GPU Accelerated Ray Tracer

June 10, 2019

Team Alien One — Alic Lien

Student ID: 861207479

University of California, Riverside

Contents

Project Overview

The idea for the GPU Accelerated Ray Tracer is inspired by my previous project in Computer Graphics, and the current advancements in GPU hardware technology. Ray tracing itself is a concept that has been around for many years, and utilized in varying capacities. In recent years however, GPU technology has allowed consumer level ray tracing, specifically in real time, with the introduction of the NVIDIA RTX series graphics cards.

With the project being GPU based, we have access to NVIDIA Tesla graphics cards, which differ from the RTX series, as the Tesla hardware lack Ray Tracing Cores that the RTX series have. With that, I wanted to see the differences in runtime between CPU ray tracing, and GPU accelerated ray tracing (without the use of Ray Tracing Cores), thus this became the basis for my project.

In Computer Graphics, I partially coded a Ray Tracer, mostly filling in the required equations for computations into skeleton code. As the required code for rendering was mostly completed, we did not need to pay too much attention as to how the code for rendering worked, or the runtime of the code.

For this project, I wanted to fully understand how ray tracing rendering works in terms of runtime and code. The process of this project involved simplifying and changing up the old C++ Ray Tracing code, converting it to CUDA, and comparing the runtimes of both programs with different accuracies. The comparison is not in most optimal runtime, but it is ideal for showcasing the differences in hardware and software capabilities of linear C++ CPU code versus parallelized CUDA GPU code.

Hardware

The hardware utilized for my project's C++ ray tracing computations is my own personal laptop computer. For the CUDA ray tracing computations, the GPU is provided by the University of California, Riverside — Department of Engineering.

Device and CPU Hardware Specifications:

MacBook Pro (15-inch, Mid 2015)

2.8 GHz Intel Core i7-4980HQ CPU

MacOS Mojave - Version 10.14.4

16 GB 1600 MHz DDR3

Provided By:
Alic Lien

GPU Hardware Specifications:

