15618 Final Project

Parallel Optimization of the Ray-tracing Algorithm

Authors: Yunxuan Xiao(yunxuan2), Tianyi Sun(tsun2)

May 5, 2022

1 Summary

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. In this project, we try to accelerate the speed of a ray-tracing algorithm using GPU and study which method is more effective in terms of getting a higher speedup.

2 Background

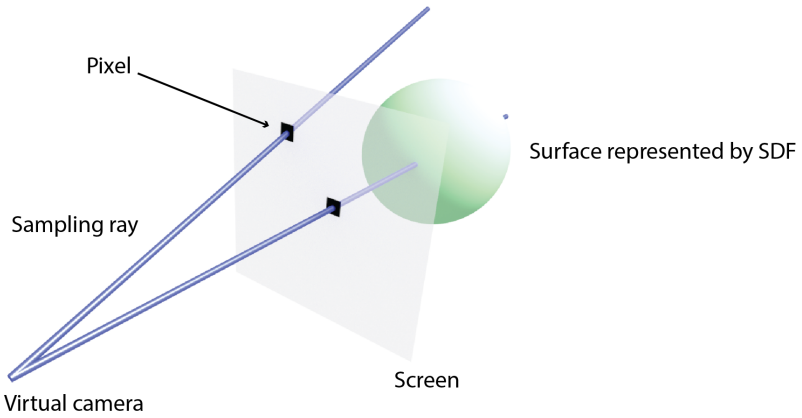

Figure 1: Illustration of ray tracing algorithm.

The basic idea of ray-tracing is to simulate the rays from a viewpoint through each pixel in the image plane. If the ray intersected with the object, we will first record the color of the object. If the object has a reflective or refractive surface, we will generate a second ray-based that does the same thing as mentioned. This is a process that actually simulates the movement of a ray in the real world and hence enables the computers to render complex optical effects.

In the physical world, millions of photons can be emitted from a single light source every second. But there will only be a small set of them that actually be seen at the viewpoint. Instead of tracing every ray which will result in a huge computation cost, we simply trace the ray back from the viewpoint, so that only the rays that actually go through the viewpoint are calculated. This can be achieved with a technique called backward ray tracing.

Another critical part is to compute whether the ray has an intersection with the object. Since this operation will be called quite frequently, its computation cost should be minimized as much as possible. We used a simple operation to check if the ray intersects with the sphere. We calculate which point on the ray has a distance to the center of the sphere that equals

the radius. If the equation has no solution, the ray and the sphere are not intersected, otherwise, it is intersected.

```
void render_pixel(colors, p) {
    colors[p] = 0;
    for s in 0...n_samples {
        sample a initial ray r for pixel p;
        initialize color c;
        for d in 0...max_depth {
            calculate the intersection point to the nearest object;
            if no intersection {
                break;
            } else {
                associate the color of intersection point to c;
                update r as the new refracted/reflected ray;
            }
        }
        colors[p] += c;
    }
    colors[p] /= n_samples;
}
```

The pseudo-code above illustrates the basic procedure of the ray tracing algorithm. The algorithm will track the number of rays coming from the camera and passing through a sample point in a pixel. It traces all objects that the ray intersects and aggregates the colors of the intersections into pixel colors according to the laws of optics. Our cuda implementation accelerates the algorithm by introducing pixel-level and sample-level parallelism and further accelerates ray-object collision detection using BVH trees. We also explored using block shared memory to reduce global memory accesses to speed up rendering execution. In the following sections, we'll dive into the details of our implementation.

3 Approach

We first test a sequential version of the ray tracing algorithm running on the CPU. We found that it took about 83 seconds to render an image with 100 spheres, which is unacceptable. Therefore, in this project, we mainly focus on implementing and optimizing the GPU version ray tracing algorithm. We first introduced a baseline implementation that supports pixel-level parallel. It split the whole image into multiple 8×8 pixel patches, then assign one thread per pixel and grouped the 64 threads into a thread block.

