15618 Project Report:
CUDA Ray Tracer

Jingguo Liang and **Ruben Partono**
Project Webpage: https://cuda-ray-tracer.netlify.app/

Summary

In this project, we implemented a serial, CPU Monte Carlo ray tracer, and accelerated using CUDA and bounding volume hierarchy.

Background

Monte Carlo Rendering

Ray tracing is a graphics rendering technique in which pixel colors are computed by shooting rays out from the camera, hitting objects in the scene, and tracing further through the reflections and refractions. This is to be contrasted with an alternative rendering method called rasterization, where geometry primitives are drawn on the canvas, which is a process that has been highly optimized by rendering pipelines that we can program with OpenGL. In general, ray tracing methods are much more compute-intensive but produce more realistic results than rasterization methods.

The basic ray tracing method shoots a single ray for each pixel, and is unable to produce effects such as soft shadows or fuzzy reflections. Monte Carlo rendering, as its name implies, makes use of Monte Carlo integration. It takes multiple samples per pixel and averages them to approximate the real color of that pixel. On hit with an object, depending on the material property, the ray will be scattered into different directions. A random generator will be in place to help determine the scattered direction, but generally the scattered direction will follow the bidirectional scattering distribution function (BSDF) of that material. In our implementation we have implemented two types of materials: lambertian and metal. When a ray hits a lambertian, it has equal probability of being scattered into any direction in a hemisphere. When a ray hits a metal, the scattered direction will be aligned to the reflected direction (with some fuzz).

The algorithm takes a scene, a camera, and parameters including image size, recursion depth, and samples per pixel as input, and outputs an image which is a rendering of the scene from the perspective of the camera. Here is a high-level pseudocode description of what the ray tracer does, using recursion:

```
for each pixel in image:
    for each sample in image:
        Create a ray shooting from the camera point towards the pixel
        Get the color of the sample by tracing the ray into the scene
    Get the color of the pixel by averaging the color of all samples
```

To get the color of a ray:

```
While recursion depth limit is not reached:
```

```
Trace the ray into the scene
If the ray hits a primitive:
    Sample a scattered ray using the material's BSDF
    Get the color of the scattered ray by tracing it into the scene (recursion)
    Return the traced color multiplied by the material's attenuation and
        the cosine of the angle between the hit normal and the scattered ray
Else if the ray hits a emissive light source:
    Return the color of the light
Else:
    Return color black (0, 0, 0)
```

In our implementation, we return a background color when a ray does not hit anything in the scene. This background color also serves as a light source for the scene.

The fact that Monte Carlo rendering takes multiple independent samples per pixel makes it a perfect candidate for parallelization. The scene is constructed once per image, and stays constant throughout the rendering. The part that is computationally heavy is the tracing part, which is also our target for parallelization. Samples are independent with each other, making the task data-parallel.

Bounding Volume Hierarchy (BVH)

In order to accelerate the process of hitting a ray with a scene, we implemented a bounding volume hierarchy structure. It is a binary tree structure where primitives in a scene are stored in the leaves of the tree, and each node maintains a axis-aligned bounding box, which is either the bounding box that encloses the two bounding boxes of its children, or the bounding boxes that encloses all its primitives. When intersecting a ray with the scene, we first intersect the ray with the root of the BVH. If there is a hit with the BVH node, we go further down into its children, until we reach the leaves and actually intersect the ray with the primitives.

Approach

Tools and Technologies

We are using C++ for the serial renderer and CUDA for the parallelized version.

Mapping from the Problem to the Machine

Each CUDA thread will map to one pixel in the image. This thread will be responsible for the computation of all samples of that pixel. Each thread block consists of 16 * 16 threads, which maps to a patch of 16 * 16 pixels in the image.

Parallelization of the Data Structures

Porting the data structures such as scene and BVH from CPU to GPU is one of the major challenges that we faced in parallelization.

In the serial implementation of scene, we made extensive use of STL containers and C++ features such as shared pointer and polymorphism. Currently our implementation supports two types of primitives: sphere and triangle. They are both derived classes of the abstract base class shape. All the primitives in the scene are maintained in a std::vector of std::shared_ptr<Shape>, so that when we intersect ray with primitives, we can utilize polymorphism to call the intersect method of each type without additional lines of code. However, as polymorphism and those C++ features are not supported in CUDA, we had to make adaptations to how our scenes should be created and maintained in GPU. The approach we took is to turn std::vectors into flat arrays, allocate them explicitly in GPU, and copy them from CPU to GPU using cudaMemcpy() method. As a substitution to polymorphism, we turned the std::vector of std::shared_ptr into multiple arrays of concrete derived classes, with each array corresponding to one derived class.

