Final Project: Parallel Path Tracing with CUDA

Karmen Liang Margaret Allen

Dec. 13, 2019

CSCI 338 - Parallel Processing

Abstract

Computer graphics is at the core of many modern applications, such as video games, animated films, digital photography, and computer displays. To meet the needs of these computationally intensive applications, there exists many algorithms aimed at digitally generating realistic images, using methodologies that account for the underlying geometry, optics, and physics of the natural world. With the advent of the graphics processing unit (GPU)—hardware specialized for processing graphics applications—these algorithms are more easily parallelized than ever.

This final project focuses on **path tracing**: an algorithm that renders two-dimensional images of three-dimensional scenes with realistic global illumination. We will explore its parallelizability with the CUDA programming environment and discuss the different optimization strategies attempted. As with most graphics applications, path tracing is heavily data parallel, making throughput the most important benchmark in choosing the optimal parallel platform.

Along with parallelizing the algorithm on the GPU, we will explore the following parallel and non-parallel optimizations:

1) Varying thread block configurations
2) Shared memory
3) Reducing branching
4) Metropolis light transport
5) Importance sampling

I. Background: Path Tracing

Since the computer graphics method of rendering a 2D image from a 3D scene can be accomplished in many ways, we begin with an overview of the rendering algorithm specific to this application.

Note that the terms path tracing and ray tracing are sometimes used interchangeably. For our purposes, ray tracing is the umbrella term for rendering techniques that trace light rays from the eye—a camera—towards a scene to simulate realistic illumination. Path tracing is a type of ray tracing that uses randomization for rendering.

More specifically, path tracing is a Monte Carlo method of rendering images with accurate global illumination. One of its characteristics advantageous for our purposes is that it naturally simulates effects that are usually added as features to conventional ray tracing algorithms. Combined with accurate models of surfaces, light sources, and cameras, it can produce high-quality photorealistic images.

Simulating Global Illumination. An important computer graphics concept for the understanding of this algorithm is the distinction between direct and indirect illumination. With global illumination, the contribution of both direct and indirect lighting is necessary for constructing a believable image.

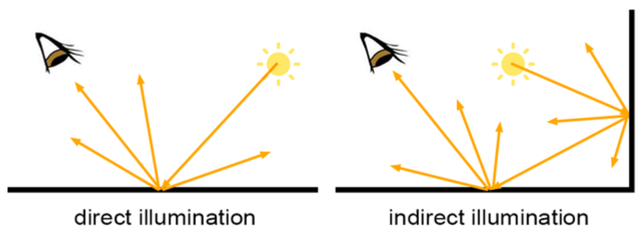

Fig. 1. Illustrating the differences between direct and indirect illumination. Realistically, the point on the ground should also take on the color of surrounding objects.

When light rays bounce only once from the surface of an object before reaching the eye, it gives the object **direct illumination**. But when light rays bounce off multiple surfaces before reaching the eye, it is called **indirect illumination**—this is why surfaces not directly exposed to a light source are not completely in shadow.

Backward Ray Tracing. In the real world, very few of the light rays reflected off an object actually reach the eye. Thus, simulating the path of a light ray from its source to the eye is highly inefficient. A more feasible method is to trace rays backward from the eye towards the scene. If the ray hits an object, we determine the light it receives by tracing another ray outward from that point.

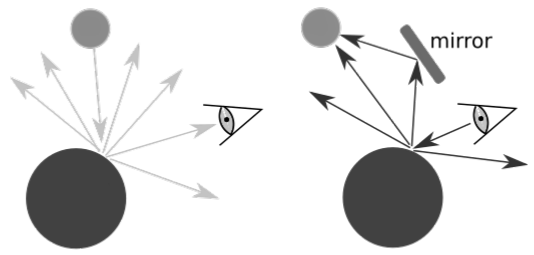

Fig. 2. Illustrating forward tracing on the right and backward tracing on the left. There are infinitely many directions for a light ray reflecting off an object to go; it is highly unlikely to reach the eye. On the other hand, every light ray traced backwards from the eye will hit something in the scene.