3.1 FP16 Quantization

Since the GPU uses its SIMD units for computation, floating-point precision will significantly affect performance. Since very high floating-point precision is not required to calculate the color of objects, we can use less precise floating-point numbers to improve performance. By replacing double-precision floating-point values with single-precision floating-point values, registers and local memory can store twice as many values, reducing memory requests and dramatically improving performance during computations.

3.2 Sample-level Parallel

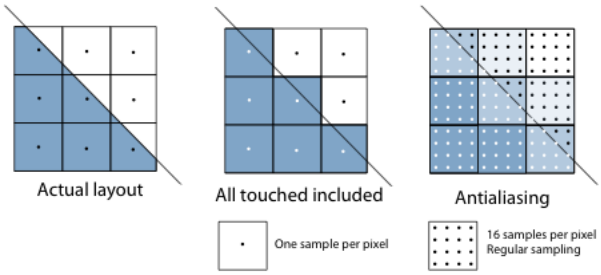

Figure 2: Explaination of anti-aliasing through pixel sampling.

As we know, pixel is the smallest unit in an image. Due to the restriction of the resolution, the edges of objects will always be more or less aliased. Anti-aliasing is a technique to soften the edges of the image to make the edges the image look smoother and closer to the real objects. This can be achieved by sampling multiple rays in a pixel and taking the average of their colors as the pixel color.

3.3 Reduce search space by BVH tree

For ray tracing algorithms, the most work is tracking which object collides with a given ray. The traditional method simply loops over all the objects in the scenario and checks whether the ray intersects with them. It records all intersections and decides which is the closest object in the ray's path. The computation complexity is proportional to the number of objects N. When there are too many objects in one scenario, the render process will traverse all the objects for each ray.

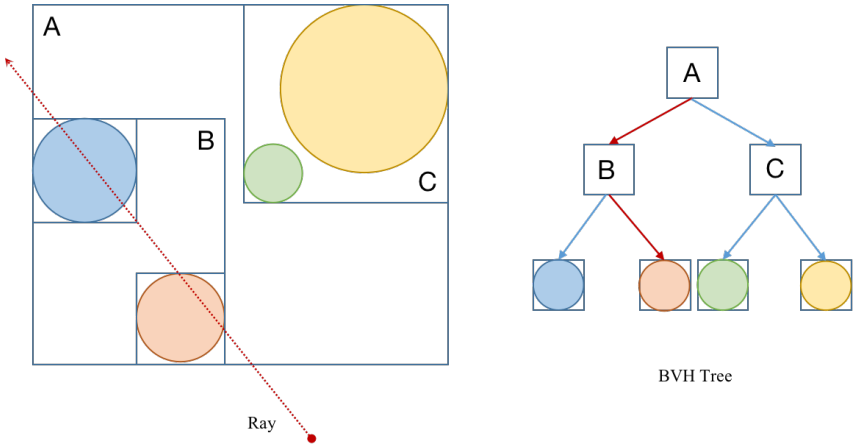

Figure 3: Example of BVH tree partitioning 2d space. Assuming that the ray intersects the bounding box A and B with the orange circle, the algorithm will search from the root node all the way down to the leaf nodes.

Here we introduced BVH(bounding volume hierarchy) Tree to recursively partition the 3D space into hierarchical bounding boxes. The key idea is to group objects with spatial proximity. As is shown in Figure 2, the leaf node represents real objects, and the internal nodes correspond to the bounding box which contains all the leaf objects in its subtree. When tracking the rays, we only have to search along with the internal nodes which have an intersection with them, thus reducing the complexity to $O(logN)$ per ray. The pseudo-code for building BVH tree is as follows:

```
void bvh::build_tree(objects, start, end, axis) {
    if (end-start == 1) {
        return objects[start];
    } else {
        sort objects from start to end along axis;
        mid = (start + end) / 2;
        bvh node;
        node.left = build_tree(objects, start, mid, (axis+1) % 3);
        node.right = build_tree(objects, mid, end, (axis+1) % 3);
        return node;
    }
}
```