BVH is also a data structure that requires lots of adaptation in order to be ported to GPU. Intuitively, BVH nodes will maintain pointers to its children. And in building the BVH tree, the nodes will be allocated in heap memory individually as necessary. The leaf nodes will hold an array of pointers to its enclosed primitives. However, moving the tree from CPU to GPU is difficult because pointers will be invalidated once moved to GPU. Thus, we changed the way we maintain the BVH structure, such that all nodes are held in a contiguous block of memory (i.e. array of nodes), and each node will keep record of its children by storing the index of the children nodes into the nodes array.

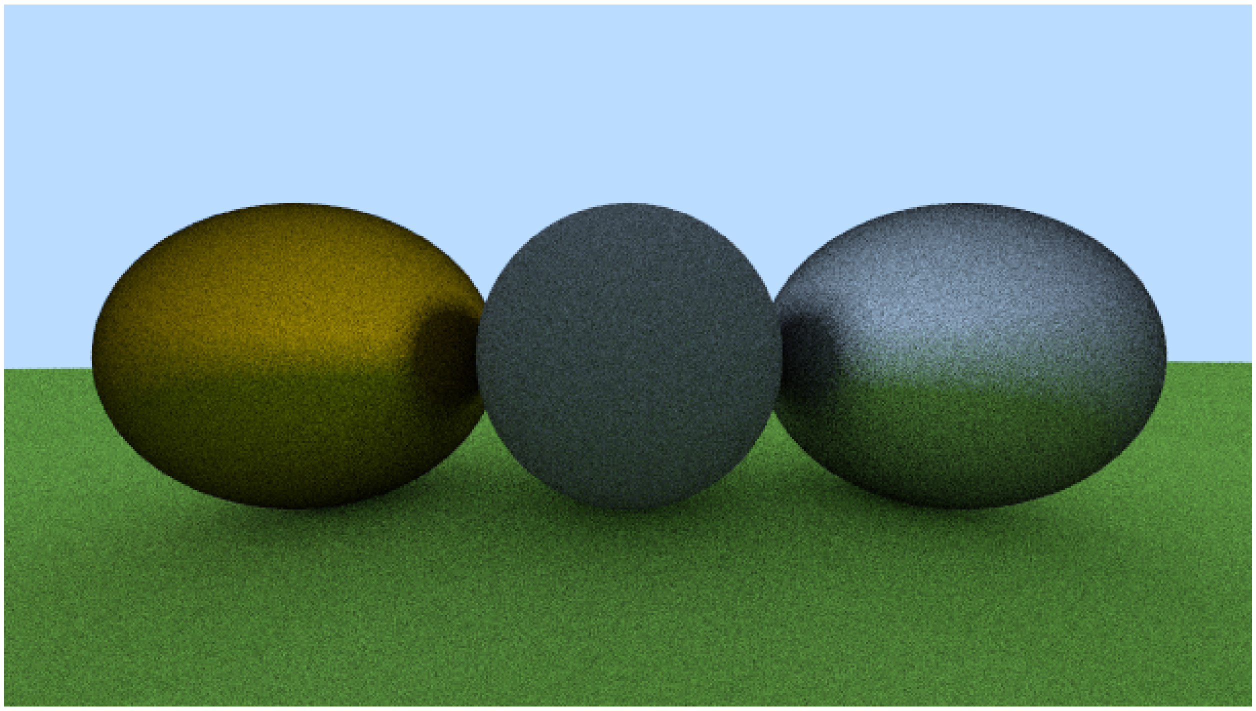
Pointers are also a challenge in parallelization. Many data structures in our serial implementation hold pointers as members. The scene maintains primitives as pointers, primitives hold materials as pointers, and BVH nodes maintain primitives as pointers. As stated above, those pointers will be invalidated if they are directly copied into the GPU. The way we tackle this issue is to first hold all the primitives and materials in flat arrays, separated according to their types. And secondly, we introduce a wrapper class, named ShapeRef and MaterialRef accordingly, to perform similar functions as pointers. They will contain an enum variable indicating which type of shape / material they are, and an index that allows us to locate them in the flat array. In this way we eliminate pointer uses in classes and are able to safely memcpy data structures from CPU to GPU.