Monte Carlo Integration. In statistics, Monte Carlo is a method that uses random sampling to approximate a parameter. This parameter is usually deterministic in principle—in our case, accurate physical models should, in theory, allow us to output the same luminance of every point of a scene from a given initial state. However, it is too computationally expensive and essentially impossible in practice to consider the infinite number of directions from which rays can be reflected.

To get around this problem, we settle for an approximation by integrating, or gathering, the light coming from a sample of all possible directions. A Monte Carlo integration is simply an average of the sum of the light coming from these random directions. This is expressed in the informal equation:

$$\frac{1}{N}\sum_{n=0}^{N}\text{castRay}(P,\ \text{random direction above } P)$$

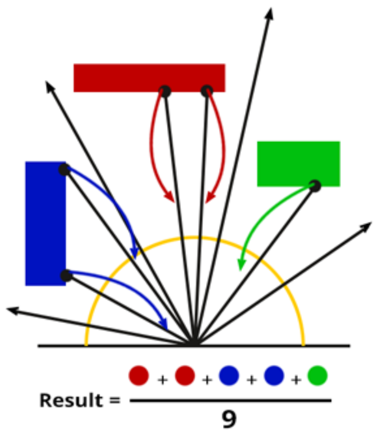

Fig. 3. A graphical representation of Monte Carlo integration. Rays are shot in random directions, and the colors of the objects they intersect are averaged.

The Algorithm. Gathering all the points discussed so far, an outline of the algorithm is as follows:

1) Cast a ray in a random direction from the eye (or camera, in this case)
2) If the ray hits an object at some point P, cast N rays in random directions from P
3) Evaluate the color of all the objects these N rays intersect
4) Average the contribution of all N rays to determine the color of P, taking surface material properties into account

Importantly, increasing N decreases the amount of noise in the image, which occurs as a byproduct of random sampling. By the law of large numbers, the image will "converge" to something that resembles natural illumination as samples per pixel increase, since we are approximating a natural phenomenon consisting of infinite light rays.

II. Serial Implementation

Our serial program implements the path tracing algorithm on the CPU. Since much of the application can be described in terms of interactions between different objects (ie. camera, spheres, rays), we decided early on to develop in C++, an extension of the C programming language that includes object-oriented features. We adapted much of the implementation from a basic ray tracer tutorial by Peter Shirley [3].

A. Parts of the Program

Output format. To begin, we implemented functionality to output an image to a `.ppm` file, a convenient text format for which our program can write out RGB values. We then call `ppmtojpeg output.ppm > output.jpg` in the terminal to produce a standard image file.

Vectors and rays. The `vec3` class allows the program to store RGB colors, locations, and directions and perform vector operations on that data. Similarly, the `ray` class allows the program to express light rays in terms of its origin, direction, and magnitude.

Positionable camera. Starting with a simple camera centered around an xyz axis, we added support for specifying the camera's viewpoint, tilt, and field of view.

Spheres. A conventional geometric object to test ray tracers with is a sphere. Calculating ray-sphere intersection is simply a matter of solving geometric functions in quadratic form, such that the number of roots in the equation determines if and how a sphere is intersected.

Surfaces. The `surface` abstract class is extended by the `sphere` class and the `surface_list` class, which maintains a list of objects that rays can hit. It has a `hit` function which is implemented by the concrete classes to calculate whether the ray hits, or intersects, an object—and if so, what point it hits.

Materials. Accurate models of surface materials allows ray tracers to produce accurate illumination on different objects. This path tracer implements a `material` class to abstract this feature from `sphere` objects. Currently, the program supports diffuse (matte) and metal surfaces.

Random scene generation. For ease of testing, the program randomly generates a scene consisting of one large sphere, representing the ground, and a medium-sized matte and metal sphere.