3.4 Shared Memory

We have a lot of memory accesses in our baseline implementations, and most of them come from accessing the objects to check if the ray hits them. GPU shared memory can accelerate these memory accesses significantly. As we have a limited amount of objects to render, we can simply put all the objects in shared memory. The tricky part is that as the baseline version was implemented using object-oriented programming, some of the programming features are not available if using shared memory such as polymorphism. We solved this problem by changing all the object-oriented code to procedural programming code. Firstly, we allocate global memory to store the object information like its shape and material. Then we specify the amount of shared memory needed when launching the kernel function. This requires us to pre-calculate the amount of memory needed to store all the data. Then we copy all the data in global memory to shared memory and use the data in shared memory to do the following calculation. This can ensure that most of our memory access will go to shared memory.

4 Results

4.1 Configurations

In our project, we developed our ray tracing algorithm in C++ and compile it with gcc -O3 optimization. We conduct the experiment on GHC server with one NVIDIA GeForce RTX 2080(CUDA Version 11.4). The maximum number of threads per block is 1024 and the GPU Clock rate is 1545 MHz.

For experiment data, we create 3 example 3d scenarios with a different number of spheres with 3 different materials(glass, metal, and frosted). The sparse scenario contains 100 objects

randomly scattered across a 3d volume. The medium scenario contains 488 objects which mainly lie on the ground of a 2d surface. The Dense scenario contains 1453 objects uniformly located in 3d space. We will use these 3 scenarios for all the experiments in this report for comparing optimization performance. The demos of these experimental scenarios are shown in Appendix. We measure the algorithm performance with wall-clock time and speedup.

4.2 Experiment 1: Single-presicion FP Acceleration

	Sparse	Medium	Dense
baseline(double-precision)	1.19	6.01	25.61
single-precision	0.67	3.07	11.06
speedup	1.77×	1.95×	2.31×

Table 1: FP16 quantization performance.

We compared the performance of basic implementation and FP16 quantized version in three scenario settings. The results show we gained approximately 2x speedup. The experiment results perfectly matched our assumption that single-precision FP operation will double our computation efficiency. After examining several sets of output image pairs, we did not notice any noticeable deterioration in image quality.

4.3 Experiment 2: Pixel-parallel v.s. Sample-parallel

As we discussed in Section 3.2, we introduce pixel sampling to aggregate multiple rays to mitigate aliasing. In this experiment, we create one block thread for each pixel sample the reduce all the colors at the post-processing phase. The more samples we use, the higher rendering quality we can get, but at the same time, computation costs will increase accordingly.

	#samples	Sparse	Medium	Dense
pixel-parallel	10	0.67	3.07	11.06
sample-parallel	1	0.05	0.26	0.81
sample-parallel	5	0.23	0.96	3.28
sample-parallel	10	0.24	1.26	4.62
sample parallel	15	0.85	3.9	13

Table 2: Sample-level parallel performance.

The baseline implementation is pixel-level parallel, we optimized the baseline by creating #pixels ×#samples block threads per kernel and reducing the sample colors of each pixel at the end. We observed that the sample-parallel achieved 2.3×speedup when #samples=10. It does not meet the expected speedup of 10×.

The main reason behind this is the increased memory access: Previously, we only have one store action to global memory in each block thread. In the sample parallel version, we have additional #pixels ×#samples reads and #pixels writes in the reduce phase. The memory access overhead increases along with the number of samples we used. Therefore,

this method is only applicable when the image resolution and sampling factor are not too large.

4.4 Experiment 4: Spatial Partition Tree

To accelerate the process of finding the nearest intersect object, we built Bounding Volume Hierarchy(BVH) Tree on 3d space and reduced the computation complexity from $O(N)$ to $O(logN)$, where N is the total number of scenario objects. We conduct a comparison experiment between BVH tree and sequential checking implementation, the results are shown below.

	Sparse (N = 100)	Medium (N = 488)	Dense (N = 1453)
baseline	0.67	3.07	11.06
BVH tree	0.26	0.85	2.8
speedup	2.57×	3.61 ×	3.95 ×

Table 3: Experiment results with BVH tree implementation.

