An Implementation of Ray Tracing in CUDA

CS179 Project Report

Leiya Ma
June 3, 2016

In computer graphics, ray tracing is a popular technique for rendering images with computers. In this project I implemented a serial version of ray tracing in C++, and a parallelized version in CUDA and conducted experiments to demonstrate speedup with CUDA.

1. Introduction

In computer graphics, ray tracing is a popular technique for rendering images with computers. It is capable of simulating a variety of light effects, e.g., reflection and refraction, generating high degree of photorealism. In consequence, its computational cost is high. In graphics literature, many techniques were proposed to accelerate the computation, including specific data structures and more efficient algorithms. In particular, the parallelism of ray tracing is explored in this report. Current trend of processor design is turning toward the multi-core processors. Dual core or quad core already becomes popular in personal computers. Mainstream GPUs nowadays even includes hundreds of cores. How to fully utilize their highly parallel computation capabilities to improve the efficiency of ray tracing becomes an important problem. see for more general information.

2. Background Knowledge

Given a scene of 3D objects and light sources, the goal is to generate an image from the viewpoint of a camera or eye. A 2D image plane consisting of pixels is placed between the viewpoint and the scene. The color of a pixel is the projection of the scene onto the image plane towards the viewpoint. Ray tracing generates the color of a pixel by sending the light ray through the pixel "backwards" away from the viewpoint. When a "backward" ray hits an object, it's recursively traced back depending on the material of the surface. In complex models more factors are considered, such as scattering, chromatic aberration, etc. For simplicity, we consider only reflections and refractions. A reflected ray for reflective surface and a refracted ray for translucent surface will be further traced.

in this project utilized the concept of inside-outside function for a closed surface [1]. Just like it sounds, the inside-outside function returns a negative number if a given point lies within the inside of the surface, 0 if on the surface, and a positive number otherwise. Simple inside-outside functions include:

$$\text{cube}(\text{x}, \text{y}, \text{z}) = \max(|x|, |y|, |z|) - 1$$

$$\text{sphere}(x, y, z) = x^2 + y^2 + z^2 - 1$$

$$\text{oct}(x, y, z) = |x| + |y| + |y| - 1$$

$$\text{cyl}(x, y, z) = \max(x^2 + y^2, z^2) - 1$$

However, instead of using a different function for each type of surface we wish to ray trace, it is easier to instead use superquadrics, as they can easily be defined to mimic those described above, amongst others.

Superquadrics themselves are a three-dimensional generalization of Lam´e curves (also known as Piet Hein's "superellipses"). While it is possible to extend the math to create toroids, we will be dealing solely with ellipsoids, defined by longitudinal shape parameter e and latitudinal parameter n, with the inside-outside function given as:

$$\text{sq}_{\text{io}(x,y,z,e,n)} = ((x^2)^{\frac{1}{e}} + (y^2)^{\frac{1}{e}})^{\frac{e}{n}} + (z^2)^{\frac{1}{n}} - 1$$

3. Algorithm Overview

Ray tracing has lots of advantages over the earlier rendering method. For example, the "ray casting" shoots rays from the viewpoint and finds the closest object that intersects the ray. Ray casting does not trace the rays further. The advantage of ray tracing is that it traces the lights bouncing among objects, which allows a much more realistic simulation of lighting over other rendering methods. A natural capability of ray tracing is to simulate reflections, refractions, and shadows, which are difficult using other algorithms. The computation can be naturally parallelized due to the independency among pixels. The pseudo algorithm is shown below.

Algorithm 1 `Ray Tracing`

```
1. for each pixel pixel in the image plane do
2.   Initialize:
     currentPoint ← the viewpoint
     ray ← the direction from the viewpoint to pixel
3.   return the color of pixel as COLOR(currentPoint, ray), which is calculated recursively as follows:
4.   COLOR(currentPoint, ray)
       1. if currentPoint is blocked by objects from all light sources then
       2.   return the ambient color
       3. else
       4.   Find the point hitPoint where ray first hits an object in the scene.
       5.   if hitPoint does not exists then
       6.     return the background color
       7.   else
       8.     Calculate the reflected ray reflectedRay
       9.     Calculate the refracted ray refractedRay
      10.     Set reflectedColor ← COLOR(hitPoint, reflectedRay)
      11.     Set refractedColor ← COLOR(hitPoint, refractedRay)
      12.     return combined color Combined(localColor, reflectedColor, refractedColor)
      13.   end if
      14.   end if
5. end for
```

4. Data Structure

Point represents a particular pixel on the screen. This structure considers the 3-dimensional screen, each point contains an x, y, and z coordinate which enables the ray tracing computations, such as rotation, scaling, and translation.

