Parallel Path-Tracing in OpenCL and OpenMP

Darwin Torres Romero
dtorresr@andrew.cmu.edu

Neel Gandhi
ngandhi@andrew.cmu.edu

1 Project Web Page

https://sirlegolot.github.io/ray_tracer.html

2 Summary of Work so Far

The initial goal for the project was to use CUDA for the GPU accelerated ray tracing. However there were several issues with using CUDA. First of all, the codebase we were building on (Scotty3d code from our work in computer graphics) was simply too large to fit on AFS, which caps at 2 gb per user. Second, even if we had the space to run on the ghc machines with the Nvidia GPUs, the codebase required to install some additional dependencies which we were unable to install on ghc without sudo permissions. As such, we decided to switch to using OpenCL and run our program on a desktop we have that is fitted with an AMD Ryzen 7 3700x CPU (8 cores, 16 threads) and a Radeon RX 5700 XT GPU.

Switching to OpenCL took a bit of time as we had to get used to the slightly different syntax and APIs. One of the bigger issues we faced is the fact that OpenCL C is the main supported language, which does not make use of C++ syntax like CUDA. Given that all of Scotty3d is written with C++17 syntax and uses various C++ specific functions, we had to rewrite a significant portion of the pathtracer pipeline in OpenCL C. This includes the various math library functions on floating point vectors and matrices, the various ray intersection methods for different types of primitives and light sources, the sampling schemes for BSDFs, BVH traversal, etc. As such, time had to be spent debugging graphics math related issues instead of focusing on the parallel aspect of the code.

With respect to the parallel aspects of the code, we have so far completed the following. The first task was to use OpenMP to produce a parallelized version of the code on the CPU. We were quickly able to do this and saw a roughly 10-11x improvement in path tracing with 8 cores (with 2-way SMT).[1] We deemed this to be reasonable given there is some innate load imbalance in ray intersections. This is because while some rays can go and hit objects in the scene and bounce around several times within the scene, a lot of rays either bounce once and exit or completely miss the objects in the scene. The speedup is very much dependent on the object configurations in the scene. There is also a portion of the code where we have to synchronize accumulating the light in pixel locations, as threads were assigned work by doing a certain number of samples for every pixel location on the grid. We had to make sure to atomically add the values together.

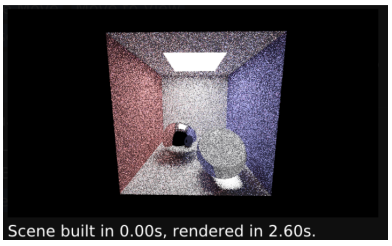
With regards to the GPU implementation, we had to make several modifications to the code base when converting to OpenCL C. One important aspect is the fact that Scotty3D was built as a recursive ray tracer, where bvh traversal and pathtracing is performed recursively. On OpenCL, recursion is not supported (which makes sense since GPU threads have limited stack space). Also, BVH's were defined recursively with C++ templates, where you were allowed to have BVHs of BVHs of different objects (spheres, and triangles). As such, we had to flatten the representation down to not be recursive and also write the BVH traversal code iteratively instead of recursively. BVH traversal is normally depth first search, which can be implemented as a stack data structure. However, using a fixed sized stack in C or even an unbounded stack implementation in C can quickly give rise to problems given that a BVH could easily consist of millions of nodes for high triangle count scenes. Instead, we

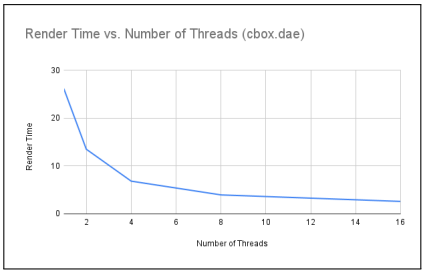
[1] On scene cbox.dae - 32 samples per pixel, 8 light samples, ray depth of 4, output image dimensions of 640x360 pixels

decided to implement a stackless-BVH traversal as described in Hapala et al. (2011) and a slightly modified version of their CUDA-specific implementation which accounted for divergent execution paths. For now, we have not implemented depth in ray traversal yet for simplicity (i.e. rays only bounce once to a light source and we collect the light from there). We also started off with no BVH traversal (simply looping through primitives in a scene to test intersections) for testing before we implemented the stackless BVH traversal.

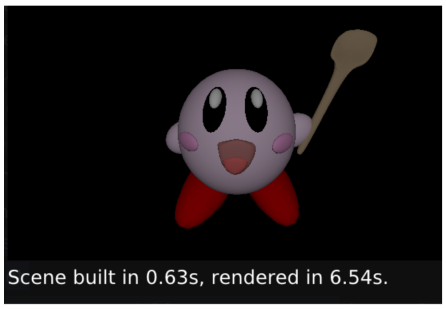
3 Preliminary Results

1. As compared to sequential code, the multithreaded CPU implementation of path tracing via OpenMP achieved a speedup of 10-11x on 8 cores (with 2-way SMT), dependent of the scene that is being ray traced.

