GPU-Accelerated Ray Tracing in OpenCL

Jason Seaman – jass2@umbc.edu, Charles Fox – charfox1@umbc.edu

I. Introduction

OVER the last 30 years, ray-tracing has become an increasingly feasible way to dynamically render images in computer animation and computer graphics. Ray-tracing is historically hindered by the computational cost of each pixel calculation. Depending on precision, a ray that reflects into a series of objects with high specular coefficients could yield a highly recursive coloring process. This expense is possible for every pixel in an image.

As a result of steady developments in SPMD (Single Problem, Multiple Data) hardware, ray-tracing capability is a selling-point for many modern GPUs (Graphical Processing Units). In this paper, we describe our process of integrating our own ray-tracing program into OpenCL, a parallel framework designed to speedup massively parallel computations. In the following sections, we will describe our motivations, technology, results, difficulties, and future areas of work should we choose to expand upon our project.

A. Motivation

October of 2019, students of Dr. Adam Bargteil's graphics course at the University of Maryland, Baltimore County were tasked with implementing their own ray tracer from scratch. Having enrolled in this class, our frustrations with the meticulous process of precisely implementing mathematical processes programmatically was exacerbated by the run-time of the standard serial execution of a ray-tracer. The process was expensive: for every pixel, calculate a ray's trajectory, loop over the shapes in the image to find an intersection, loop over the shapes again to determine a shadow and recursively repeat the process for a specular reflection ray. A mere dual-core 1.8GHz MacBook Air took over 30 minutes to generate the single resulting image from the base assignment. The frustration we experienced in those long running times provided half of our motivation for our project.

The other half of our motivation came from our desire to develop code for a specific piece of embedded hardware. Courses at UMBC mention embedded systems and GPU acceleration as extensions to many course topics, but seldom provide direction or assignments to explore those applications of systems. Without providing students with a standard device, we believe it's unreasonable to expect a course to meet that request. Therefore, we took the initiative here to explore this topic on our own in this project below and are very satisfied with the results.

B. What is Ray Tracing?

The idea behind the ray-tracing rendering technique is simple. Given Cartesian coordinates of a view, or camera, calculate what objects in a scene are visible to the camera between the eye and infinity. Every shape and property of the output image is outlined by an input image, in our case, an neutral file format (.nff) file. This file type provides the details of the scene including background color, camera location, location of light sources, and shapes, each with their own color, location, and format for shading properties.

A ray can be fired for each pixel in an image. Each ray is defined with an origin and direction. The origin for each pixel, is the camera coordinate. The direction is calculated by the dot product between the camera's basis vectors and the image-space index of the pixel. These image-space coordinates are defined by the resolution of the image and the direction vector is then normalized to identify where a potential intersection occurs via the parametric equation of a line.

To find an intersection between the ray and a shape, the ray must triangulate its potential intersection along its parametric line with the vertices that define the bounds of each shape. This triangulation must occur for every shape in the input image to determine which shape is visible to the camera view. When an intersection occurs, a ray must be created and fired from the point of intersection to each light source in an image. If a ray between the intersection and a light source is found uninterrupted by another shape, then the shape's color is applied to the pixel. If the path is blocked, then the pixel color darkens to simulate a shadow. For each additional light source, the color, shadow and diffuse components are normalized between the number of lights in the scene.

The recursive component of ray-tracing lies in the specular properties of a shape. A ray that intersects a shape with a non-zero specular coefficient must recursively calculate the ray defined by its reflection in order to return to the camera. Intuitively, when you look at a mirror, the initial camera ray begins at your eye, and ends at the physical position of the mirror. The reflection ray starts at the mirror & extends orthogonally to your surroundings visible through said mirror.

Since each pixel calculation is independent of one-another, the process of ray-tracing is a poster-child of SPMD calculations and provided us with an enticing opportunity to extend our prior work to a new domain of GPU acceleration.

C. OpenCL

OpenCL is a framework for parallel computing providing a baseline of operations, functions, and libraries between heterogeneous multi-processing components. OpenCL allowed us to interface with our GPU components, as well as define generic kernel code that could be run on any CPU or GPU. The kernel code is the actual code to be executed on a specific core for each processing element. OpenCL defines its own variant of the C programming language with added support for parallel computations via pre-defined data vectors, common linear algebra functions, and secure methods to ensure data integrity across every work group for each work item.