NVIDIA Tesla M60 GPU
x 4

NVIDIA-SMI 410.79

Driver Version: 410.79

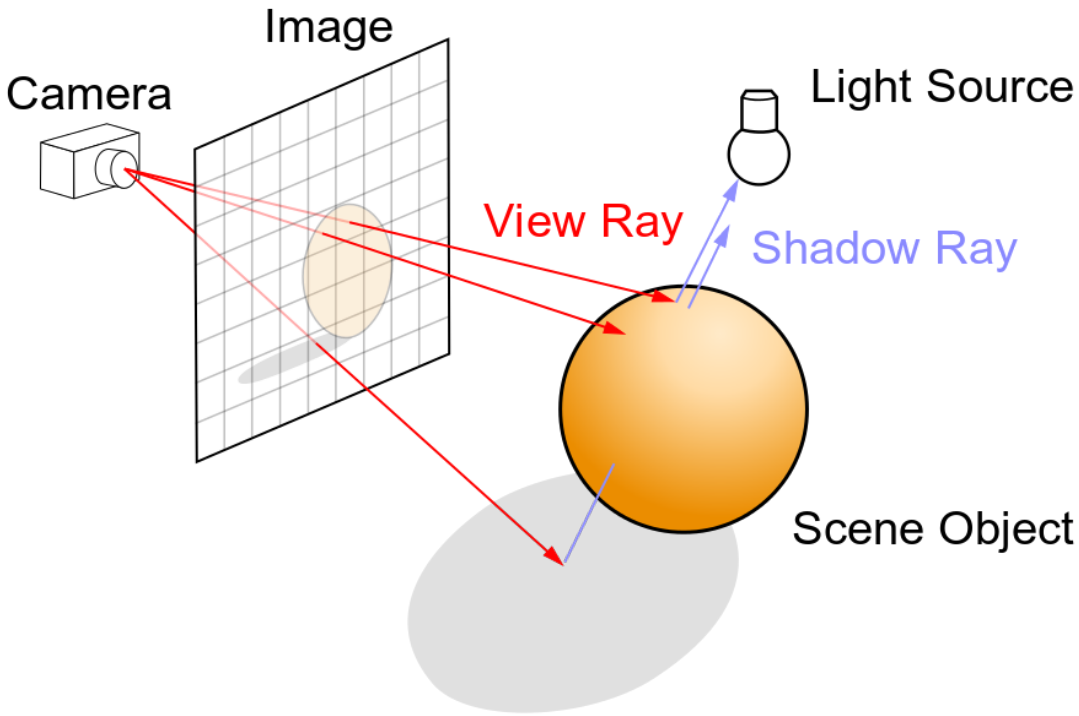
CUDA Version: 10.0

Provided By:
University of California, Riverside
Department of Engineering
SSH Server: Bender

GPU Application

GPU is used to accelerate ray tracing, as it more optimal to compute ray tracing this way. It's in the name, Graphics Processing Unit, and Ray Tracing itself is a computer graphics rendering technique.

To explain a bit about Ray Tracing first, the rendering process involves tracing rays from camera origin, through the lens, to a hit object, and any objects that might reflect the first object hit, pixel by pixel.

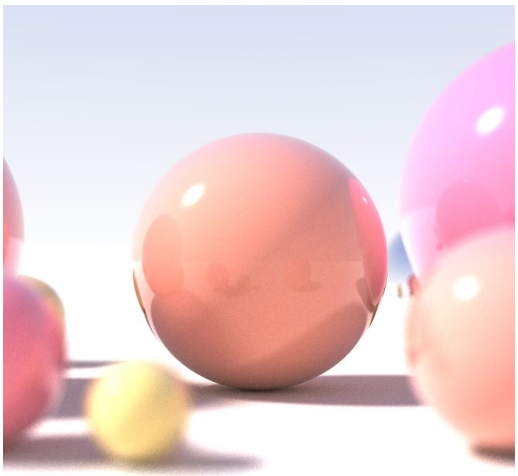


Henrik - Wikimedia Commons

Computation involves calculating how every single pixel will look during the rendering process, as such with larger more detailed images, the rendering will take longer time to complete. Ray tracing renders are much more realistic and detailed compared to ray casting or scanline rendering.

Ray tracing allows for a much higher degree of realism with images, through simulating various optical effects such as various lighting, reflections (ambient, diffuse, and specular), refractions, scattering, dispersion, and shadows.

With the amount of detail possible in one rendered image with ray tracing, it is no surprise that ray tracing requires very high computational costs. As such, utilizing CPU hardware is not ideal for ray tracing; CPUs are reaching their hardware limitations, as predicted by Moore's law. Due to such limitations imposed by CPU hardware, GPUs are ideal and more efficient in computing ray tracing. With programs that are already optimized, it is up to the hardware technology to speed up the process of the programs, as runtime is also hardware dependent.

