15418 Final Report

Anjali Thontakudi, Winston Cheung

April 2022

1 Summary

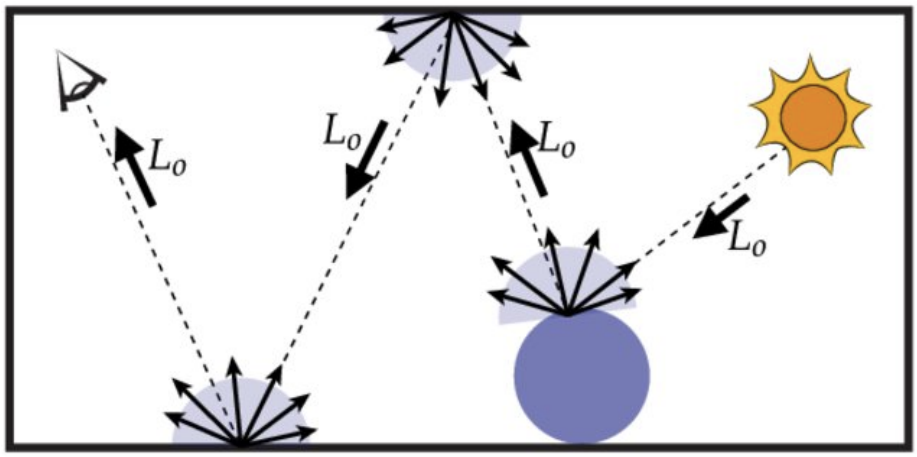
We implemented parallel BVH construction and ray tracing in openMP and CUDA, and report the speedup we found for both implementations. We ran our openMP implementation on the eight-core, 3.0 GHz Intel Core i7 processors on the Gates machines, and our CUDA implementation on the NVIDIA GeForce RTX 2080 B GPUs on the Gates machines.

2 Background

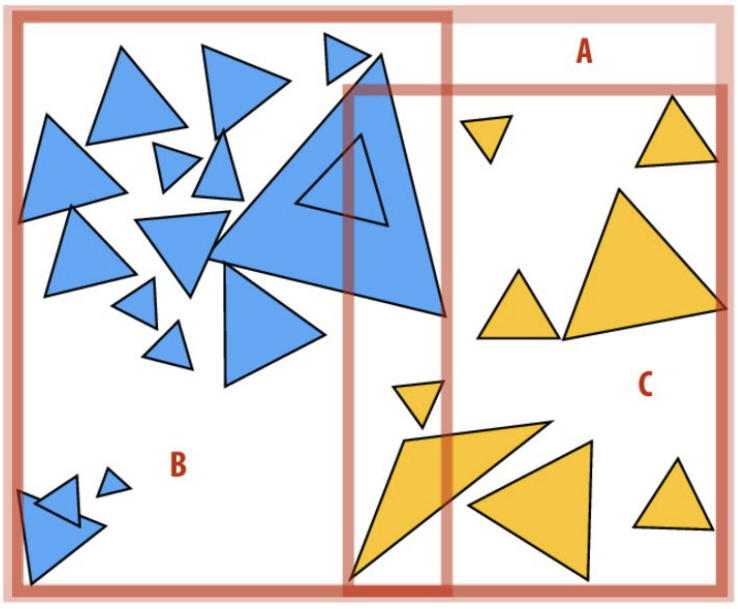
Our goal was to parallelize ray tracing and bounding volume hierarchy (BVH) construction and traversal to improve speedup.

2.1 Key Data Structures

In ray tracing, we trace a ray from the camera lens into the scene we're trying to render. As we trace the ray backwards, we detect collisions with the scene and use Monte Carlo integration to get the value of the pixel from different light sources. After detecting this collision, we can then trace the ray farther back into the scene to simulate light bouncing off of different objects and stop when the ray hits a maximum depth.



To speed up rendering, we used Bounding Volume Hierarchies (or BVHs), which split up the space into boxes. When we trace rays, instead of checking collisions with all primitives in the scene, we instead check if the ray has collided with a bounding box. If so, we can recursively check the boxes within the box, until we finally reach a subset of primitives in the scene.