D. OpenMP As a Comparandum

In our distributed processes course, we found OpenMP to be the go-to parallel framework for most students to complete the course's homework assignments. We thought that it would make a good baseline for how our OpenCL implementation compared to what we believe to have been the course-standard. Implementation of OpenMP is omitted from the methods section and used to analyze the results.

E. Related Works

Throughout this project, a frequent reference we used was NVIDIA-engineer Sam Lepere's 2016 blog entry on ray-tracing spheres and cosine-weighted sampling[1]. The specific ray-tracing algorithm implemented by Lepere was not used, however his implementation within OpenCL provided a great baseline and interface to the OpenCL-compatible processing units on our machines. Without this guide, it would have been much more difficult for us to complete our implementation.

II. Methods

The beginning of our project is defined by our struggles in maintaining and updating old project code. The last time we peeked at our ray-tracing code was a little over a year and a half ago. The highlight of the base project code was a 200-line long main function containing the parser and camera ray initialization. To better setup our parallelization, we had to modularize the code into specific functions that could be executed more easily in parallel. Once this conversion was made, we implemented two driver functions, one for each parallel framework, to use the modularized functions in parallel.

To begin development in OpenCL, we defined a boilerplate kernel initialization similar to the one described in the related-works section. From there, every ray-tracing subroutine had to be ported into the OpenCL-specific code. This process took a considerable amount of time considering the original implementation relied heavily on the C++ Eigen library that is not supported in the OpenCL Clang variant. Thankfully, we were able to convert most vectors and operations into 'double3' OpenCL objects since the framework optimizes these exact calculations and data structures.

One by one, we ported the ray-tracing functions into the kernel file in OpenCL. One necessary part of the conversion for us to re-implement the recursive component in each pixel's reflection ray. GPU hardware via OpenCL does not permit non-deterministic stack sizes for the GPU since the memory architecture of the GPU differs significantly than that of a CPU.

Throughout the process of porting the functions to the kernel file, we had to define reduction functions for our standard C++ objects into the OpenCL-specific structs. OpenCL provides a convenient way to map onto OpenCL structs prior to passing data through the kernel. Data types proceeded by "cl_" would map identically to their OpenCL counterpart in the kernel. Each of these objects necessitated their own defined input buffer stream to ensure the entirety of data was being read without losing integrity. We gave the kernel a specific buffer for each the output image, list of shapes, list of lights, and camera since these objects had context-dependent data size.

With the buffers in place in our OpenCL driver file, we queued the buffers to write their data to each device and mapped each device parameter with its respective kernel counterpart. Once queued, we defined the global work size by dividing the number of work items (pixels) by the number of processing elements on a user-selected hardware. In the case of the CPU, we used four processing elements. For the GPU, we used twenty processing elements.

For the final step of our GPU acceleration, the driver called the imported kernel code to execute with the input data buffers and the output image was computed and returned to the output buffer less than a second. The full combination of results are listed in section 3.

Our analysis for the GPU acceleration relied on three key comparisons: (1) Speedup between standard serial execution and parallel execution on a CPU, (2) Speedup between parallel execution on a CPU in both OpenMP and OpenCL, and (3) Speedup between GPU and CPU in OpenCL. In order to analyze the parallel runtime, we used the same development environment, input image, and hardware for each combination of parameters. The input that we used across all tests is the famed "Newell Teapot" known by all in the academic graphics community. The output of our GPU-accelerated ray tracer is provided below. The run time of our tests began before the passing of data through the buffers and ceased just before the output buffer was written to the .ppm file. We averaged the runtime over five iterations for each setup.

A. Implementation Specifics

We calculated the results generated for the CPU in the next section on an Intel Core i5-4670 @ 3.40GHz. The GPU we tested on is an NVIDIA GeForce GTX 1080. Our development environment consisted of Visual Studio 2019 on Windows 10 (64bit).

