Parallel Path-Tracing in CUDA 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

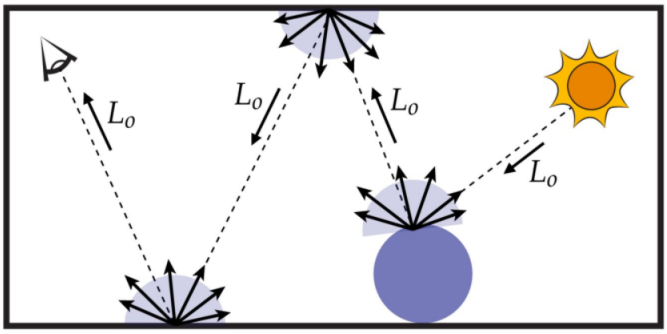
2 Summary

We are planning to implement a parallelized path tracer using the OpenMP and CUDA frameworks. We plan on starting off with a baseline sequential implementation to compare against. This baseline implementation will then be updated to make use of OpenMP to achieve speedup from parallelization. The real bulk of our project will be in the CUDA implementation and trying to optimize it to achieve similar or better results than the OpenMP implementation.

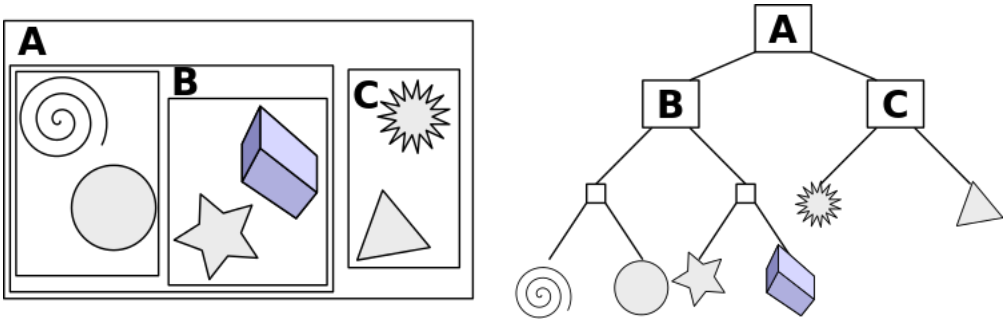
3 Background

Ray tracing is a technique used in computer graphics to render more realistic lighting of objects in a scene. The basic idea, which can be simply gathered from the name, is to trace rays from the camera to the scene, and determining lighting based on how those rays intersect with objects in the scene, as well as the material property of the objects. The main computation of ray tracing thus boils down to being able to quickly send these rays and tell whether/how they intersect objects in the scene.

In a Monte Carlo path tracer, the light contribution for each pixel in the image is a function of the material the ray from that pixel intersects with as well as integrating the indirect light contribution from around the point of intersection. The "rendering equation" which describes this is recursive in nature as for a hit point on a material, we generate new rays whose direction is in the distribution of possible angles of incoming light for the material. The contribution of those new rays is accumulated into the light value for the pixel. The diagram below visualizes this concept:



The typical naive algorithm would have each ray loop through all the objects in the scene to determine if that ray intersects with the object. This can quickly get extremely slow if the number of object in the scene is high. One widely used method to accelerate ray tracing is the use of a bounded volume hierarchy (BVH). A BVH aims to organize the objects in a scene in the form of a tree so that a ray only needs to traverse down a single/few paths in a tree to check which objects it intersects with, as shown in the diagram below.



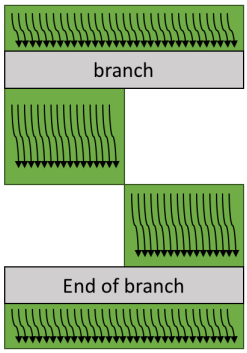
Ray tracing stands to greatly benefit from parallelism simply from parallelizing the computation of rays, since there isn't any dependency between rays. However, optimizing for different architectures is the difficult aspect of doing so.

4 The Challenge

At the surface, since each ray computation can be done independently, it seems that getting a parallel implementation up and running will not be so difficult. The only interaction between different samples are the updates of the output pixel value at the end of each calculation. There is no changing state in the scene that each sample has to worry about, meaning we can immediately start new samples without synchronization. For a framework like OpenMP, this is all good news and makes it easy to get started. However, the same can not be said for CUDA. There are other problems that must be considered when using CUDA to build a ray tracer.