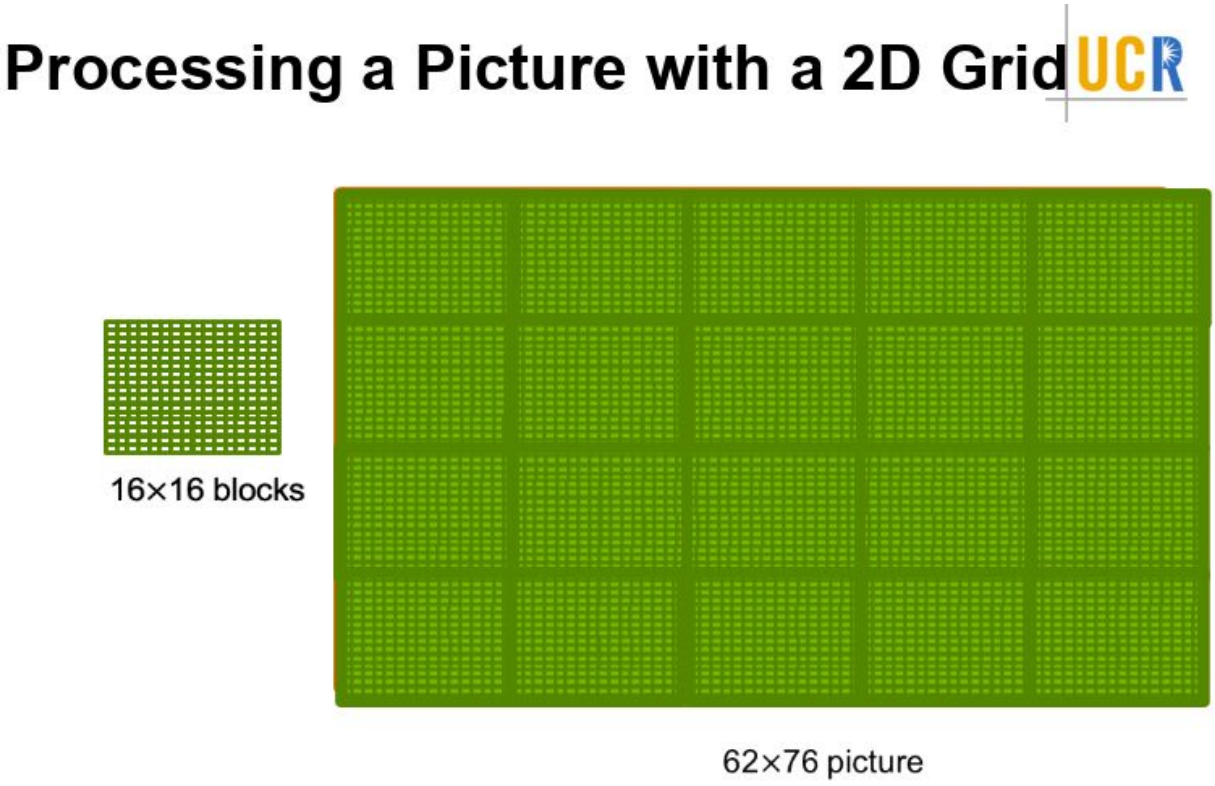


Tim Babb - Wikimedia Commons

With my project, there is a straight forward purpose, to overcome the limitations of CPU hardware, by running ray tracing on the GPU to speed up the rendering process. Therefore, with my project I modified, then converted C++ ray tracing into CUDA.

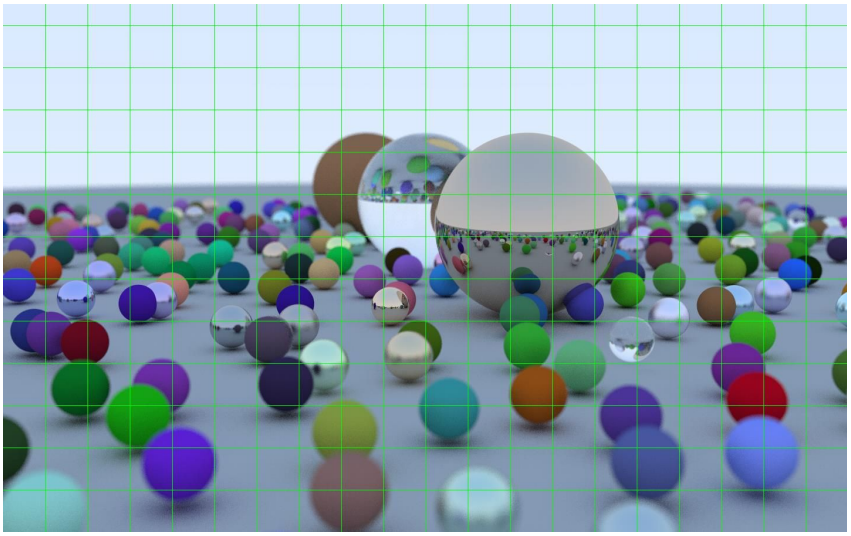
GPU accelerated ray tracing allows my program to render an image in a quarter of the time it takes to run on CPU. This is done by parallelizing the rendering process, specifically the ray tracing rendering calculations on objects. This also allows me to utilize thread blocks, and kernel grid to render my image.