III. Results

Fig. 1: The final ray traced teapot image from GPU acceleration

A. Serial Vs Parallel in OpenMP

As shown in the graph below, while the speedup was not theoretically perfect, adding more threads through the use of OpenMP lead to consistent improvements in runtime. The runtime between 1 thread in OpenMP and a serial execution without it was identical. In OpenMP, the runtime between a specified 8 threads and no declared of threads specified was identical. This indicates that 1, 4 and 8 threads shows the breadth of OpenMP and serial runtimes.

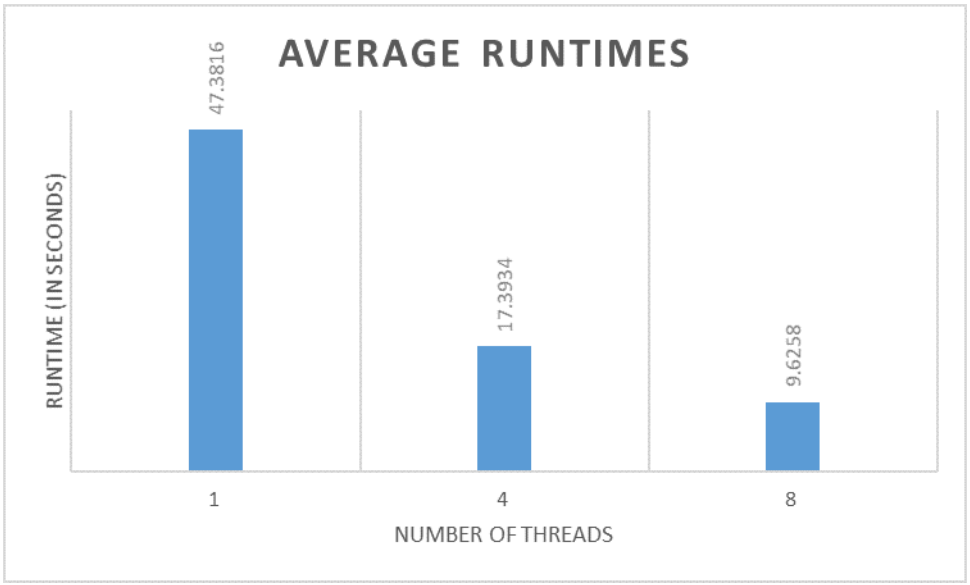

Fig. 2: Average runtimes across one or more threads using OpenMP

B. OpenMP Vs OpenCL (on CPU)

While it may seem like OpenCL on a CPU would be very similar to OpenMP code, our results show that there is a significant (almost 3x) speedup. This may be due to how OpenCL code is written, which we detailed above, as well as how bloated our C++ code was written (involving many libraries). Another possibility is the unwinding of the recursive process in the kernel code.