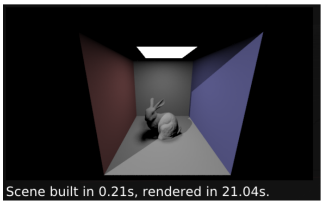


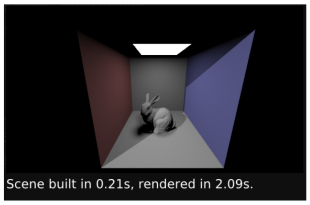


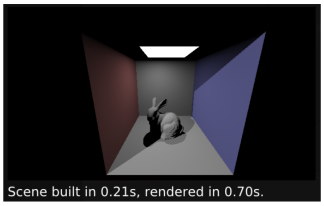
2. With no BVH acceleration, the GPU implementation achieved a speedup of 28x that of 16-thread OpenMP implementation[2]. However, large scenes were not able to be rendered due to GPU threads taking too much time when looping through primitives, and the AMD drivers on the desktop shutting down the program after thinking that the GPU has been frozen for 5 seconds.



3. With stackless BVH traversal slightly optimized for GPUs, the speedup is roughly 3x that of the multithreaded CPU implementation [3]. This is a lot lower than with no BVH, but it makes sense because of high branch divergence when traversing a BVH on a GPU. This change also helped prevent the timeout issue that was previously encountered.
4. Single-Threaded vs. OpenMP (8 SMT cores) vs. GPU —- (all with BVH traversal)







[2] On scene kirby.dae - 32 samples per pixel, 8 light samples, ray depth of 1, output image dimensions of 640x360 pixels

[3] On scene bunny_box.dae - 32 samples per pixel, 8 light samples, ray depth of 1, output image dimensions of 640x360 pixels

4 Issues

1. So far, we have implemented a GPU version of the code without too much modification of the overall algorithm approach, except for implementing stackless BVH traversal on the GPU as described in the paper mentioned earlier. There is still too much branch divergence that is preventing significant speedup beyond the multithreaded CPU implementation. We need to figure out a different approach to achieve speedup beyond what we currently have.
2. This goes hand in hand with the first one, but right now we have not implemented depth in the ray tracer. While changing the ray tracing to be implemented iteratively shouldn't be too much of a problem, the problem is that the GPU watchdog timers will stop gpu threads if they keep running beyond 5 seconds, which is extremely likely to happen if we bounce rays more than once, as each bounce requires additional intersection calculations. We will have to get around this by splitting the kernels into smaller chunks and lunch multiple kernels after each other, or some other approach.

5 Goals and Deliverables

With respect to goals and deliverables, we are slightly behind schedule, due to issues with having to switch to using OpenCL. Chances are that the "hope to achieve" goals will not be realistic due to the limited time remaining from Thanksgiving break.

For the final poster presentation, we plan to have graphs of speedup comparing various scenes as well as several pictures of renders we made with the GPU. Ideally, it would be nice to have an interactive demo where students are able to render from a select set of models live, but bringing over our desktop to the presentation location is not feasible.

6 Updated Schedule

Week 1 (10/31-11/6):
Set up the initial framework for running the renderer (potentially simplified version of the Scotty3D renderer). Get the sequential version of it running. Get OMP at least compiling, and a simple pragma for loop running.

Week 2 (11/7-11/13):
Switch to OpenCL and rewrite all logic and data structures in OpenCL C. Debug issues here.

Week 3 (11/14-11/20):
Implement no bvh traversal and stackless bvh traversal on the gpu, with not too much focus on branch divergence. Prepare milesone report with some initial results.

Week 4 (11/21-11/27):
Research approaches to reduce branch divergence, ultimately come up with a new approach than the one currently used to tackle speedup issues. Unfortunately, we will have no access to desktop being used in the project during Thanksgiving, so this week will be more focused on devising and pseudo-coding out a new approach. Testing will be limited.

Week 5-1:
Add support for ray depths greater than 1 for the GPU implementation, as well as sample counts greater than 256 (local group size limit on GPU). - Neel
Begin coding out new approach, which will potentially involve rewriting some structure of the code. - Darwin

Week 5-2:
Optimize BVH traversal code. Focus on preventing divergent execution. - Neel, Darwin

Week 6:
Further optimizations. Finalize project writeup and poster. Measure final numbers. Any last minute changes made here.