i am speed: Path Tracing on the GPU

Dora Gurfinkel Michael Jarrett

1 Introduction

Ray tracing is a widely-utilized technique for quickly generating shaded displays for scenes of 3D geometry. The algorithm consists of casting a ”ray” from the viewer’s eye through each screen pixel toward the scene, performing intersection tests with nearby geometry, and calculating the expected color of that pixel by applying a shading model, most commonly the Phong shading model. However, these shading models typically used in ray tracing algorithms often make several simplifying assumptions and closed-form approximations of true light transport, leading to inaccurate results in general.

Path tracing is an improved technique, which attempts to approximate a more general formula for shaded display, the *rendering equation*, which calculates the total amount of light to be emitted from a surface in a given direction, based on its material properties and the surrounding objects. The path tracing algorithm, rather than emitting rays, casts entire paths, consisting of several rays which collide with, and reflect off of the objects in the scene. By implementing Monte Carlo random sampling, several paths, or ”samples”, can approximate the incoming light, or ”radiance” for the pixel being traced, and as more paths are traced, the resulting color converges on the true apparent color of the object.

Because path tracing approximates the rendering equation rather than making a deterministic, simplified estimate, the algorithm can more accurately represent real-life optical phenomena. For example, a path-traced image can exhibit soft shadows, depth-of-field, ambient occlusion, and many more realistic effects which are difficult to approximate with standard ray tracing.

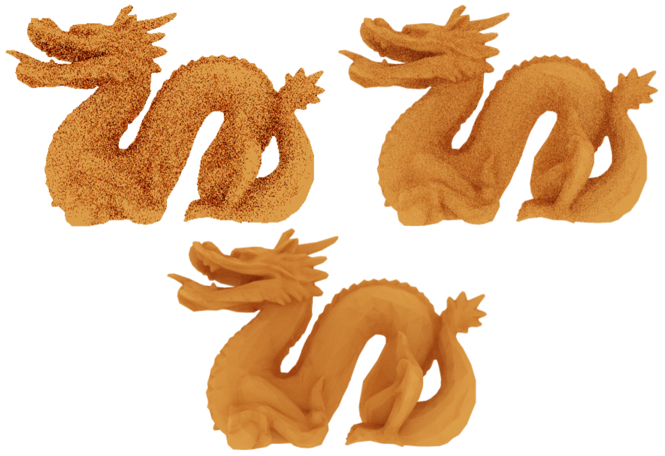
Fig. 1: Three renders of ”dragon” with (clockwise) 1, 9, and 400 samples per pixel.

However, path tracing is not without its disadvantages. Most importantly, the random sampling algorithm requires several, often hundreds, of iterations *per pixel* in order to generate a precise image. With fewer samples, the image is likely to have high color variance, resulting in a ”noisy” or ”grainy” texture (see Figure 1). It can therefore be much more computationally expensive to produce path-traced images, compared to ray-traced images.

Fortunately, since each path can be traced individually, path tracing is a highly parallelizable algorithm. In this paper, we explore a CUDA implementation for path tracing on a GPU. This is the parallelism which we seek to exploit through this project. We have successfully implemented the path tracing algorithm in a monolithic CUDA kernel called from C++.

2 Algorithm

The algorithm for path tracing consists of iterating through all the pixels in the image and casting a path through that pixel. If a ray misses the scene, then the path is terminated, and any radiance collected up to that point becomes the final color of that pixel. Otherwise, if the ray collides with an object in the scene, then another ray must be cast to continue the path. If the object has a diffuse material, then the next ray is randomly sampled from the hemisphere above that surface. If the material is glossy, then the next ray is the reflection of the incoming ray about the normal of that surface. If the material is translucent or transparent, then the next ray is cast under the surface of the object, according to Snell's law.

For each surface intersected by a ray along the path, its color ”tints” the contribution of the rays further along the path, such that the sum of these contributions more closely represents the color of light that would arrive to the viewer had a photon begun at the end of the path, and traversed the path backward, toward the eye.