Similar to what we learned in class on parallelism, we utilize the thread blocks in a 16 x 16 thread dimension into a grid to process the image, with almost every thread utilized to calculate one pixel per thread. I found that using block size of 16 x 16 threads yield the most optimal computation time compared to 8 x 8 or 32 x 32.



Lecture Slides - CUDA Parallelism

The lecture slides provide a good example of how thread blocks are utilized in a grid to render an image using ray tracing. With each ray going through each pixel on a grid to calculate how the pixel will look, using threads to render pixels is an optimal way as it provides a much faster runtime compared to CPU.



***Rough Example as to how grid/thread blocks are used to render image.*

Implementation

Originally the section of C++ code that deals with rendering was embedded within main.cpp:

```
for (int i = pixel_y - 1; i >= 0; i--) {
       for (int j = 0; j < pixel_x; j++) {
           vec3 col(0, 0, 0);
           for (int k = 0; k < precision; k++) {
               float u = float(j + drand48()) / float(pixel_x);
               float v = float(i + drand48()) / float(pixel_y);
               ray r = cam.get_ray(u, v);
               col += color(r, world,0);
           }
           col /= float(precision);
           col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );
           int ir = int(255.99*col[0]);
           int ig = int(255.99*col[1]);
           int ib = int(255.99*col[2]);
           outfs << ir << " " << ig << " " << ib << "\n";
       }
}
```

The C++ code shows a very linear rendition of the code, based on the dimensions of the image, and the value of `precision` indicated to determine how accurate the calculations are. With the C++ render, we need a nested loop, which drastically increases the runtime and makes computations very costly.

Rendering a 1440 x 900 image with 100 precision on CPU took around 632 seconds, 10 minutes, just to render a single 15 MB image. As such we can see that using ray tracing to render an image on CPU takes a very long time, and is not ideal for most situations.

For CUDA, this section of the code was moved from main.cpp into its own render function:

```
__global__ void render(vec3 *fb, int pix_x, int pix_y, int precision,
camera **cam, hitable **world, curandState *rand_state) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    if((i >= pix_x) || (j >= pix_y)){
        return;
    }

    int pixel_index = j * pix_x + i;

    curandState local_rand_state = rand_state[pixel_index];

    vec3 col(0,0,0);

    for(int k = 0; k < precision; k++) {
        float u = float(i + curand_uniform(&local_rand_state)) /
                    float(pix_x);
        float v = float(j + curand_uniform(&local_rand_state)) /
                    float(pix_y);
        ray r = (*cam)->get_ray(u, v, &local_rand_state);
        col += color(r, world, &local_rand_state);
    }

    rand_state[pixel_index] = local_rand_state;
    col /= float(precision);
    for(int n = 0; n < 3; n++){
        col[n] = sqrt(col[n]);
    }
    fb[pixel_index] = col;
}
```

As shown by the CUDA implementation, each pixel is represented by the X and Y locations of the thread, and we once again use `precision` to increase the accuracy of the ray tracing. with CUDA, we no longer need a nested loop for rendering pixels.

Documentation

The Ray Tracer is very straight forward in terms of how to run. In both folders, there included is a Makefile. With that, it is very simple to compile and then run the code:

1. Navigate to desired folder (C++_Ray_Tracer or CUDA_Ray_Tracer)
2. > make
3. > ./Ray_Tracer for C++ >./CUDA_Ray_Tracer for CUDA
4. The program will then output an image specified within the main functions of both code.

C++ main.cpp Variables:

```
81. string filename = "CPP_100_Image.ppm";  //Outputted filename
84. int pixel_x = 1440;          //Pixel dimension for X
85. int pixel_y = 900;           //Pixel dimension for Y
86. int precision = 100;    //Increase value for higher precision
```

These variables can be changed for desired output. **`filename`** allows you to change the rendered image's filename (keep the .ppm extension). **`pixel_x`** and **`pixel_y`** modifies the image dimensions. As mentioned before, **`precision`** modifies the quality and accuracy of the ray tracing computations, but as such increases runtime.