B. Monte Carlo Sampling

Randomization is needed at several steps in the program: to determine the direction to shoot a ray out from the camera and the direction to bounce from an intersected object. To calculate the color of every pixel in the image, the program linearly scans through the image and subjects each pixel to the Monte Carlo integration technique described earlier.

Calculating pixel colors. The path tracer take N samples for every pixel p of the scene. In other words, N rays are shot in random directions from the camera into the scene. If a ray hits p, we trace it backwards by sending it towards a randomly selected direction within a hemisphere perpendicular to the surface and centered at p.

This process continues recursively for multiple bounces, if the ray continues to intersect objects. The number of bounces is controlled by a depth parameter. The final color of p is determined by averaging the cumulative color sum of all N samples.

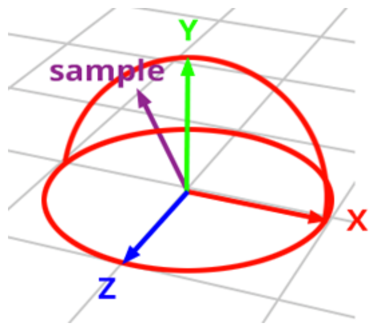

Fig. 4. An example of backward tracing for one sample that hits a point p, by a generating random ray within a hemisphere oriented on a surface normal.

C. Preliminary Testing

With a working sequential program, some initial data was gathered with a randomly generated scene of 50 spheres, using 100 samples per pixel and a backward tracing depth of 50. Testing was done with three different image sizes (600x300, 1200x600, and 2400x1200), and the average of three program executions is reported.

As expected, the execution time increases exponentially as the image size doubles. In the following graph, the time it takes for the program to complete goes from around 1 minute, to 5 minutes, to 22 minutes. Our next step is to parallelize the program with CUDA and improve the execution time.

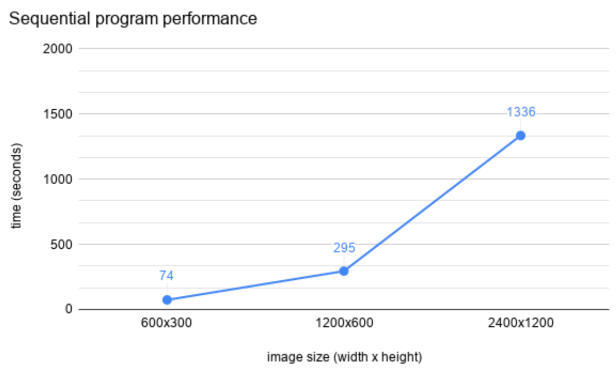

Fig. 5. Preliminary results from the sequential program. Doubling the input size (image size) increases execution time exponentially.

III. Parallelizing with CUDA

In moving towards a parallel program, we note that each light ray sample required for Monte Carlo integration is computed independently of each other. Due to these independent computations, throughput becomes the most important measure to optimize in this application. This makes CUDA the best environment to parallelize path tracing with. By taking advantage of the hardware design of the GPU, which favors highly data-parallel programs, it may be possible to see a 10+ times speed improvement in the path tracer.

The throughput-oriented design of GPUs means we must consider how to work with small caches and simple control flow while keeping a large number of threads fed with computational work. To do this, the sequential program will be split into different parts: CPU code and CUDA kernels that run on the GPU.

A. CUDA Kernels

Several portions of the sequential program can be moved onto the GPU for execution. One unexpected benefit of this modularity is that the entire program, not just the rendering portion, is sped up as a result.

Initializing random states. Random number generation in CUDA requires a specialized library called `cuRAND`. Using the device-side header file, the device can set up random number generator states and generate sequences of random numbers. In order to do this, the program must initialize and remember the state for every thread on the GPU. This is done in a separate `render_init` kernel. The resulting random state is passed to the `render` kernel, where a local copy is made and used per-thread. This ensures that every thread is referring to the same pseudo-random sequence of numbers.