To allow a wide variety of possible materials, the behavior of a ray intersection is determined probabilistically. For example, an object with a red material color and a reflectivity of `0.5` will appear to be an equal mixture of red and the reflected color of its surroundings. Concretely, for each ray intersection, a random number is generated, and the branch of code used for shading that intersection is selected based on that random number and the material properties. Averaged across several ray intersections, the material's appearance converges to the correct linear combination of reflections, diffusions, and transmissions.

3 CPU Implementation

Our solution includes a CPU-only version of the above algorithm, which uses C++11 threads to process the pixels in parallel. The image is divided into squares of a configurable size, and the blocks are added to a naïve locked queue, from which the worker threads remove a unit of work, populate each pixel with a color, and repeat. When all the blocks are processed, the threads will terminate, and the resulting image will be output into a file (png, jpg, and bmp formats are supported).

The objects are operated upon polymorphically, extending a `Geometry` class, which specifies a bounding box, and an intersection function. These Geometry instances are backed into a Bounding Volume Hierarchy (BVH), which accelerates intersection tests using spatial locality to cull the search space. Rather than having to test intersections against all triangles, one constructs from the bottom-up of bounding boxes that enclose a given geometry, to bounding boxes that enclose those bounding boxes, etc. all the way up until the root is one bounding box that bounds all geometries in the given scene. For a well constructed tree, this provides O($NlogN$) intersection rather than O(N^2). Given that this is run for every single ray, it is a very important operation to optimize. The CPU BVH implementation also benefits from the flattening of the tree discussed later in the GPU section (we applied it to both implementations). Although it did not seem to provide significant speed-up for CPU (half a second at most on dragon.ray s25, r10), it does provide a more regular access pattern and could really only help the cache.

4 GPU & CUDA Implementation

4.1 Algorithm

The GPU implementation was achieved by creating a new instance of a Tracer, a GPU-Tracer! Identical logic is performed relative to the CPU implementation in terms of constructing the BVH tree and loading the Scene. The GPU implementation then spends time copying over all of the Geometries, BVH nodes, and pixel buffer for modification on the GPU. Then, there is one MASSIVE kernel (this actually ends up being a point research for Nvidia, how to break up the notoriously massive kernel) called tracePixel. We launch it with enough threads to trace every pixel.

TracePixel then samples the pixel as determined by parameters. It performs an iterative, rather than recursive, instance of the BVH traversal and adds contributions in a fashion similar to the CPU implementation. It then modifies the pixel buffer, copies it back to host, and displays in an image file.

4.2 Optimizations

Throughout our attempt to optimize our GPU performance, we made liberal use of nvprof. This allowed us to see where the bottlenecks of our program were, namely, divergence and memory latency.

4.2.1 *Block Size*

The GP104 has 20 SM. Each SM has a maximum of 32 blocks and each block has a maximum of 2048 threads. We empirically compared runtimes for 64, 128, and 512 blocks per thread. However, we saw that 64 threads per block had the best performance so we sort of left it at that.

4.2.2 *Divergence & Russian Roulette*

When sampling paths, the intersection code continues executing until there is no intersection. One can easily come up with pathological scenes that would cause for infinite intersections if no measure is introduced to avoid this. Russian Roulette is a standard means of terminating paths without introducing bias by reweighing finite samples according to their likelihood to have happened (or that the remaining contributions were infinitesimally small). It allows us to not have to follow infinite paths to accurately represent an image, given that the number of samples approaches infinite. However, this introduces an issue on the GPU. In particular, since every thread is mapped to a pixel, if one path in a warp terminates prior to others, we start wasting compute as threads get marked as inactive. There are two remedies to this issue.

The first is to merely remove the element of probability: have all paths terminate after fixed n iterations. This way no threads will have finished prior to others, although there may still be other forms of divergence. For example, if a ray intersects with nothing on the first iteration it is inactive for the remaining n-1 iterations (WLOG for missing on the second, third, etc.). The large majority of rays miss on the first iteration because most scenes are mostly background with very few objects.