Results

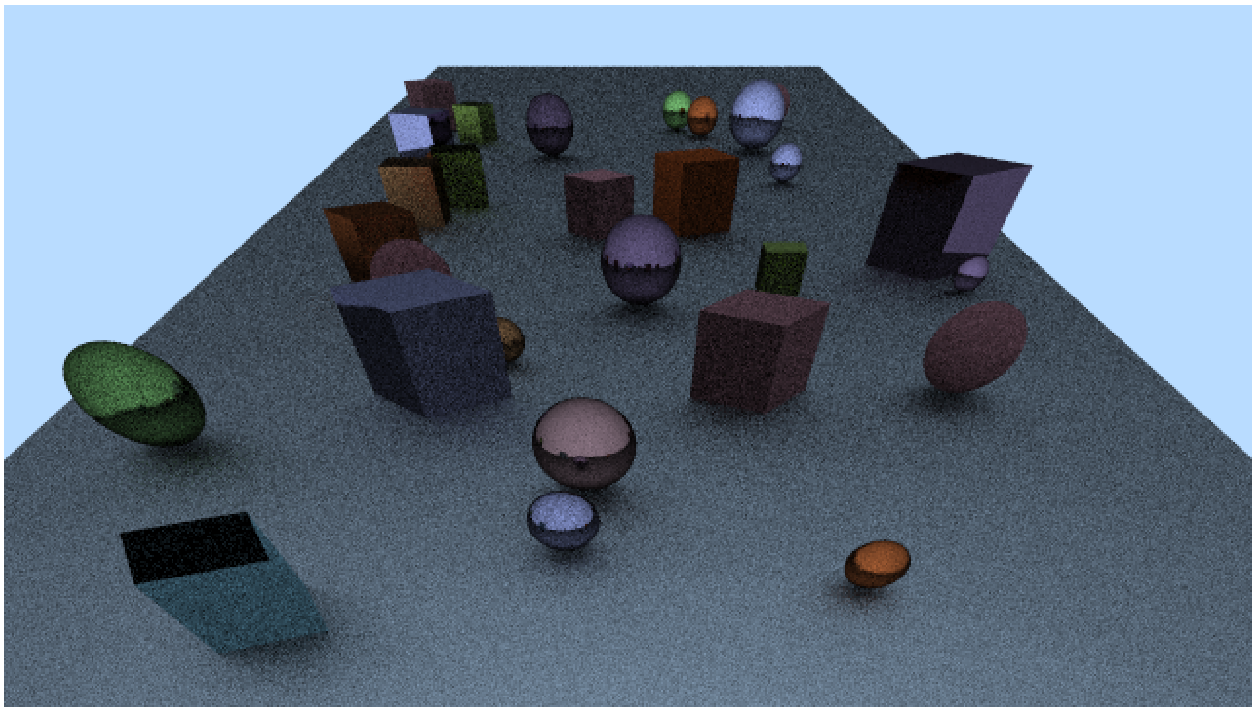
Figures

32 Samples per pixel, 640 x 480 pixels

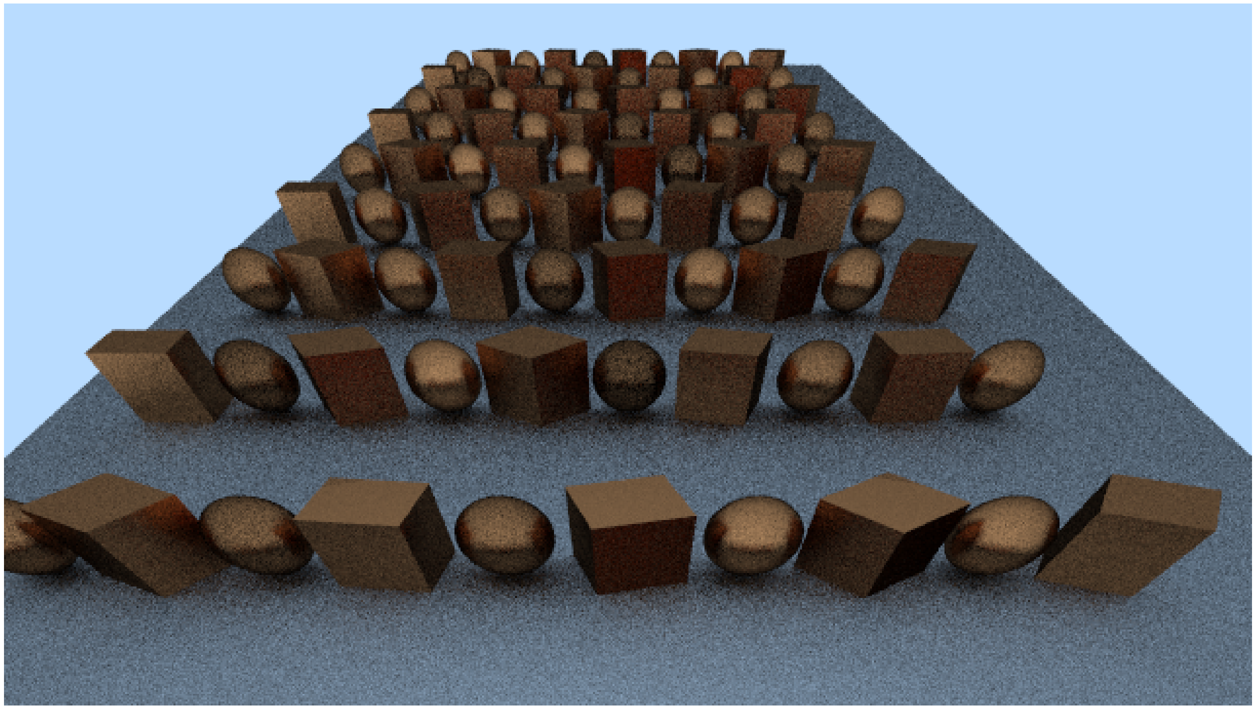
Small Scene (5 primitives)



Medium Scene (158 Primitives)



Big Scene (552 primitives)



Render times are attached at the end of this document. Below are the calculated speedups that happen due to (1) CUDA parallelization and due to (2) the bounding volume hierarchy.

(CUDA vs Serial Speedups)	Small scene	Medium scene	Large scene
1 SPP NO BVH	9.99526759	5.12652467	4.55783957
1 SPP BVH	11.1783969	4.65049721	4.30282537
32 SPP NO BVH	2.57600346	0.79671483 (*)	0.84792458 (*)
32 SPP BVH	5.4857214	1.84695831	2.12398666

(BVH vs NO BVH Speedups)	Small scene	Medium scene	Large scene
1 SPP CUDA	0.997989297	3.34384609	8.34779515
1 SPP Serial	0.892361413	3.68612402	8.8425413
32 SPP CUDA	0.549451406	7.95026583 (*)	19.7716797 (*)
32 SPP Serial	9.68932	3.42947355	7.89312548

Discussion

The speedup table above shows that our CUDA implementation achieves a speedup of around 2x to 10x over the serial implementation. There are two odd entries, however, where the workload is highest (32 samples per pixel without the help of a BVH) where the CUDA version ran slower than the serial version. Upon further inspection, these particular 32SPP runs much slower than 32x the corresponding 1SPP run. One of them is actually 50x slower. These disproportionate slowdowns are likely caused by some resource contention (available memory, memory access, etc.) that is only triggered above a certain workload threshold, though we have not collected the data to confirm what resource the problem belongs to.

Other than those anomalies, We observe a generally decreasing speedup compared with the serial implementation as the scene grows larger. This can be explained by the increasing amount of time spent on memory accesses in the larger scenes. In the small scene, where the whole scene is likely to fit into the cache, speedup with CUDA is most significant as the performance is mostly dominated by computation. And in the larger scenes, where temporal locality no longer exists, a lot of time will be spent on accessing the primitives or BVH nodes in memory, which makes the performance dominated by memory bandwidth.