Experimental results show that the BVH tree achieves a 2-4×speedup in three test scenarios. When there are more objects to render, the speedup is higher. This is explainable because BVH tree can discard more irrelevant objects in the first few layers when the scenario contains too many objects.

The acceleration effect is even more significant if the object is uniformly located in the 3d space(e.g. Dense scenario) so that the BVH tree will be more balanced and shallower. The experiment results above are consistent with our assumption.

4.5 Experiment 4: Shared Memory acceleration

	Sparse (N = 100)	Medium (N = 488)	Dense (N = 1453)
baseline	0.67	3.07	11.06
Shared Memory	0.004	0.30	-
speedup	167.5×	10.23×	-

Table 4: Experiment results with BVH tree implementation.

By using shared memory, we can see that the execution speed has been improved significantly. This is mainly due to the fact that the memory access cost can be minimized by letting all the accesses go to shared memory. The downside of this approach is that as we store all the objects in shared memory and shared memory is relatively small, we can only render a limited number of objects. This is why we do not have the result in the dense scenario.

4.6 Experiment 5: Effect of max recursion depth

As we discussed in previous sections, the ray tracing algorithm will keep track of one ray until it reaches the maximum recursive depth or it hits nothing. We tried to investigate the effect of choosing different depth limits on render performance. Our assumption is that the

more complex the scene, the more recursion the ray tracer will need, since rays are likely to bounce off multiple objects. In the following chart, we plot the render time versus the different max depths which validated our assumptions.

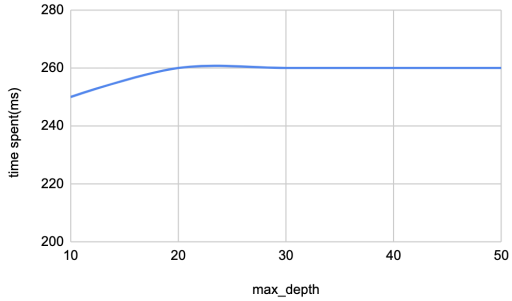

Figure 4: Sparse

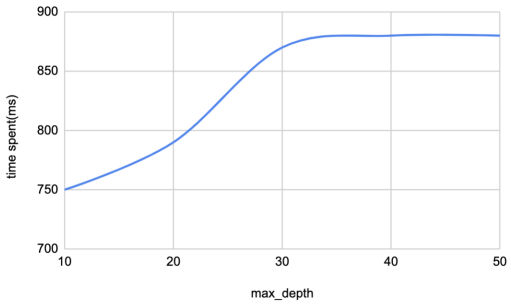

Figure 5: Medium

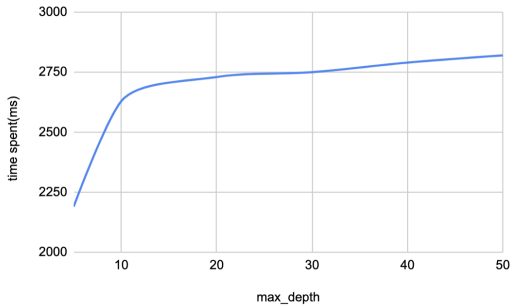

Figure 6: Dense

The experiment results show different behaviors in different scenarios. In the sparse scenario, the total time hardly changes when the maximum depth increase from 10 to 50. While for the medium scenario, the execution time keeps increasing and the converged after depth exceeds 30. For the Dense scenario, execution time keeps increasing even when maximum depth reaches 50, which indicates that many sample rays are reflected or refracted more than 50 times before projecting to outer space.

4.7 Execution Speed of All Approaches

From Figure 7 we clearly see the effectiveness of different optimization approaches. Shared memory gave us the highest speedup as it accelerates the memory access speed by more than 100x. The BVH tree method placed the second. It can greatly reduce the number of spheres we need to check by simply discarding the spheres in cubes that are not intersected with the ray. Sample parallel also gave us a noticeable speedup. It balanced the workload among different CUDA threads and therefore reduce the time wasted on the stragglers. Replacing a double-precision float point with a single-precision one gave us a 2x speedup as expected. We think that according to the characteristics of the SIMD unit, choosing the floating-point with the lowest acceptable precision is always a good practice.