The other solution is to have a pool of work, namely rays that need to be intersected and the pixel they're contributing to, and merely pick up

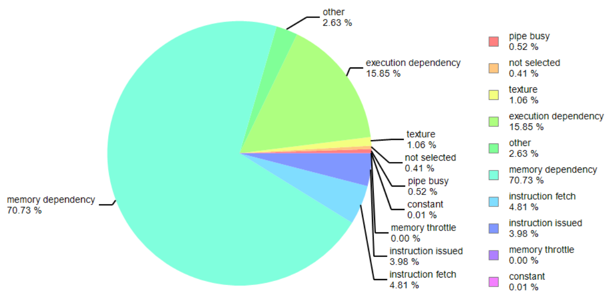

Fig. 2: Breakdown of kernel execution when rendering dragon.ray with 25 samples / pixel, max depth of 10.

work at each iteration if a particular thread's previous ray intersected with nothing. This solves the issue of wasted compute, but is slightly more complicated to implement. This solution is also compatible with Russian Roulette, which would allow us to create an unbiased image.

We began with solution one and are still working on implementing solution two. An update will come if we achieve.

4.2.3 *Single vs. Double Precision*

We are running on a GP104 chip and a Pascal architecture. This particular combination gave us access to 128 single-precision ALUs and 4 double-precision ALUs per SM. Initially, all of our calculates were using doubles, but upon learning the aforementioned tidbit, we quickly dumped all references to doubles for floats. All trigonometric operations are also done using single-precision functions.

4.2.4 *Intrinsic Types & Memory Alignment*

GPUs have massive parallelism but work just like any other architecture. That is, load/stores are done at a certain level of granularity. For Pascal, that granularity is 4-byte, 8-byte, and 16-byte aligned addresses.

Path-tracing does a lot of operations with 3D vectors. CUDA provides intrinsic support for 2D, 3D, and 4D float vectors (float2, float3, and float4) and enforces separate alignments for each. float3 is 4-byte aligned, meaning that it takes up to 3 separate loads to load a singular float3. However, float4 is 16-byte aligned. It can be loaded in a singular load. Therefore, relative to the number of loads, there is much higher throughput when one uses a float4, even if there is inefficient memory usage.

The first step in optimizing our code for this was to get rid of all GL's vec3 because they were doing absolutely nothing for us given the intrinsic CUDA types (and including glm gives us pages full of warnings). We also needed to wrap float4 in a wrapper struct to be able to override the [] operator (which for some reason isn't supported out of the box) and provide vector operations (dot, cross, normalization). Following this, Michael Jarrett learned what zero indexing was (we're proud!). You can find this source code in ”src/vec.h”

This provided a significant speedup, since initially a lot of our slowdown was because of memory latency. (Fig. 2)

4.2.5 BVH Cluster and Geometry Locality

At some point along the way, we also learned that our L1 hit rate was extremely low. That is, we didn't have access patterns and data structures that lent themselves to caches. The first step towards cache-friendliness was flattening our BVH tree. Initially, in order to copy the BVH into GPU memory we merely recursed through it, independently mallocing and copying each node (i.e. Clusters). This doesn't ensure contiguous memory allocations. We want contiguous memory allocations because ideally if all the threads in a block are hitting the same subtree, for example, that subtree could be cached in the L1 cache, rather than each thread accessing distinct parts of the trees and forcing reads from global memory. Similarly, if the majority of rays in a block are intersecting the same triangle or sphere, perhaps that Geometry could be cached as well. The operations done on Geometries and Clusters on the GPU are all read-only or const operations, and so we know that we shouldn't be abnormally worried about the overhead of cache coherency.

