GPU Accelerated Computing
Assignment 03 Report

Syed Muhammad Ali — Saad Abdul Hakim

April 6, 2024

1 Introduction

Path and RayTracers have a lot of room for parallelization since they compute color values for each pixel independently. In this assignment we have implemented small pathtracer from Kevin Pearson using CUDA.

2 Approach

2.1 Identiying points where parallelization can be done

The first task involved indentifying the parts of code where parallelization can be done. Since we need to assign a value to each pixel of the image, we can parallelize the loop which iterates over all the pixels serially on CPU (or minutely in parallel using CPU multithreading) to gain siginifcant speedup.

2.2 The Approach

We started with the 99 liner cpp implementation of smallpt from the provided reference and then started converting function and classes into device functions. Then we made a cuda kernel for converting the serial assignment of pixel values to parallel. Finally, this kernel is called inside the main function.

2.3 Code parts

2.3.1 Converting Vec to float3

Since CUDA supports the float3 data type, we can convert all the instance of Vec class in smallpt to float3 instances. Additionally we use vecutils.h, header which provides device functions for different operations such as dot, cross product using float3 object.

2.3.2 Conversion of struct methods to device functions

Now we can convert the structs such as Ray and Sphere so that they can be initialized on the GPU

```
struct Ray
{
    float3 o, d;
    __device__ Ray(float3 o_, float3 d_) : o(o_), d(d_) {}
};
```

Listing 1: Converting structs and its methods into device class/functions

Also, we convert other miscellaneous functions such as clamp and toInt into host,device functions

```
inline __host__ __device__ double
clamp(double x)
{
    return x < 0 ? 0 : x > 1 ? 1
                             : x;
```

```
}

inline __host__ __device__ int toInt(double x) { return int(pow(clamp(x), 1 / GAMMA) *
    255 + .5); }
```

Listing 2: Converting structs and its methods into device class/functions

2.4 Random Initialization

We use curand for random uniform initilzation instead of erand48() or std uniform generator since we need to initialize random values on device.