Ray represents a particular light ray of light which encounters an object. Once the light encounters an object, the distance and the color of the intersection surface are stored in this data structure. This data structure contains a distance member, three integer members that represent the intensity of the color's components Red, Blue, and Green.

pointLight simply holds the position, color, and attenuation constant (essentially how much energy dissipates with time for that light).

Superquadric data structure contains a position matrix, an orientation matrix, a scaling matrix, and two eccentricity values. The matrices specify the size, location, and orientation of the superquadric on the screen, and the eccentricity values indicate what shape the superquadric is. What's more, the data structure also contains three color sets: ambient color (the natural color of the object), diffusive color (affects how light is diffused onto the object's surface), and specular color (which dictates how light reflects off the object). Finally, there is a shininess attribute that dictates how *much* light is reflected.

Isq function represents an "inside-outside" function explained in section 2. The x, y, and z represent the pixel coordinates that are found through the ***Point*** data structure. The inputs "e" and "n" represent the eccentricity values of the surface. The "e" represents an "east-west" eccentricity parameter, while the "n" represents a "north-south" eccentricity parameter. The class contains methods such as ***isq***(Point *) and ***contains***(Point *) that checks the position of a point with respect to the superquadric.

Camera & Screen object are needed for the actual ray tracing implementation. The ***Camera*** object represents the "eye" when viewing the image, and the ***Screen*** image is necessary to determine the 2D projection of the superquadric object.

5. Implementation & Results

The project basically rooted from homework in previous graphics laboratory class. Where I modified the source code I implemented before as the CPU version code. However, directly modifying the CPU version code to GPU version is not realizable with the time scale. And utilizing the OpenGL to display real-time ray-traced image and enable arcball rotating is out of the scale of CS179 GPU course. Therefore, I decided to produce a Ray Tracer which produce static .ppm image.

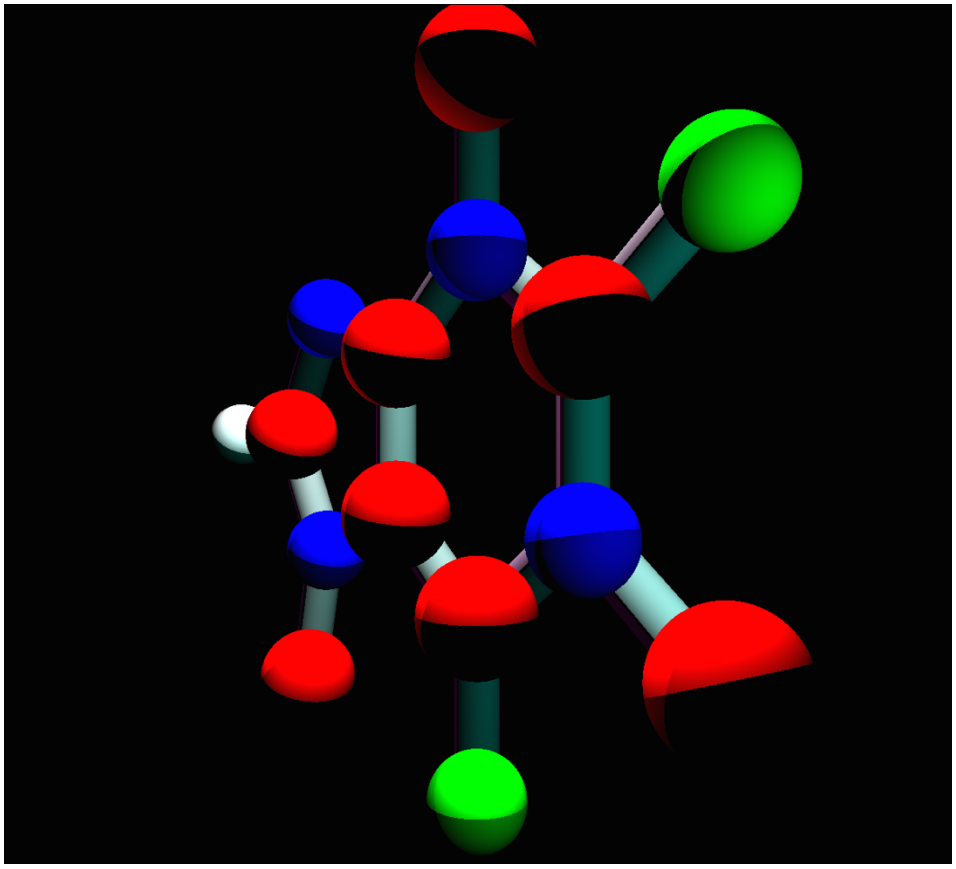
Environment: UNIX, Mako Remote Server
Language: c++, CUDA