What was interesting as well was how one could flatten the tree. We settled on pre-order after some finagling, but provide means of flattening into in-order and post-order as well. Post-order actually performed similarly, with in-order performing slightly be worse than both. This, in a sense, makes sense. Bounding Boxes aren't necessarily disjoint using the heuristic we chose. So it is possible that a given ray will have to traverse both children of a given node. If many of the rays in a given warp also have to do this, these subtrees may benefit from caching. Both pre-order and post-order put children subtrees in contiguous memory, whereas in-order unnecessarily breaks the block of memory up with the root in the center. Or at least this is our hypothesis for why we saw a slight performance gain with pre-order.

4.2.6 Domain Partitioning

In order for caches to work in our favor, threads scheduled to the same warp or even block had to be accessing similar objects. Initially, we were dividing up blocks such that all the pixels mapped to a block formed a horizontal line 64 pixels long. That is, we broke up the image left to right, top to bottom. However, it is unlikely that for a normal scene, rays on the far left of a block would be hitting images on the far right. In order to minimize distance between the threads' pixels, we organized the distribution of work so that each block was assigned to a grid of size 8x8 pixels. Threads in a given grid are more likely to hit the same objects for natural scenes. Since all the threads in a block are scheduled to the same SM, this would allow for better cache performance.

4.2.7 Further Cache Considerations

Somewhat obsessed over the cache we thought, ”But can we do even better?” In particular, we hadn't done anything to pad to cache-line size. However, cache-line padding is primarily to decrease cache-contention; however, since we perform only read operations on cached objects, cache coherency wouldn't create that much contention, we'd just have multiple read copies. The L1 cache for Pascal architectures is 128B long; the L2 cache is 32B. Padding Clusters and Geometries to multiples of these lengths did, as expected, absolutely nothing for our speed.

5 Results

5.1 Specs

We are running Windows 10 on an Intel i9 8950hk processor with 6 physical cores. For the

GPU, we are running locally on the Nvidia GP104 Chip with a Pascal architecture. We also developed on the Eldar-10 lab machine, however saw better performance locally.

5.2 General Considerations

Some things in general restrict the speed-up that the GPU sees over the CPU:

- Open spaces: Rays that do not intersect anything are marked inactive early on in a given warp's iterations. This translates to inactive GPU cores during a majority of the program's lifespan. In the worst case, only one thread in a warp could remain active, causing massive under-utilization of the available compute resources.
- Number of scen objects: BVH traversal is the most computationally expensive task in path-tracing, aside from the sheer quantity of rays necessary to gather accurate samples per pixel. An effort has been made to increase BVH traversal efficiency on the GPU, replacing the initial recursive tree traversal algorithm with a stack-based iterative one, and flattening the BVH in memory into a large congituous array, but this still remains the most significant factor in the program's running time on both platforms.
- Finite parallelism: Although the GPU provides a massive amount of parallelism, it is still finite. A maximum of about 40k threads can be running on our chip at a given time, excluding the increase in throughput we see through things like pipelining, hyper-threading, etc. We are rendering a 512 x 512 image, which amounts to about 260k pixels. At n samples per pixels, this soon amounts to a ridiculous number of threads. There is only so much parallelism achievable on the GPU until we are merely dealing with concurrency (in that we're handling all these separate lines of execution, but aren't necessarily running them synchronously).

5.3 Scenes

For all scenes, we expect to see speed-up level off as s grows due to the finite parallelism on the GPU.

`dragon.ray` is a scene with 10,000 triangle meshes and an open scene. That is to say, it is a bad case for our GPU and we expect to see the smallest speedup for this scene. In Figure 3, we see it level off at about 3x speed-up.

`recurse_depth.ray` is an open scene with a small number of polymeshes. We expect a smaller speed up relative to path.ray as a result of the lost compute as warps along the edges of objects see some threads intersecting and other threads not at all. In Figure 4, we see speed-up peak at about 7, but on average it is below 6 and levels off at about 6.5.