2.4.1 The Radiance Function

The main part of the ray tracer is the radiance function which actually is responsible for tracing the ray. We first tried to make a recursive device function just like in smallpt. However, we found that due to nondeterministic stack intializations, it performs very poorly, and in some cases randomly crashes. Therefore, we converted the recursive radiance function to a loop-based radiance radiance function. This significantly improved the performance of our code. However, still, we had to recursively call the randiance function if the object material type is transparent since two recursive calls were being made in the smallpt radiance function. This again reduces performance by a bit, but is necessary for getting the exact same output. If we remove this recursive call, then the perfromance improves, but in scenes with transparent object, they slightly differ from smallpt rendered scene.

```
__device__ float3 radiance(Ray &r, curandState state, int d = 0)
{ // returns ray color

    float3 pixelcolor = make_float3(0.0f, 0.0f, 0.0f);
    float3 mask = make_float3(1.0f, 1.0f, 1.0f);

    for (int depth = d; depth < 10; depth++)
    {

        double t;   // distance to closest intersection
        int id = 0; // index of closest intersected sphere

        // test ray for intersection with scene
        if (!intersect(r, t, id))
        {
            return pixelcolor + (BACKGROUND * mask); // for scene 3
        }

        const Sphere &obj = spheres[id];
        float3 x = r.o + r.d * t;
        float3 n = normalize(x - obj.p);
        float3 nl = dot(n, r.d) < 0 ? n : n * -1;
        float3 f = obj.c;
        double p = f.x > f.y && f.x > f.z ? f.x : f.y > f.z ? f.y
                                                           : f.z;

        // Russain Roulette. Can be turned on but it reduces code perfromance (due to
            branch divergence maybe)
        // if (depth > 5)
        // {
        //     if (curand_uniform(&state) < p)
        //     {
        //         f = f * (1 / p);
        //     }
        //     else
        //     {
        //         pixelcolor += obj.e * mask;
        //         break;
        //     }
        // }

        // create 2 random numbers
        float r1 = 2 * M_PI * curand_uniform(&state);
```

```
43          float r2 = curand_uniform(&state);
44          float r2s = sqrtf(r2);
45
46          if (obj.refl == DIFF)
47          {
48              float3 w = nl;
49              float3 u = normalize(cross((fabs(w.x) > .1 ? make_float3(0, 1, 0) :
                    make_float3(1, 0, 0)), w));
50              float3 v = cross(w, u);
51              float3 d = normalize(u * cos(r1) * r2s + v * sin(r1) * r2s + w * sqrtf(1 -
                     r2));
52              pixelcolor += mask * obj.e;
53              r.o = x + nl * 0.05f;
54              r.d = d;
55          }
56
57          else if (obj.refl == SPEC)
58          {
59              r.o = x + nl * 0.05f;
60              r.d = r.d - n * 2 * dot(n, r.d);
61              pixelcolor += mask * obj.e;
62          }
63          else if (obj.refl == REFR)
64          {
65              Ray reflRay(x + nl * 0.05f, r.d - n * 2 * dot(n, r.d)); // Ideal
                    dielectric REFRACTION
66              bool into = dot(n, nl) > 0;                                 // Ray from
                    outside going in?
67              double nc = 1, nt = 1.5, nnt = into ? nc / nt : nt / nc, ddn = dot(r.d, nl
                    ), cos2t;
68              if ((cos2t = 1 - nnt * nnt * (1 - ddn * ddn)) < 0) // Total internal
                    reflection
69              {
70                  pixelcolor += mask * obj.e;
71                  r.o = reflRay.o;
72                  r.d = reflRay.d;
73              }
74              else
75              {
76                  float3 tdir = normalize((r.d * nnt - n * ((into ? 1 : -1) * (ddn * nnt
                         + sqrt(cos2t)))));
77                  double a = nt - nc, b = nt + nc, R0 = a * a / (b * b), c = 1 - (into ?
                         -ddn : dot(tdir, n));
78                  double Re = R0 + (1 - R0) * c * c * c * c * c, Tr = 1 - Re, P = .25 +
                        .5 * Re, RP = Re / P, TP = Tr / (1 - P);
79                  if (depth > 2)
80                  {
81                      if (curand_uniform(&state) < P)
82                      {
83                          pixelcolor += mask * obj.e * RP;
84                          r.o = reflRay.o;
85                          r.d = reflRay.d;
86                          // mask *= RP;
87                      }
88                      else
89                      {
90                          pixelcolor += mask * obj.e * TP;
91                          r.o = x + nl * 0.05f;
92                          r.d = tdir;
93                          // mask *= TP;
94                      }
95                  }
96                  else
97                  {
98                      pixelcolor += mask * obj.e * Re;
99                      // pixelcolor += radiance(reflRay, state, depth + 1) * Re; // a
                            recursive call. Reduces the performance of code but is
                            important.
100                     // r.o = reflRay.o;
101                     // r.d = reflRay.d;
102                     r.o = x;
```

```
                    r.d = tdir;
                    pixelcolor += mask * obj.e * Tr;
                }
            }
        }

        // multiply with colour of object
        mask *= f;
    }

    return pixelcolor;
}
```

Listing 3: Converting structs and its methods into device class/functions

2.5 pathTracer Kernel

Finally, to take advantage of thread level parallelism, we make a pathtracer kernel to simultaneously calculate pixel values. After calculation, we copy the image on the host.

