12
`GetReflectionDir_Sphere()`	26
`GetRefractionDir_Sphere()`	34
`Intersect_Sphere()`	17 or 21
`Intersect_Table()`	1

Table 3: Secondary functions in `Objects.cu` and their FLOPs.



(a)

	Method	#Calls	GPU usec	%GPU time	instruction throughput
1	Render2	1	2.73699e+06	99.12	0.522099
2	memcpyHtoD	4	7162.91	0.25	
3	memcpyDtoH	3	16875.2	0.61	

(b)

	Method	Occupancy	static shared memory per block	registers per thread	mem transfer size (bytes)	branch	divergent branch	instructions	warp serialize
1	memcpyHtoD				3196				
2	memcpyHtoD				12006000				
3	memcpyHtoD				320000				
4	memcpyHtoD				320000				
5	Render2	0.167	3244	60		120341576	844630	473349707	884940
6	memcpyDtoH				320000				
7	memcpyDtoH				320000				
8	memcpyDtoH				12006000				

(c)

Figure 10: `cudaprof` output of *cyclic* implementation.

References

[1] A.Arthur, "Some techniques for shading machine renderings of solids," in *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference.* Atlantic City, New Jersey, New York, NY, USA: ACM, 1968, pp. 37–45.

[2] J. F. Blinn, "Models of light reection for computer synthesized pictures," in *SIGGRAPH'77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques.* New York, NY, USA: ACM, 1977, pp. 192–198.

[3] `http://en.wikipedia.org/wiki/Ray_tracing_(graphics)`