1 Project Overview - https://github.com/henryliu5/ray-tracer

For our final project we chose to implement path tracing in CUDA. Our main goals were to implement a reasonable path tracer on CPU, generate sample scenes to verify its effectiveness, and port the implementation to GPU to observe speedup.

The goal of our path tracing algorithm is to simulate global illumination by rendering using the full rendering equation. To estimate the integral we use Monte Carlo sampling. The below equation (notation from here) models the relationship between the rendering equation we already know and how we can estimate it using discrete samples. $L(x \to v)$ means the light going from point x to point v, E_x is the light emitted by point x, f is the BRDF, etc.

$$L(x \to v) = E_x + \int_\Omega f(x, \omega \to v) E_y \cos(\theta_\omega) d\omega = E_x + \frac{1}{N} \sum_{i=1}^{N} (f(x, \omega \to v) E_y \cos(\theta_{\omega_i}) \frac{1}{\rho(\omega_i)})$$

Our algorithm uses the idea from class where the Monte Carlo sampling is on the space of paths. Thus for each pixel in our camera we shoot multiple rays into the scene, and at each intersection point one secondary ray direction is sampled from the hemisphere. The results from the rays are then averaged together. Since we are shooting multiple rays through each pixel anyways, we add a small jitter to the ray within the pixel to achieve anti-aliasing.

To select the secondary ray direction (at least for diffuse surfaces) we implemented both uniform and cosine weighted hemisphere sampling. Using cosine weighted sampling is a form of importance sampling as since the incoming light will be weighted by the cosine angle anyways, we can better approximate the actual PDF by directly sampling from the cosine weighted distribution. Cosine weighted hemisphere sampling is enabled by default.

Our path tracer supports simple diffuse, perfectly specular, and dielectric/conductor (refractive/reflective) BRDFs. For our diffuse BRDF, we sampled the hemisphere (cosine weighted sample) to determine the direction of the secondary ray to shoot. For our perfectly specular BRDF, we assumed that the secondary ray would behave as a reflective ray. For the dielectric/conductor BRDF, we assumed that either the secondary ray will represent total internal reflection, or represent a reflective/refractive ray. The decision for total internal reflection is based on the refractive indices of the material the ray would be entering/leaving. In determining whether to reflect or refract, we approximated Fresnel's coefficient with Schlick's approximation (detailed at here), which determines "how reflective" a material is. With this, we can sample from the distribution determined by the coefficient and shoot a reflective or refractive secondary ray.

To extract more meaningful samples and determine when to terminate in an unbiased manner, we implemented Russian Roulette Path Termination, where we would terminate based on a constant that shrinks per depth. With this shrinking constant, we are reducing the likelihood of longer paths.

Lastly, we implemented Depth of Field for our path tracer through jittering the starting point and passing the primary ray through a predetermined focal point.

To migrate our path tracer to CUDA, we had to address challenges associated with control flow and memory management. From an algorithmic perspective the main differences are in structuring the problem to run on CUDA kernels and reducing branch divergence. For starters when dealing with surfaces that were a blend of surface types, (i.e. specular and refractive), in our CPU implementation we could simply just generate a secondary ray for each type at the intersection point. This was because the implementation was recursive. However, when ported to a CUDA kernel simulated each bounce iteratively due to stack size limitations and branch divergence. Thus for each intersection we could only shoot one secondary ray - to handle our different BRDFs we one again turn to a Monte Carlo technique and choose which type of bounce to produce based on a probability weighted by material type.

We wrote two "styles" of kernels structuring the path tracing problem differently.

In the first implementation our kernels are structured such that a single thread is responsible for a single pixel. The thread will iterate over the desired number of samples per pixel before averaging and writing back to a device buffer. For each sample the same thread will iterate over the desired number of bounces (depth). This implementation is fairly simple but introduces a decent amount of branch divergence due to the fact that random rays will drop out at each depth.

To address this our second implementation treats paths as disjoint sets of rays. That is, each kernel once again handles a subset of pixels, but each kernel launch is just a single bounce. We relaunch the kernels in several waves to account for the desired number of bounces and samples per pixel. This minimizes branch divergence since the behavior of a single ray is similar and we can prune out rays that terminated early.

Lastly, we also traverse our BVH iteratively rather than recursively for better performance, once again due to stack limitations and branch divergence.

2 Implementation/Engineering

We used the ray tracer project code as a starting point for this project.