One big issue is that each ray can experience vastly different execution paths, resulting in heavy divergent execution. This can mean inefficient utilization of the GPU, limiting the potential speedup we can achieve. One challenge in implementing a good CUDA ray tracer will be in preventing divergent execution by making sure we group rays experiencing similar execution paths into the same thread warps. Another problem is management of GPU memory. Complex scenes may require a large Bounded Volume Hierarchy that occupies a lot of memory. If the data can not fit in the available GPU memory, we would need to move data between RAM and GPU memory, which can hurt performance.

```
...
if ( threadIdx.x < 16 )
{
    ... A ...
}
else
{
    ... B ...
}
...
```



Visualization of divergent execution in Nvidia GPUs. Within a warp, if different groups of threads have to run separate code paths, then these code paths are taken sequentially.

5 Resources

For starter code, we aim to build off of the Scotty3D implementation we completed when we took computer graphics, although we may simplify it down or use other base code (such as the scratchapixel website). With regards to approaches to combat branch divergence, we have a document here with a list of potential papers that aim to combat it. One of the main ideas, as was discussed in lecture, is using packet ray tracing, where we try to put rays that are going down the same traversal path in a BVH together. In the case of a GPU, that is in the same warp. Other ideas include the concept of a Wide Tree, where instead of a binary BVH, use a n-ary tree, with n being 4,8,16 etc. This aims to make memory fetches more coherent and possibly improve the SIMD efficiency, though from testing on CUDA in one of the papers, it does not seem so. We aim to further read into approaches and make a better decision of what to implement by the end of the first week.

6 Goals and Deliverables

Plan to Achieve:

- Complete the sequential implementation of the Path Tracer
- Complete the OpenMP implementation of the Path Tracer. We hope to achieve a speedup of at least 6x across different scenes. We expect to see close to linear speedup, and 6-8x seems like a good target to hit with 8 cores on the GHC machines.
- Complete the CUDA implementation of the Path Tracer. We are not sure what speedup to expect as there are many factors that affect this implementation, but we are hoping to achieve better numbers than the OpenMP implementation. Our starting target is at least 10x speedup compared to the serial implementation.
- We are hoping to get more insight on the effects of divergent execution on computation times and the techniques that can be used to counteract its negative influence. Using this information, we want to determine the feasibility of using GPUs as the primary hardware for ray-tracing over CPUs.
- Have an interactive demo that runs high-quality renders of simple scenes. At the poster session, we plan to showcase completed renders of more complex scenes and graphs of speedup and render times of each implementation across various scenes.

Hope to Achieve:

- Parallelize the construction of the BVH tree
- An ISPC implementation to compare against OpenMP and CUDA

Backup Goals (If we fall behind):

- If we are unable to achieve good speedup on CUDA, we will shift our focus on performance evaluations between OpenMP and CUDA and identifying the bottlenecks that limited the performance.

7 Platform Choice

With CUDA, we can execute thousands of threads at the same time on an Nvidia GPU. We hope to utilize the GPU's numerous CUDA cores to achieve greater speedup using CUDA threads when compared to OpenMP. To compile and run our code, we plan to use the GHC machines as they already have the nvcc compiler installed and are equipped with RTX 2080s capable of running CUDA software.

8 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. Continue reading into different approaches for reducing divergence in CUDA.

Week 2 (11/7-11/13):
Get a very basic version of CUDA compiling and potentially running. Right now, just set up transferring the BVH (which was generated on the CPU) to GPU device memory. The main issues here will be with the fact that pointers will be different on CPU vs GPU memory. Also figure out how to split work if needed if scenes are very large and we cannot fit the object array in CUDA. Ignore divergence issues when first setting this up. Finalize what approaches to take to tackle divergence issue.

Week 3 (11/14-11/20):
Optimize the performance of our CUDA implementation (using ideas from papers potentially as well as our own), prepare milestone report.

Week 4 (11/21-11/27):
Further optimizations of CUDA (we expect this might take a while). If we are way ahead of schedule, start on stretch goals.

Week 5 (11/28-12/4):
Buffer in case we fall behind on schedule. If time is available work on stretch goals.

Week 6 (12/5-12/9):
Finalize project writeup and poster. Measure final numbers. Any last minute changes made here.