2.6 Constant Scene

Since the scenes are constant and arent modified, we load them on the constant memory. This improves the performance in comparison to using global memory.

```
__constant__ Sphere spheres[] = {
    {1e5f, {1e5f + 1.0f, 40.8f, 81.6f}, {0.0f, 0.0f, 0.0f}, {0.75f, 0.25f, 0.25f},
        DIFF},    // Left
    {1e5f, {-1e5f + 99.0f, 40.8f, 81.6f}, {0.0f, 0.0f, 0.0f}, {.25f, .25f, .75f}, DIFF
        },    // Rght
    {1e5f, {50.0f, 40.8f, 1e5f}, {0.0f, 0.0f, 0.0f}, {.75f, .75f, .75f}, DIFF},
                    // Back
    {1e5f, {50.0f, 40.8f, -1e5f + 600.0f}, {0.0f, 0.0f, 0.0f}, {1.00f, 1.00f, 1.00f},
        DIFF}, // Frnt
    {1e5f, {50.0f, 1e5f, 81.6f}, {0.0f, 0.0f, 0.0f}, {.75f, .75f, .75f}, DIFF},
                    // Botm
    {1e5f, {50.0f, -1e5f + 81.6f, 81.6f}, {0.0f, 0.0f, 0.0f}, {.75f, .75f, .75f}, DIFF
        },    // Top
    {16.5f, {27.0f, 16.5f, 47.0f}, {0.0f, 0.0f, 0.0f}, {1.0f, 1.0f, 1.0f}, SPEC},
                    // small sphere 1
    {16.5f, {73.0f, 16.5f, 78.0f}, {0.0f, 0.0f, 0.0f}, {1.0f, 1.0f, 1.0f}, REFR},
                    // small sphere 2
    {600.0f, {50.0f, 681.6f - .27f, 81.6f}, {12.0f, 12.0f, 12.0f}, {0.0f, 0.0f, 0.0f},
        DIFF} // Light
};
```

Listing 4: Using Constant memory for scenes

Moreover, we had to reconstruct scene 3 completely and scene2 partially since we could find their exact sources codes. Also, it is important to note that we needed to recalibrate the gamma correction and background color values for different scenes to produce the same output as provided in the assignment

3 Results on CPU

4 Results on GPU

Following are the results obtained on parallel pathtracer at 5000 samples.

5 Performance Comparision

At smaller number of samples, the CPU and GPU have comparible performance, however, when the number of samples are increased, the GPU code outperforms CPU by a significant margin. Following