Implementing the path tracer on CPU consisted of splitting off our path tracing code into a separate `tracePixel`. We added a command line argument `-s <num samples>` to make it easy to set the number of samples per pixel. We also added a `-g` flag to the command line arguments to enable/disable GPU acceleration. Note our CPU implementation is not strictly sequential as we use multithreading to parallelize the processing of each pixel - we run our CPU implementation with threads equal to the number of cores on the machine, for us 64.

Another engineering challenge associated with migrating to CUDA was getting the scene into device memory. To accomplish this we created a subset of several representations of things in the scene, such as `ray`, `isect`, `BVH`, etc. except overloaded the `new` keyword for all of them so they will be allocated on device memory. To reuse the ray tracer parser, we allow it to always create everything in host memory first. Then the first step in our GPU implementation is copying the desired subset of constructs over to the GPU version that will be on the device.

As mentioned in the previous section we have a set of kernels that represents the path tracing problem as several independent bounces. In the code we set up a `PathSegment` class that represents the segment of the path that still needs to be processed, and this contains the relevant information to h color the path's original pixel, such as color so far, attenuation, maximum remaining bounces, etc. Each call to the `pathTrace` kernel specifically will update the list of `PathSegments` to the next bounce. We also use the `thrust` library to help do a parallel partition based on which of the `PathSegments` still have remaining bounces. The next set of kernels is then only launched on `PathSegment` with bounces remaining, preventing any warps from immediately having branch divergence due to "dead" `PathSegments`. In practice this partitioning is only done every several depths, as partitioning the segments does incur some overhead.

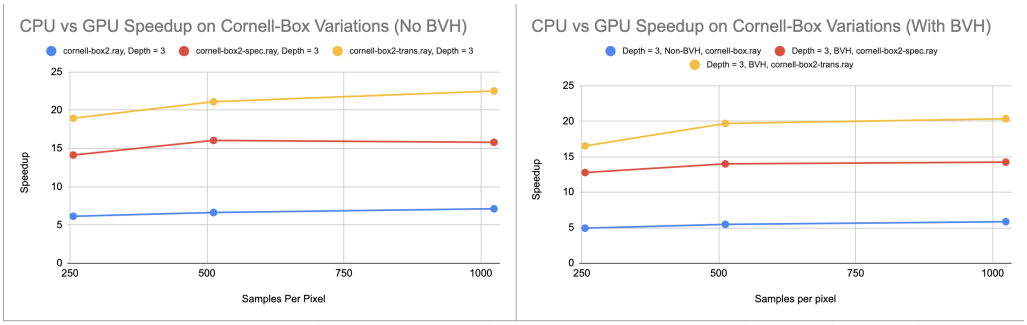

Figure 1: Speedup (CPU time / GPU time) produced by our CUDA Path Tracing implementation. Our CPU implementation is running multithreaded on 64 cores. We tested GPU using a Quadro RTX 6000.

All renderings taken for this benchmarking had a recursion depth of 3 as we noticed that the increase in accuracy/rendering detail was neglible when comparing the results of depth 5 and 7. We had three cornell box scenes that we used: cornell-box2 was standard, cornell-box2-spec had an increased specular constant, and cornell-box2-trans had a transmissive constant and refractive index. We noticed that the speed up at these samples per pixels were significantly smaller compared to significantly large samples per pixels, which we believe is most likely due to the overhead introduced from kernel launches and moving memory from device (GPU) to host (CPU). All three of these cornell box scenes are fairly simple with only 35 faces per scene, which leads to the BVH not bringing much speedup. However, after increasing the samples per pixel 16,384, we realized an 37x speedup on the cornell-box2-transmissive. We believe the GPU implementation should scale well - a test on a 40k face dragon (showed in presentation) still yields speedup scaling with samples per pixel similar to the above.

One of the primary limitations of our implementation is that currently only trimeshs are supported on the GPU. When setting up the GPU constructs we wanted to have a reduced set of classes so we could more easily debug - one particular challenge is that virtual functions on the GPU are barely supported. The virtual function table will usually be allocated in host memory so virtual functions will not work on the GPU. This made our migration much more difficult and thus why our GPU `Geometry` class uses enums to emulate virtual functions. Thus adding support for new objects is very possible but we ran out of time to do so.

As usual the entry point is practically just `RayTracer::traceImage`, where you can see where we switch to the GPU implementation and trace the code from there. Our CUDA kernels are primarily located in `path_tracer.cu` and `improved_path_tracer.cu`.