For BVH vs. no BVH configurations, we observe low or even negative speedups in small scenes, and the speedup goes higher as the scene becomes larger. This is totally expected because with the small scene, the number of primitives are small. There will be only 2 BVH leaf nodes and 1 BVH root node. For intersection with BVH, it is likely that the ray turns out to intersect the bounding box of both leaf nodes. In that case the ray will still intersect with all the primitives in the scene. Taking into consideration that BVH intersection has the additional overhead of intersecting with bounding boxes, BVH configuration may achieve negative speedup compared with no BVH configuration.

However, as the scene grows larger, BVH will grow deeper and primitives will be distributed among more leaf nodes. In this case using BVH will eliminate the necessity of intersecting with many primitives, whose benefit surpasses the overhead of intersecting with bounding boxes. The larger the scene, the more significant this effect will be. Thus, we are observing increasing speedups as the scene becomes larger.

We also used the tool Nsight Compute to profile our GPU program using 32 SPP BVH configuration. The result shows that our renderKernel kernel call has a computation throughput of 27.5%, meaning that this kernel has only utilized 27.5% of the full computation capacity.

One reason for the low throughput is divergence within warps. Traces of rays into scenes will terminate as long as the ray no longer intersects with the scene. Thus there will be a lot of divergence depending on when each ray terminates. Furthermore, different rays will intersect with different BVH nodes, which in turn may intersect with different numbers and different types of primitives with different materials. This also brings a lot of divergence with a warp.

Another reason for the low throughput is the amount of time spent on memory accesses. Each trace of ray into the scene requires the access of almost all types of data, including shapes, materials, and BVH nodes.

Those accesses are highly random and independent, either spatially or temporally. Spatially, as an example, when intersecting a ray with the leaf of BVH, the ray tracer needs to access all the primitives in that node. However, those primitives are stored in potentially different arrays, or their indices differ greatly. Temporally, as an example, there is no guarantee that a primitive accessed will be accessed again soon. For each hit with the primitive, the scattered ray is sampled from the material's BSDF. This means that for lambertian materials, even hitting the primitive with the same ray multiple times will yield very different scattered rays. Thus, after several bounced in the scene, there will be no alignment among the scattered rays, which means that those rays are not likely to hit the same primitive again.

References

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

[2] Accelerated Ray Tracing in One Weekend in Cuda.
https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda

Distribution of Work

Work was distributed evenly between Ruben and Jingguo. We were working together throughout the whole project.

Ruben has worked on: scene data structure in CUDA, helping out with serial BVH, porting BVH to CUDA, construction of scenes, some measurements, website maintenance.

Jingguo has worked on: writing serial ray tracer, writing serial BVH, helping on CUDA debugging, some measurements.

Appendix: Render times

Render times in seconds

Format:
Scene name
Samples per pixel SPP
BVH usage
(serial time)
ray tracing kernel time
[speedup, if both times recorded]

Small scene
1 SPP
NO BVH
(0.287244) serial
0.028738 cuda
[9.99526759] speedup

BVH
(0.321892) serial
0.0287959 cuda
[11.1783969] speedup

BVH speedup:
0.892361413 serial
0.997989297 cuda

32 SPP
NO BVH
(9.40334) serial
3.65036 cuda
[2.57600346] speedup

BVH
(9.68932) serial
1.76628 cuda
[5.4857214] speedup

Bvh speedup:
9.68932 serial
0.549451406 cuda

Medium scene
1 SPP
NO BVH
(2.19101) serial
0.427387 cuda
[5.12652467] speedup

BVH
(0.594394) serial
0.127813 cuda
[4.65049721] speedup

Bvh speedup:
3.68612402 serial
3.34384609 cuda

32SPP
NO BVH
(65.1696) serial
81.7979 cuda (*)
[0.79671483] speedup

BVH
(19.0028) serial
10.2887 cuda
[1.84695831] speedup

BVH speedup:
3.42947355 serial
7.95026583 cuda (*)

Big scene

1SPP

No BVH
(10.3746) serial
2.27621 cuda
[4.55783957] speedup

BVH

(1.17326) serial
0.272672 cuda
[4.30282537] speedup

BVH speedup:
8.8425413 serial
8.34779515 cuda

32 SPP:
NO BVH
(289.523) serial
341.449 cuda (*)
[0.84792458] speedup

BVH
(36.6804) serial
17.2696 cuda
[2.12398666] speedup

BVH speedup:
7.89312548 serial
19.7716797 cuda <W>