Creating the scene. The program allocates the scene's objects on the GPU and calls the `create_world` kernel to construct them in a fashion similar to that of the sequential program. To mimic the random scene generation of the sequential program, another random number generator state needs to be created for the duration of this kernel.

Rendering. The meat of the parallel path tracer lies in the `render` kernel, which repeatedly calls a device-only `color` function to perform the Monte Carlo calculations. Unlike the sequential version, the parallel version of the `color` function must be written iteratively to prevent overflowing the limited stack space available.

Every thread is assigned to one pixel of the image, as determined by the the CUDA built-in variables specifying thread ID, block ID, and block dimensions. Each thread writes RGB values to a global frame buffer, which is used by the CPU to output the image. A call to `cudaDeviceSynchronize` on the host lets the CPU know when the GPU is done rendering, at which time it is safe to start writing out the actual image. In all subsequent testing of the parallel program, only the `render` kernel is timed.

B. Preliminary Testing

To compare the parallelized path tracer with its sequential counterpart, some preliminary data was gathered using the same specifications—a scene with 50 spheres, 100 samples per pixel, and a max depth of 50—and image size inputs.

The resulting results matched the trend seen in the sequential program—program time grew exponentially in relation to the input size. It was immediately clear that program execution on the GPU was, in fact, around 10 times faster than execution on the CPU. This aligns with our understanding that a heavily data-parallel program such as a path tracer could take advantage of the high-throughput design of the GPU. Graphics applications such as rendering is, after all, what graphics processors were made to accelerate.

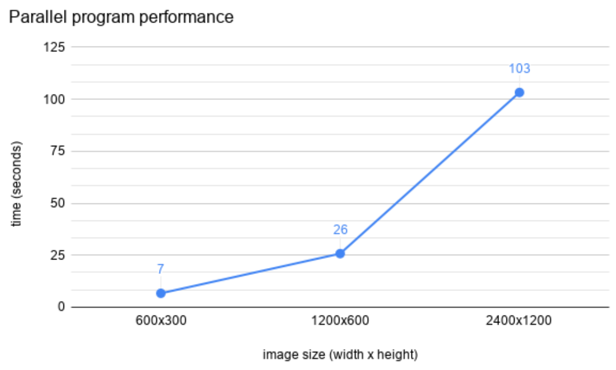

Fig. 6. Results from testing different thread block configurations. Note that this graph uses milliseconds instead of seconds as a measure of performance.

IV. Parallel Optimizations

To begin exploring possible optimizations to the parallel program, we experimented with different thread block configurations.

A. Thread Block Configuration

The configurations we tested are 4x4, 8x8, 16x16, and 32x32. A kernel using a 64x64 block configuration will not launch, due to hardware constraints that limit blocks from holding more than 1024 threads in total.

Many of the specifications from earlier tests were kept to run these tests. The number of samples per pixels was reduced from 100 to 50 to speed up the execution, and a 600x300 image size was used.

Doubling the thread block sizes naturally leads to the expectation that performance should improve exponentially. However, the results show that speedup levels off quite quickly. In fact, the only speed improvement we see is that of going from a 4x4 to an 8x8 block size configuration. After that, there is a leveling off—perhaps even an slowing down—of performance.

This may be because the larger blocks result in many unused threads—threads that must be checked against the bounds of the image and are "turned off" by taking the `else` branch of the `render` kernel. With increasingly larger blocks, there are increasingly many threads left unused, slowing down the entire program as a result.