CUDA main.cu Variables:

```
34. ofstream outfs ("CUDA_50_Image.ppm");
36. int pixel_x = 1440;      //Pixel dimension for X
37. int pixel_y = 900;       //Pixel dimension for Y
38. int precision = 50;      //Increase value for higher precision
39. const unsigned int BLOCK_SIZE_X = 16;   //Sets num threads on X
40. const unsigned int BLOCK_SIZE_Y = 16;   //Sets num threads on Y
128. ifstream infs("CUDA_50_Image.ppm", ios::binary | ios::ate);
```

CUDA variables are the same, with the exception of setting block sizes (default 16 x 16) with `BLOCK_SIZE_X` and `BLOCK_SIZE_Y` and you must rename **`outfs`** / **`infs`** variable name strings manually, and they need to be the same.

Evaluation of Results

The results were quite straightforward, and as expected. The CPU code usually takes an extremely long time to render the image, while on the GPU with the exact same specifications, the image rendered in a quarter of the CPU time.

Runtime Statistics:

Image Dimensions: 1440 x 900 px
GPU Thread Block Dimensions: 16 x 16

	Runtime (Seconds)	
Precision	CPU C++	GPU CUDA
100	632.25	75.53
50	323.98	42.84
40	251.71	35.41
30	189.56	23.64
20	126.60	22.39
10	64.33	15.85

As we can see, the ray tracer running on GPU renders much faster than running on CPU, 4-8 x faster than CPU. This shows that due to CPU hardware limitations, and a linear implementation, the rendering process runs at very slow speeds. With GPU hardware and CUDA implementations, we parallelized the process and increased rendering speed 8 times at its fastest.

Sample Images

The images were originally outputted as .ppm files, a Netpbm format known as portable pixmap format. The images are represented by 3 numbers on each line of the file, each of the 3 numbers correspond to the RGB values for Red, Green, and Blue for each pixel.

A sample portion of the .ppm file in raw text:

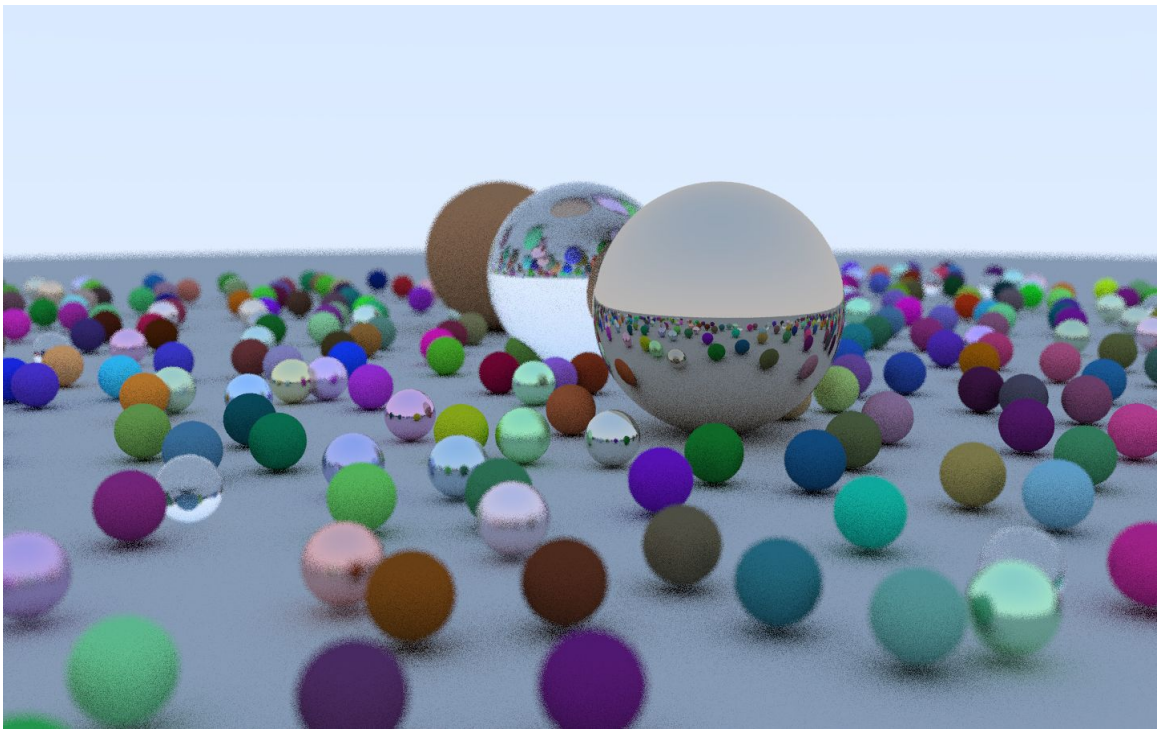
```
P3
1440 900
255
217 233 255
217 233 255
.
.
.
107 130 240
95 119 227
105 129 241
94 117 222
99 123 233
92 115 218
98 121 229
```