2.2 Key Operations

The main operations on the BVH are creation and traversal. To create a BVH, we generally enclose the entire scene in a bounding box, then recursively build boxes along the x, y, and/or z axes until each primitive is enclosed in a box. In traversal, we usually start by checking intersection with the outermost bounding box, and recursively check the children if a hit occurs until we find that the ray has intersected a primitive.

2.3 Inputs and Outputs

In our new starter code, the input to our algorithm is a list of primitives we want to include in our scene. In this case, our scene contains a set of spheres made of different materials and placed randomly throughout the scene. The output is a rendering of the image: effectively, this just means writing the RGB values of each pixel into a .ppm file.

2.4 Benefits From Parallelization

For ray tracing, we can speed up computation by tracing rays from pixels in parallel, and check collisions with the BVH in parallel. For collision checking (which is more computationally expensive), we can also increase speedup by considering both children of the current box in parallel instead of waiting for one branch to finish before starting the next.

For BVH creation (the most computationally expensive part), we can speedup computation by building each level of the tree in parallel. Once we decide how to split the scene space, we can build the children of the current level in parallel.

2.5 Workload Breakdown

The dependencies primarily come from the recursive structure of building the BVH and checking for collisions: in both cases, we need to start at the top of the tree before building or checking the next level. There is significant parallelism in traversing/building the nodes in parallel, however: especially for a tree that's about 16 levels deep (which can occur with many primitives), there are 2^{16} leaves we can process in parallel. In terms of data parallelism, we can assign nodes that are at the same level to different processors. However, at the bottom-most level, we most likely will not achieve good locality: primitives that are allocated close to each other in memory may not actually be located next to each other in the scene, since they're randomly assigned positions to start. We could modify this by fixing the positions of the spheres in the scene, or sorting the list that contains the primitives by their position (this is something we implemented for our CUDA BVH implementation).

In general, BVH traversal and creation is not amenable to SIMD because of

its recursive structure. In terms of creating the BVH, we may be able to use SIMD by finding all nodes at a level, storing the result, and using this intermediate value for the next batch of instructions. We can do something similar for ray tracing: instead of recursively bouncing the ray through a scene, we can do one bounce, store the results of the pixel values, and then bounce all the rays again. BVH traversal, however, involves more branching since we may traverse one child or another, so rays that do not intersect a bounding box may be forced to wait for others in the same group that have more collisions.

3 Approach

We originally wanted to use existing Scotty3D code from a Computer Graphics class Anjali had taken, but we found this required libraries that we did not have permission to install on the Gates machines. Though we did remove these dependencies, they turned out to be necessary to seeing the render window, so we had to switch our starter code so we could actually see our images.

For our new starter code, we use an open source ray tracing project called "Ray Tracing in One Weekend," written in C++ by Peter Shirley. The starter code for standard raytracing does computation per pixel, sampling points and tracing a ray for a depth of up to 50 reflections. The BVH is constructed recursively: at each step, the code randomly chooses to split the space along the x, y, or z axes, and creates a bounding box around the two child nodes once they are finished being constructed. The leaf nodes are just bounding boxes around the primitives in the scene.

From this algorithm, for OpenMP, we parallelized over pixels using a dynamic work assignment, as some pixels may end up bouncing more or less times than others, and thus may finish computation faster, so we let other threads take up work as soon as they are done. This was chosen instead of the parallel BVH traversal, since the number of pixels in the image far exceeds the number of primitives in the scene. Similarly, we used tasks to implement the BVH: each thread was responsible for creating an individual node, and new threads could pick up new tasks as nodes were generated recursively. When some threads finished creating a lower-level node, they could pick up work from higher-level nodes.