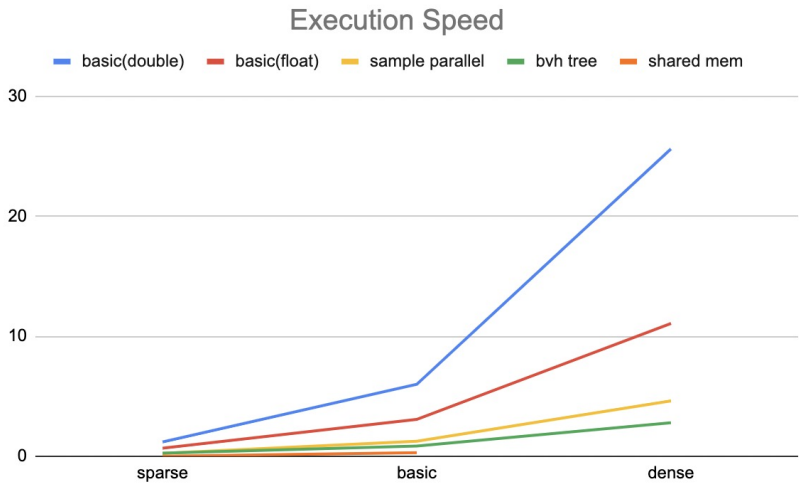

Figure 7: The execution speed of different optimization approaches.

5 Conclusion

In this assignment, we developed and optimized a CUDA-based static ray tracing algorithm. We implemented both pixel-parallel and sample-parallel versions and conducted a detailed analysis of the speedup and performance of these two methods. We then explored using single-precision floating points to save computation time and introducing BVH tree to reduce the complexity of ray-object collision detection. Experiment results show consistent improvement in rendering cost and speedups of these techniques. Finally, we explored putting all scene data into shared memory, and although there are still limitations due to insufficient shared memory size, and CUDA shared memory does not naturally support virtual function polymorphism, we still achieved significant speedups. Future directions include increasing rendering speed and achieving real-time dynamic rendering and scattering subtrees into shared memory different blocks to further accelerate the program.

6 References

[1] Peter Shirley. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html

[2] Daniel Meister et. al. A Survey on Bounding Volume Hierarchies for Ray Tracing.

[3] Juha Sjoholm. Best Practices: Using NVIDIA RTX Ray Tracing https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/

[4] Roger Allen. Accelerated Ray Tracing in One Weekend in CUDA. https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/

7 Work Distribution

Yunxuan Xiao [50% Contribution]: He conducted research on ray tracing theory and implemented code for FP16 quantization, sample-level parallelism, and BVH trees. He also designed and experimented with these optimization techniques and wrote half of the project report.

Tianyi Sun [50% Contribution]: Design and explored possible directions of optimization on GPU. He mainly focused on using block shared memory to reduce global memory access and also did experiments and deep analysis in this direction. He also wrote half of the project report.