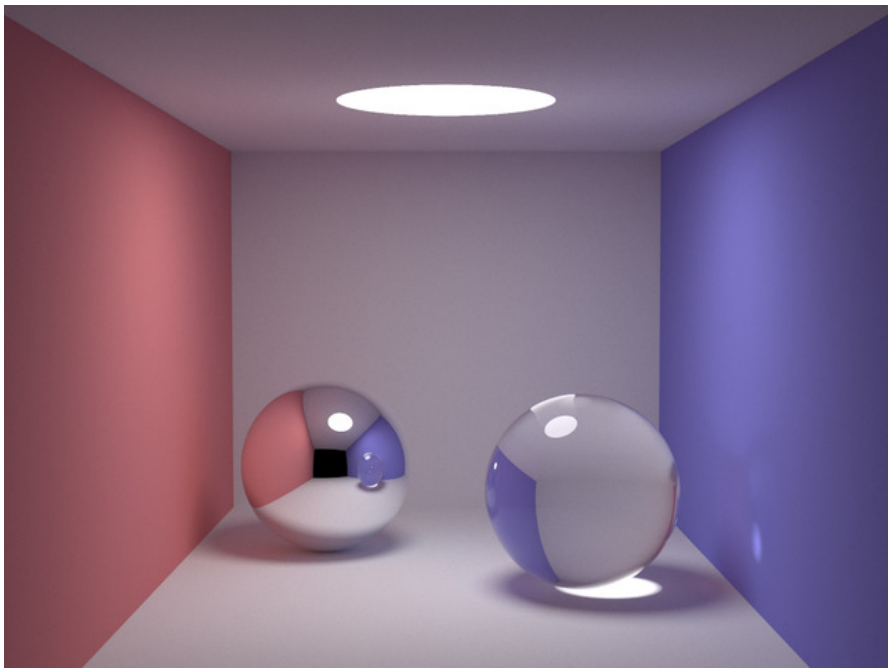

Figure 1: Scene 1 on CPU

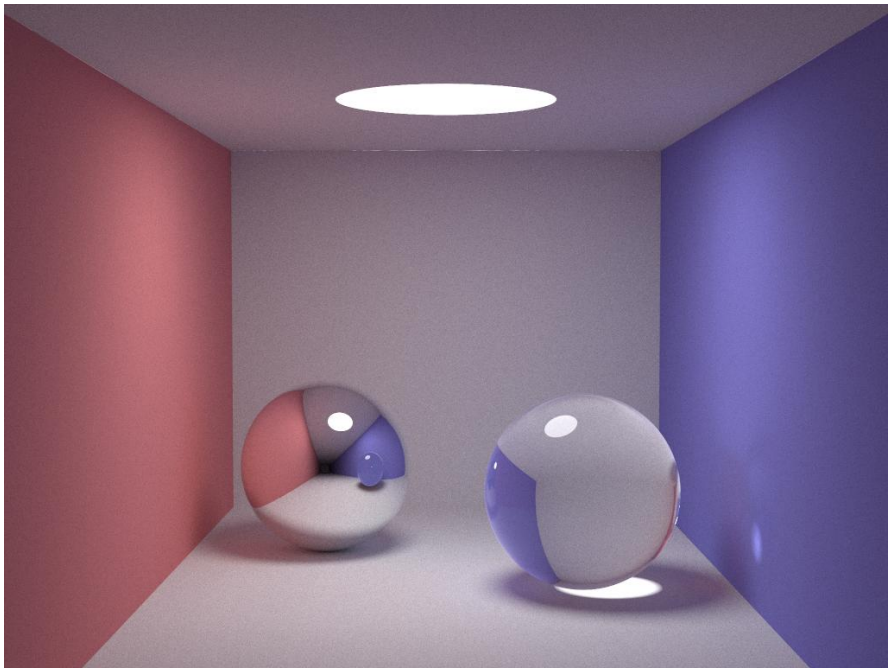

Figure 2: Scene 1 on GPU

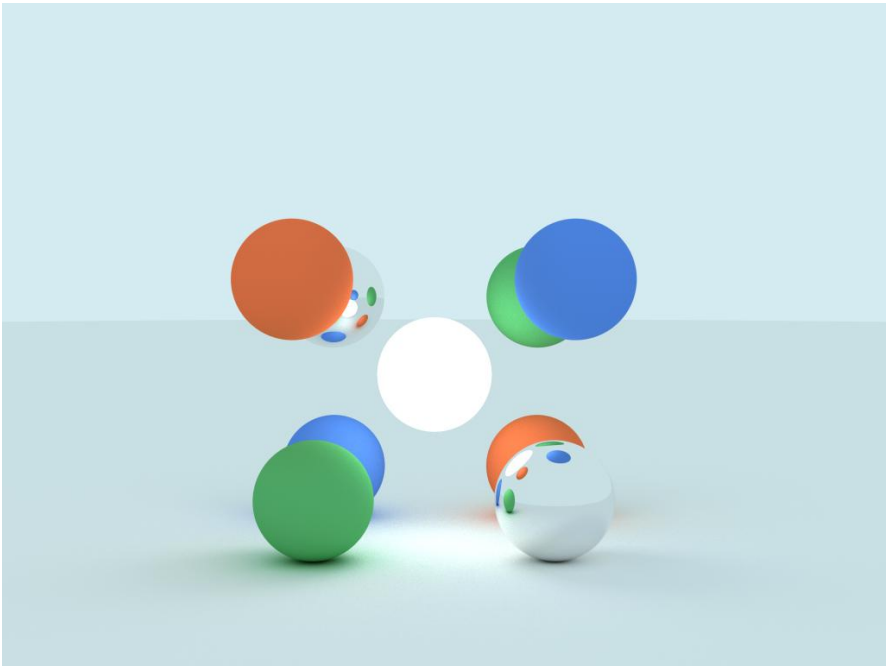

Figure 3: Scene 2: 9 sphere scene on GPU

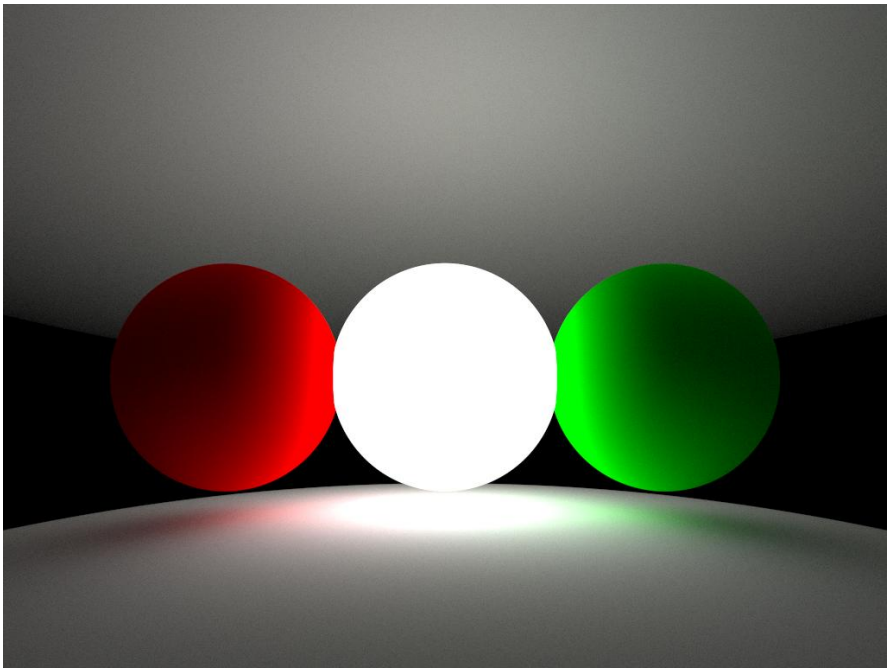