For porting raytracing to CUDA, we decided to parallelize via GPU execution by launching a thread per pixel. Some care was taken to avoid blowing up the stack from the original recursive functions, which were rewritten to be iterative. Most of the work here involved porting all classes and data structures to be CUDA compatible, which meant allocating everything within the GPU and making sure that all member functions were `device` callable. In particular, the starter code makes heavy use of C++ `shared_ptr`s. At first, we tried creating our own implementation of a CUDA `shared_ptr` which counted references and freed itself manually when necessary. However, this was error-prone and buggy, so we instead replaced them with normal pointers which are allocated and freed manually at the beginning at end. Though this does not lend itself to

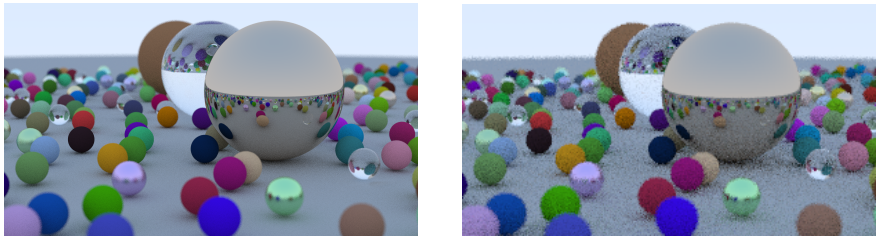
extendability very well, it sufficed for our purposes.

We also attempted to implement BVH in CUDA, which involved modifying the node and primitive definitions more heavily to allow for parallel execution. We started off with the original recursive implementation, since we had a small number of parameters for each function call. This worked for a small number of primitives (e.g. 100 spheres) in the scene, but quickly ate up too much space once we tried rendering scenes with something on the order of 10^3 spheres. We then tried a bottom-up algorithm, based on the paper paper "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees" by Tero Karras. In brief, this algorithm involves sorting the primitives in space by encoding their positions in Morton Codes, building bounding boxes around each of the primitives in parallel, and then building the internal nodes of the tree by determining the range of the sorted nodes that each internal node is an ancestor of. The latter part of this algorithm can be done by parallelizing a for loop, so each thread is responsible for creating one node once the leaves are created. It's also important to note that threads within the same block would be working on objects/nodes that are close to each other in the image and in memory: this opened up the possibility to improve speedup by mapping objects that were close to each other in the scene to the same block. This way, nodes could be calculated in shared memory, and then copied over to global memory when all blocks were finished.

While we were able to implement the Morton Codes and for-loops for building the leaves and nodes, the biggest issue was sorting the primitives. We tried using existing libraries (like thrust) to sort the primitives in parallel, but faced difficulty in modifying it so it could sort an array of pointers to primitives instead of sorting the primitives directly. Though we did find a way to implement a parallel radix sort, we discovered that the BVHs created were incorrect: the Morton Code implementation requires the positions of primitives to be non-negative x, y, and z coordinates for sorting. However, the starter code was structured to split the space into a grid with both positive and negative coordinates. This meant that primitives that were lower in the scene were all rounded up to the 0 position and ended up in the wrong bounding box, and rays that originated from pixels in the negative regions were cast from the wrong position and reflected off the wrong objects. We tried to remedy this by shifting all the axes and points by a constant amount so they would be non-negative, but this changed the starter code's ray reflection logic for different materials and objects. We ultimately ran out of time trying to fix these errors.

4 Results

We use speedup and render time as performance measurements. Everything was tested on the Gates machines, which use NVIDIA GeForce RTX 2080 B GPUs and eight-core, 3.0 GHz Intel Core i7 processors. We would have tested on PSC too for higher thread counts for the OpenMP sections, but we had issues running our code on PSC (likely because the Gates machines and PSC machines had different versions of the libraries/dependencies used).



(a) 2400px Width (b) 300px Width

Figure 1: Two renders of different sizes

4.1 CUDA Raytracing Results

We tested different image widths, with a constant aspect ratio of 16 to 9, comparing the baseline single core CPU time for a render to the CUDA time for a render. The time measured is strictly the computation time, not including any time for setting up or tearing down the scene in memory.

Our implementation improves the performance of a single CPU render by up to 164x, using CUDA code. Notably, a scene that originally took 25 minutes to render on the CPU only took 9 seconds to render on the GPU. See the following table and graph.

Width (px)	Baseline Time (s)	CUDA Time (s)	Speedup
2400	1500.18	9.14	164.13
1200	372.92	2.39	156.03
600	93.01	0.73	127.4
300	23.16	0.30	77.2
150	5.79	0.15	38.6

Table 1: Raytracing CUDA Speedup

Interestingly, the speedup increases linearly with pixel width initially, but then begins to level out at the very top, where the width is 2400 pixels. This may be because at higher pixel counts, we are better able to take advantage of more of the lanes of the GPU.