`enclosed_room.ray` is an enclosed scene with a small number of polymeshes. We expect the GPU to see the highest speed up here relative to the CPU per the aforementioned. In Figure 5, we see speed-up do consistently well at north of 6, peaking close to 7. That is to say, although it is similar to performance on recurse_depth.ray, it sees a higher speed-up at lower samples indicative of its enclosed nature.

6 Future Work

We plan to continue developing this project short-term to hopefully improve GPU performance further. For example, we plan on tacking the Open Spaces issue (See §5) by switching to an iterative simulation, where paths are constructed one ray at a time, and between each iteration, rays which entirely miss the scene geometry, and thus terminate a path, will be culled from future iterations, and the remaining rays will be redistributed among the GPU cores, to maximize utilization. We also plan to expand the existing GPU-capable feature-set to include textures and cubemaps—skybox textures used to color rays which miss.

7 Sources

1) Resource on CUDA Intrinsics and Alignment
2) Nvidia Pascal Tuning Guide
3) ALU Make-up of GP104
4) Whitepaper for Pascal Architecture
5) Blog Whose References We Used Religiously

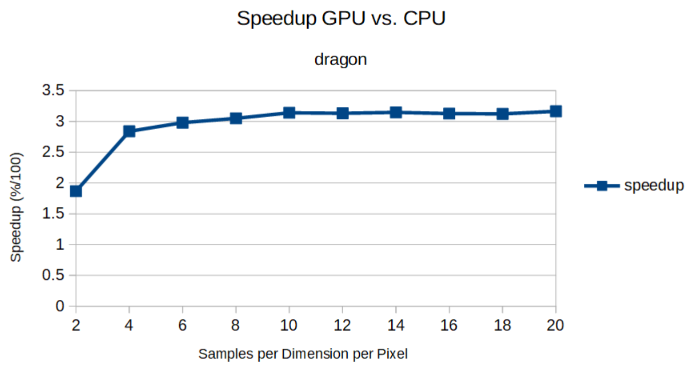

Fig. 3: Performance on dragon.ray with varying samples / pixel, max depth of 10.

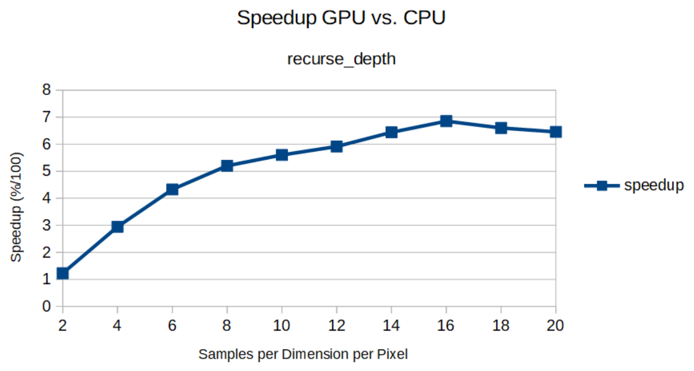

Fig. 4: Performance on recurse_depth.ray with varying samples / pixel, max depth of 10.

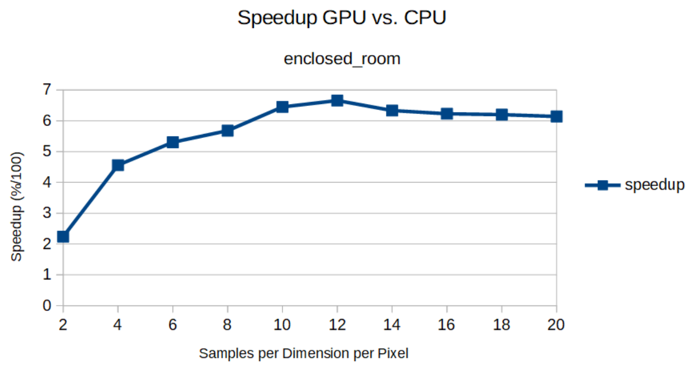

Fig. 5: Performance on enclosed_room.ray with varying samples / pixel, max depth of 10.