`P3` determines the file type, the following line, `1440` and `900` correlate to the images dimensions, `255` correlates to the maximum color value, in this case 255 RGB. The remaining numbers on each line correlate to the RGB values between 0 to 255 for each pixel within the image.

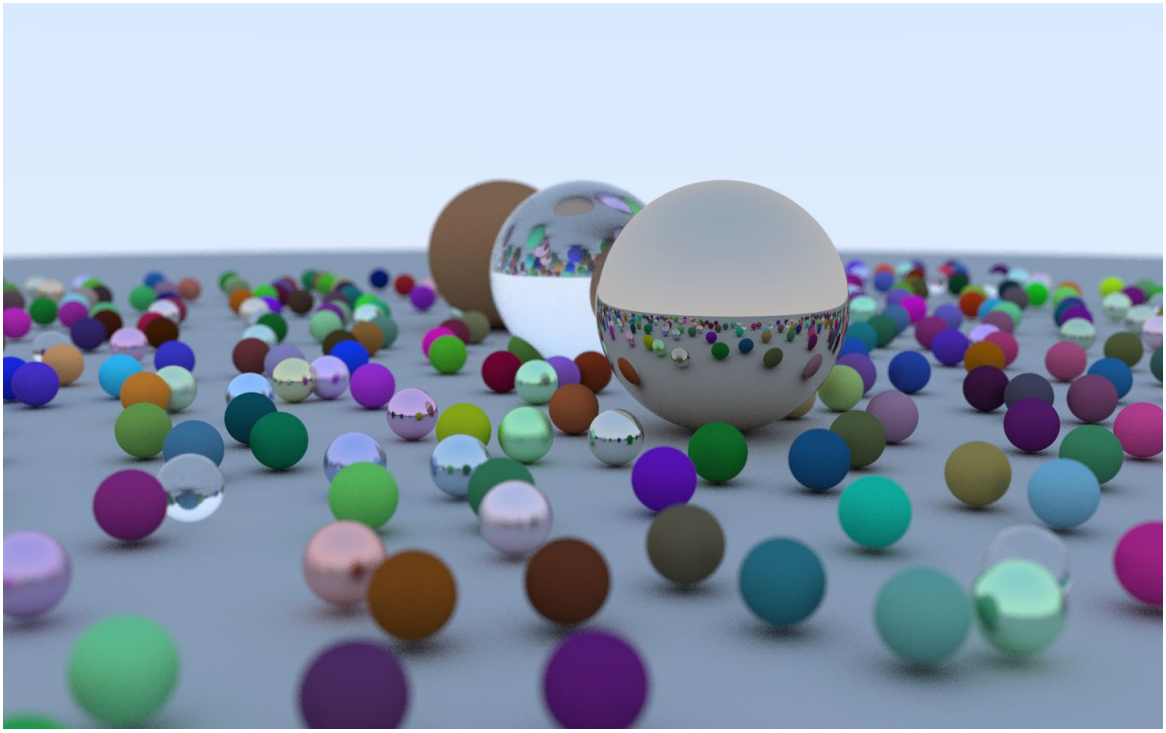
On the next page I have included sample images from my renders, to showcase the varying capacities of CPU and GPU rendering qualities, and how precision affects the outcome image quality.