Compile & Run instructions:

make –f Makefile
./raytracer input_image_file.txt <numBlocks> <threadsPerBlock>

The rendered image is a caffeine structure model which is an open-source txt file. I modified its structure to fit into my parser function. During the project, one challenge that I had to overcome was that CUDA does not support virtual classes and pointer references. When memcpying from the host to the device, the pointer reference may be lost. Therefore, I had to refactor the original CPU version code to exclude virtual functions and pointer references in every

classes. A consequence is that the optimization of the ray tracing decreases, but I believe that the overall gains from parallelization far outweigh the minor optimization lost.



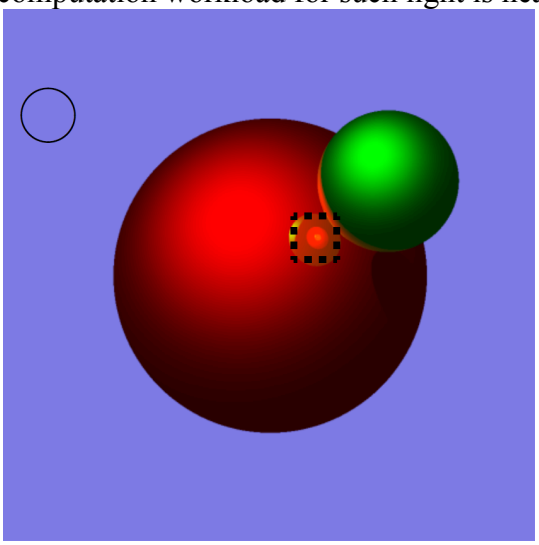
To estimate the performance of GPU parallelization, I finished this GPU computation in 460.74ms. The parameters of the image is 1920 × 1080 pixel (the figure above was modified in favor of tidy display in report). The number of blocks is 60 and the number of threads per block is 512. While the same image produced using CPU without parallelization costs nearly 232470.2ms. In consequence, we can conclude that GPU parallelization makes speed up with factor around 500. But the timing of CPU also counts into the scene preparation which may shrink the GPU speed up factor a little bit.

```
leiyama1993@mako:~/src$ ./raytracer sample.txt 60 512
Preparing for CPU Raytracing...
Preparing for GPU Raytracing...
Scene Done Being Prepared
Raytracing...
Done with raytrace...
Printing...
GPU RayTracing done! Time is 460.74
```

6. *Analysis & Future Work*

Ideally, if every pixel requires the same computation workload and GPU has N cores, the speedup of the parallelized ray tracing would be N, compared with serial implementation in a

single core processor (assume their clock rate is around the same). However, this assumption is not true: some pixels require more computation than the others. Imagine that the light from a pixel has no intersection with objects, the color of the pixel is just the background color; the computation workload of this pixel is very light. On the other hand, if the light intersects with an object, then the color of the pixel should be the color of the object, which may include the color of other objects that have reflections on it. The computation workload for such light is heavier.



The figure above demonstrates the variety of workloads among pixels. Workload for pixels in the circled region are minimum in the sense that the tracing will return after one step, as the light only "intersects" with the background. In the counterpart, workload for pixels in the boxed region is high. The red ball and the blue ball reflect onto each other multiple times. When we trace the light starting from this region, the number of steps is much more than the circled region.

This unbalanced workload hurts the performance. The total execution time of the algorithm is bound by the pixels whose computing time is the longest. Thus, the benefit gained through parallelism is greatly limited. More importantly, adding more cores doesn't solve the problem as the "busiest" pixel is still computed by a single core. This means that future processors with more cores cannot not improve the execution time, but only reduces the parallel efficiency, which is a very bad news for a parallel implementation.

I would continue to study the workload balancing problem of ray tracing. In general, two types of strategies can be applied to solve the problem: static scheduling and dynamic scheduling. The static balancing estimates the workload beforehand, and assigns the workload to computing units accordingly. The dynamic balancing adjusts the workload for different computing units on the fly.

7. Reference

1) http://courses.cms.caltech.edu/cs171/
2) A.Arthur, "Some techniques for shading machine renderings of solids," in AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference. Atlantic City, New Jersey, New York, NY, USA: ACM, 1968, pp. 37–45.
3) J. F. Blinn, "Models of light reection for computer synthesized pictures," in SIGGRAPH'77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques. New York, NY, USA: ACM, 1977, pp. 192–198.
4) https://en.wikipedia.org/wiki/Ray_tracing_(graphics)