8 Appendix

8.1 Experiment Scenarios

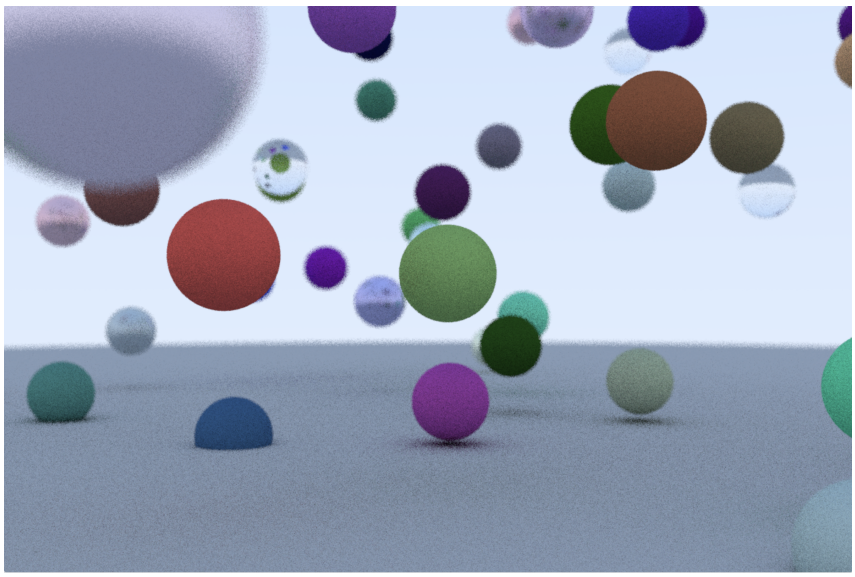

Figure 8: Sparse.

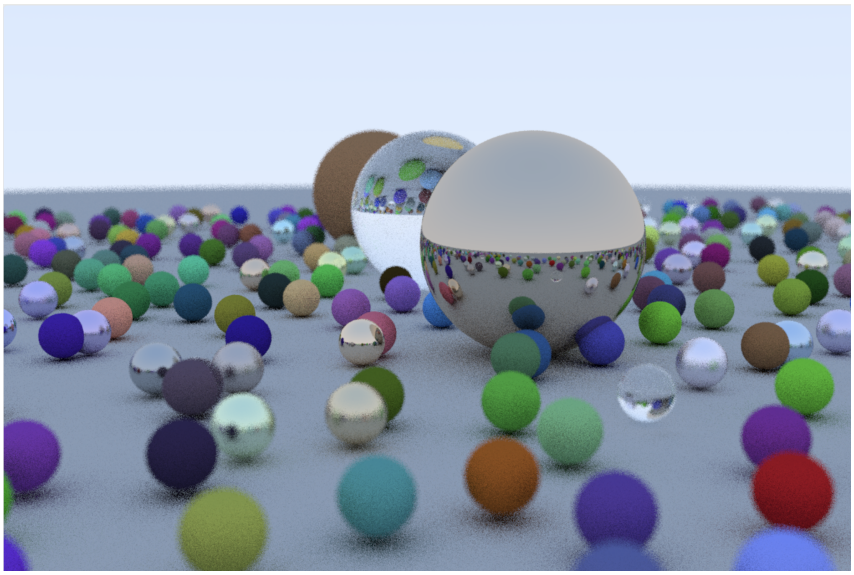

Figure 9: Meduim.

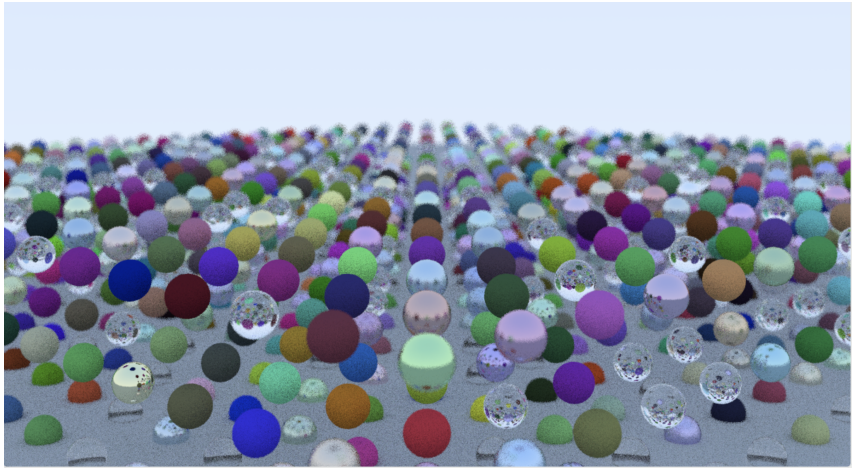

Figure 10: Dense.