*Note: These images were converted from .ppm to .png files in order for the rendered images to appear within the document.

CPU C++ Ray Tracing Rendered Images:

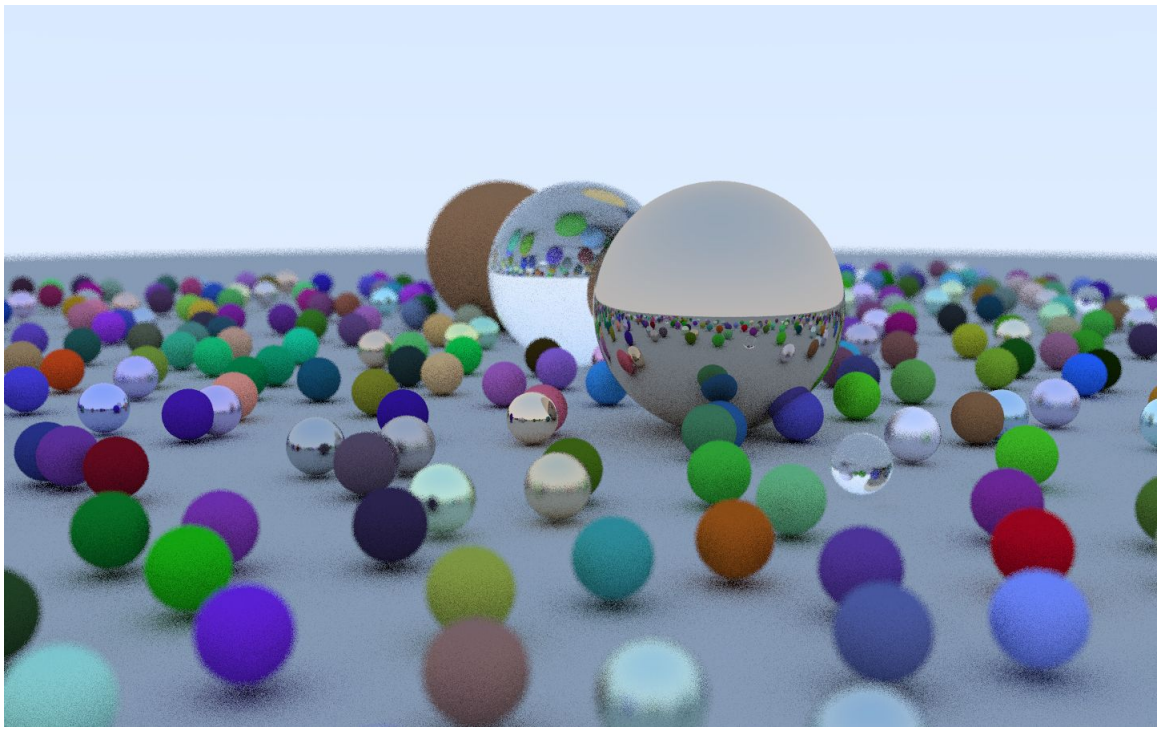


1440 x 900 px @ 10 precision*

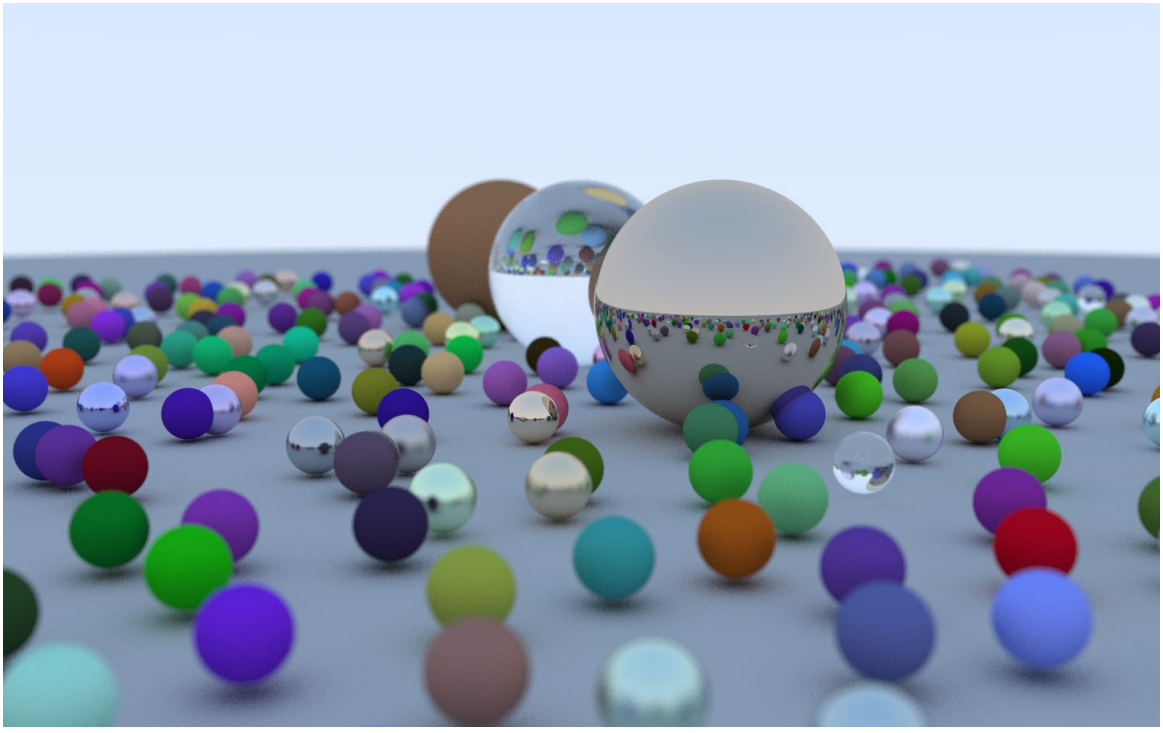


1440 x 900 px @ 100 precision*

GPU CUDA Ray Tracing Rendered Images:



1440 x 900 px @ 10 precision* 16 x 16 block size



1440 x 900 px @ 100 precision* 16 x 16 block size

Development Problems

Now that we have reached the Development Problems portion of the report, is this where I start complaining about every single problem I ran into during coding?

First off, the first problem I ran into is that...I am working alone on this project, therefore I had to do everything on my own (not too bad of a problem, but having projects in 2 other classes did not help with my workload or stress).

Second problem was that my original C++ Ray Tracing code from computer graphics were too complicated with a lot of aspects that I did not need for this project, so I had to streamline the ray tracer, and change up various parts of the C++ program to run ray tracing and render in the way I want it to, and time the program correctly.

After that, converting the simplified C++ code into CUDA was not too much of a problem, my only difficulty was that I had to figure out which section of the program to parallelize correctly, and formatting the functions correctly between `__host__`, `__device__`, or `__global__`. Converting the rest of the code into CUDA was not as bad. I also moved most of the functions within main.cpp to device.cu to create a more organized CUDA code format.

The major problem I ran into was that, originally I tried to configure my desktop computer to run CUDA code as I have a NVIDIA GTX 1070ti capable of running said code. To natively run CUDA with my GPU, I had to attempt to install and dual boot Ubuntu on my computer...However while trying to install Ubuntu to my desktop computer, I accidentally deleted my entire Windows partition, and all my files stored on my computer. This was a great setback, as I had some changes that I did not push to Git at the time.

Another problem was that, while running C++ code with different variable values, I ran into an infinite recursion problem, that I could not find. I spent around 6 hours trying to fix the code before I realized I had working code stored on Git, and that the code I was trying to fix was an old broken backup, therefore wasting 6 hours of my life that I could have spent working on this report.

Overall this project was fun but stressful to do due to many...inconveniences.

Project Task Breakdown

This is my favorite part of the report.

Task	Breakdown
Everything	Alic Lien - 100%

Me working on the project alone without the boys

With that, I conclude my final report for my GPU Accelerated Ray Tracer. Thank you for taking your time to read the report.