Figure 4: Scene 3: Wikipedia 3 spheres on GPU

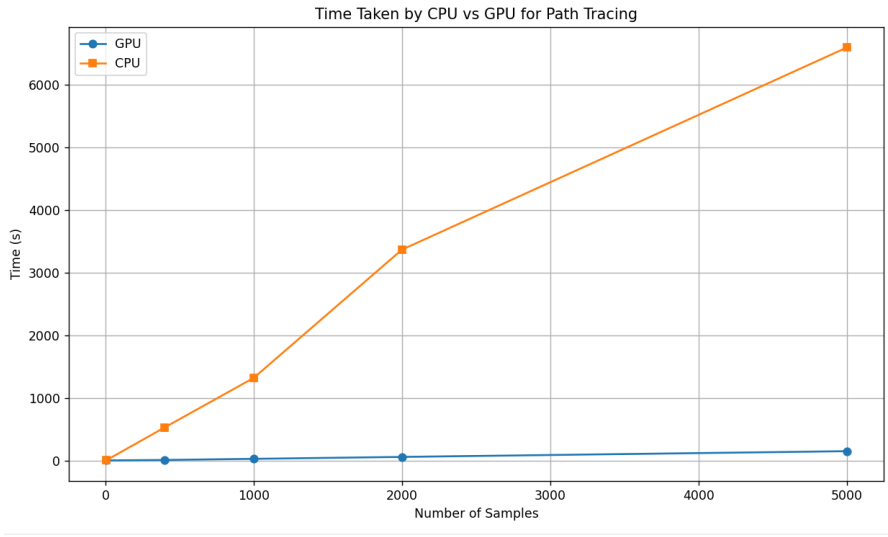

Figure 5: Performnace of GPU vs CPU

are the time comparisions of rendering scene 1 on CPU (with OpenMP) and GPU (Cuda) for different no of samples.

Hence, the GPU gives a significant speedup compared to GPU.