However, this speedup is not perfect, i.e. not equal to the number of lanes in the GPU, due to the fact that rays in one warp may have different number of reflections, and thus the workload distribution is not perfect. This effect is mitigated by having pixels close to each other in the same warp, as they are more likely to render similarly, but this still represents a bottleneck in the parallelizability.

4.2 OpenMP Raytracing Results

Though we were not able to run at higher thread counts on PSC, the following represents our results for applying OpenMP to raytracing for up to 8 threads on different image widths.

Num Threads	Time (s)	Speedup
1	372.92	***
2	190.19	1.96
4	98.53	3.78
8	52.83	7.05

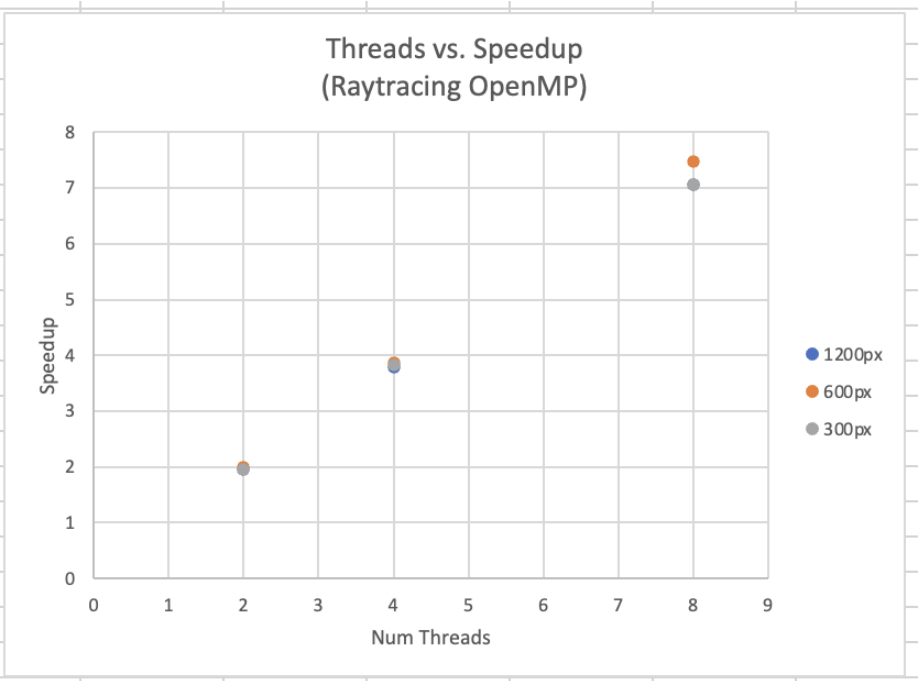
Table 2: Raytracing OpenMP Speedup 1200px Width (Gates)

Num Threads	Time (s)	Speedup
1	93.01	***
2	46.66	1.99
4	24.10	3.86
8	12.45	7.47

Table 3: Raytracing OpenMP Speedup 600px Width

Num Threads	Time (s)	Speedup
1	23.16	***
2	11.79	1.96
4	6.06	3.82
8	3.28	7.06

Table 4: Raytracing OpenMP Speedup 300px Width



All image sizes had relatively similar speedups for each thread count. In particular, they were all relatively close to linear. This mostly likely is due to having dynamic work assignment, and the fact that most pixels have relatively similar computation times, since even the longest computations are bounded by the maximum depth for reflections.