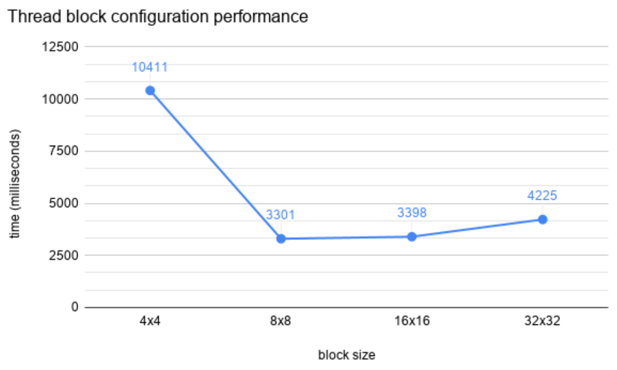

Fig. 7. Preliminary results from the parallel program. Much like the sequential program, doubling the input (image size) increases execution time exponentially.

V. Non-Parallel Optimizations

There are many other possibilities for non-parallel optimizations. These include improving branch predictability and using different memory schemes, and we will discuss a small portion of them.

A. The Problem with Shared Memory

Although shared memory has approximately 100x faster access than global memory, our parallel path tracer cannot take advantage of this. Each executing thread must have the full scene available to it to determine what objects are intersected during its Monte Carlo calculations.

Tiling memory is also not possible, for similar reasons. The fact that every thread must have a full view of the input makes it impossible to optimize by improving memory performance.

B. Reduced Branching

With every ray trace, a thread can take one of two branches, depending on whether or not the ray hits an object. Instead of making a special case for objects that do not reflect light (the background, mainly), we can simply assume it reflects some constant amount, such as zero.

Since one of the main performance bottlenecks in CUDA programs is branch divergence among threads in a warp, this could potentially greatly improve execution speed.

C. Metropolis Light Transport

A variant of the Monte Carlo method we used is called metropolis light transport. This algorithm can result in lower-noise images with fewer samples by converging to a smoother image faster. It does this by making slight modifications to existing paths, by "remembering" successful paths from light sources to the camera. In this way, difficult-to-find light paths such as those that pass through narrow corridors or small holes can be explored more easily.

D. Importance Sampling

Another technique that could potentially reduce the number of samples needed to reach an acceptable image quality is called importance sampling.

As its name suggests, the algorithm samples a probability distribution in such a way that high probability events are sampled more often than low probability events. Concretely, the path tracer casts more rays in directions where luminance is more likely to be greater, resulting in far fewer rays being needed.

VI. Future Work

Far more testing can be done to confirm the optimization possibilities outlined in the previous section. Reduced branching, for one, should be a relatively straightforward addition to the path tracer.

More testing can also be done on the different thread block configurations, since the results do not completely align with what we were expecting. We suspect that using larger image sizes would reveal a clearer, more accurate trend of exponential performance improvement.

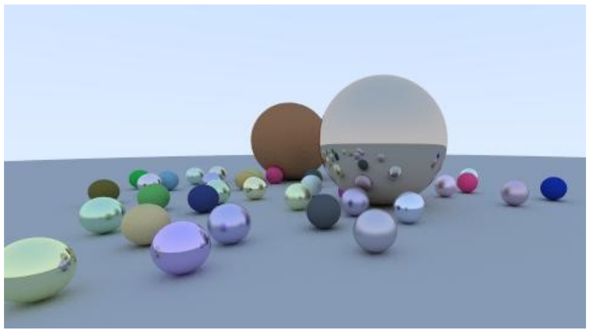

Fig. 8. An example output from the sequential program, containing 50 randomly generated spheres, 100 samples per pixel, and a max depth of 50.

References

[1] R. Allen. Accelerated Ray Tracing in One Weekend in CUDA. `https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/`.

[2] Bacterius. Importance Sampling. `https://www.gamedev.net/blogs/entry/2261086-importance-sampling/`

[3] NVIDIA. CUDA Toolkit Documentation v10.2.89. `https://docs.nvidia.com/cuda/`.

[4] Scratchapixel 2.0. Global Illumination and Path Tracing. `https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing`.

[5] P. Shirley. Ray Tracing in One Weekend. `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.