4.3 OpenMP BVH Results

We applied OpenMP to BVH for up to 8 threads, calculating speedup by comparing to a single-threaded implementation on the Gates machines. The times measured were only for BVH construction, not for tear-down of memory or rendering. We held the image size constant, and varied the number of primitives in the scene to change the size of the resulting BVH. The following table and graphs represent our results

Num Threads	Time (s)	Speedup
1	10.96	***
2	5.72	1.92
4	3.16	3.46
8	2.32	4.73

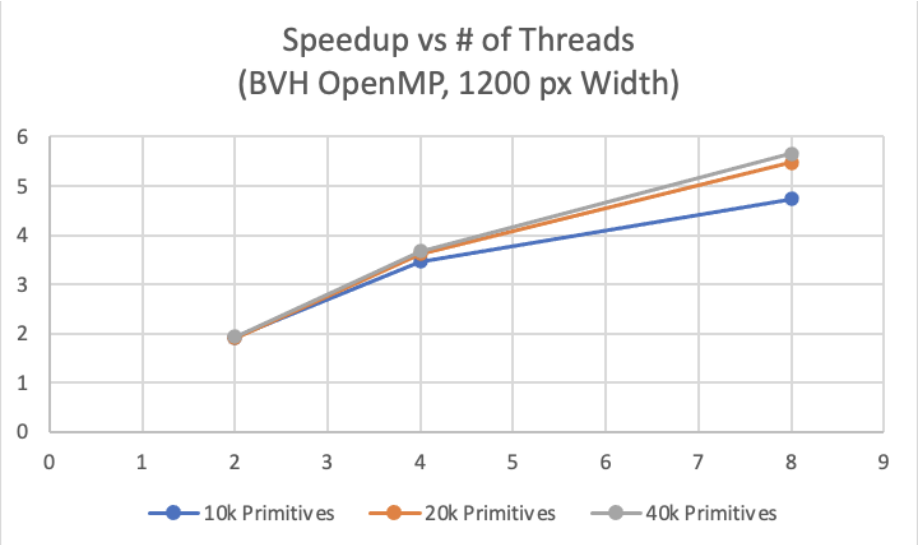
Table 5: BVH Construction OpenMP Speedup 1200px Width, 10,000 primitives

Num Threads	Time (s)	Speedup
1	45.04	***
2	23.55	1.91
4	12.44	3.62
8	8.23	5.47

Table 6: BVH Construction OpenMP Speedup 1200px Width, 20,000 primitives

Num Threads	Time (s)	Speedup
1	176.10	***
2	91.26	1.93
4	47.95	3.67
8	31.17	5.65

Table 7: BVH Construction OpenMP Speedup 1200px Width, 40,000 primitives



We can see that the speedup increases linearly with the number of threads initially, but quickly drops off once we get closer to 8 threads. This is likely because the OpenMP implementation is recursive and has long linear dependencies along each branch going from root to leaf of the tree. Ultimately, tasks that are generated higher up in the tree (for bigger bounding boxes) must wait for the lower level nodes to complete, which significantly hinders speedup. If we had more time, we could have remedied this by trying the bottom-up algorithm we switched to for CUDA: this would have allowed us to parallelize over two for-loops without creating recursive dependencies.

Interestingly, increasing the number of primitives slightly increases our speedup. This is likely because increasing the number of primitives increases the number of nodes that are at the same level in the tree: for example, we'd have 40k leaves for 40k primitives. So even though we still have linear dependencies along branches, there's more work at each level that can be done independently of other nodes, which adds to our speedup.

4.4 CUDA BVH

We ultimately were not able to implement the BVH in CUDA because of the issues with modifying the starter code to view non-negative space only, as described in the approach section. If we had more time, we would've finished modifying the starter code and tested our results by building BVHs with different numbers of primitives. One other area we could have also improved speedup is by using shared memory: in the bottom-up algorithm, objects that are close to each other in the scene are also close to each other in memory, so we could map nodes that are nearby each other in the tree to the same block. This would allow us to utilize shared memory, and copy over to global memory when all threads were finished.

5 References

Karras, Tero. *Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees.* High Performance Graphics, 2012.

Shirley, P (2018) "Ray Tracing in One Weekend" (Version 3.2.3) [Source code]. https://github.com/RayTracing/raytracing.github.io

6 Work, Credit Distribution

Anjali (50%): Removed dependencies from Scotty3D code, implemented BVH in OpenMP and attempted CUDA, wrote final report.

Winston (50%): Implemented ray tracing in OpenMP and CUDA (ported over some data structures we both needed), wrote final report