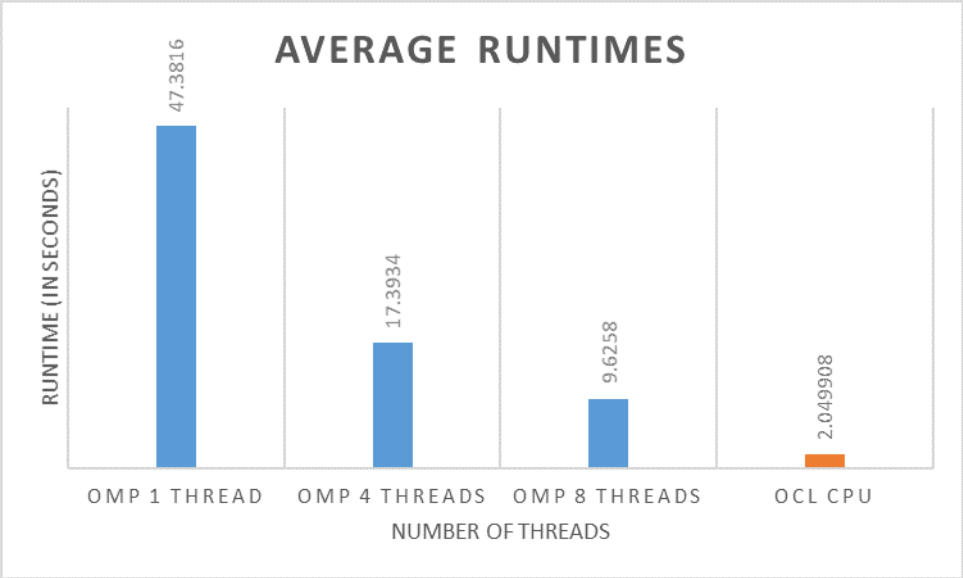

Fig. 3: Average runtimes in OpenMP compared to OpenCL used on CPU

C. CPU Vs GPU using OpenCL

Finally, when the OpenCL code is run on a GPU, the runtime increases once again, although not as much as from OpenMP to OpenCL on the CPU.

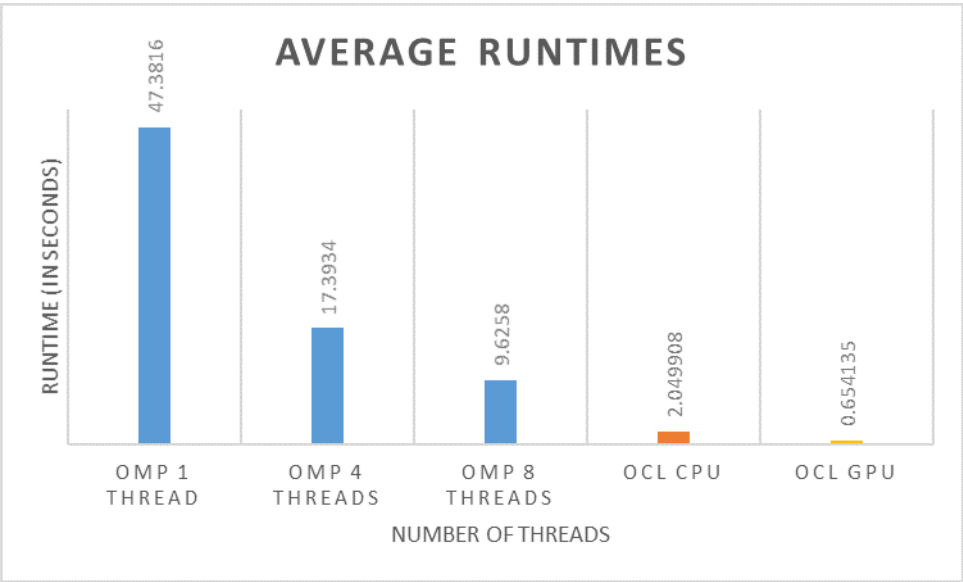

Fig. 4: Average runtimes in OpenMP compared to OpenCL (on CPU and GPU)

IV. Discussion

A. Difficulties

Diving into GPU programming appeared easier than it was. Going into this project, we feared it would be on the easy side, and prepared to stretch our implementation in other ways. However, in execution, writing the kernel code in proper form took significantly longer than expected due to the properties of the GPU's hardware. These difficulties manifested themselves in a couple different ways.

One of the premiere features of OpenCL is the programmer having the ability to pre-define their data objects before code is sent to the kernel. A slight caveat made this otherwise simple task an hours-long affair. For an object to successfully port to kernel Clang in OpenCL, the object in C++ space must be memory-aligned with the object in OpenCL kernel space. When defining the OpenCL structs in C++ space, the total size of the struct must be divisible by four bytes. The compiler assumes the developers are aware of this, and will not notify the programmer of overflows and/or loss of data integrity.

This memory-alignment problem manifested itself by setting the color of a shape, instead of its color-value, to the location of its fourth vertex. Through which, both the shape was misrepresented in the image and intersection calculations, as well as providing a color of seemingly no origin for us to debug. Since the compiler did not recognize this inconsistency, we misspent many hours debugging perfectly-fine linear algebra while convinced our data was pure. The problem was ultimately remedied by adding dummy variables to complete the size of the struct until the size-requirement was met.

A problem that perpetuated the above, was the inconsistency of debug values on the GPU. A simple print statement would repeat itself twenty times, and frequently well out of order given the asynchronous nature of operations. During these observations, we swore off value displayed from these OpenCL print statements, and used them only to ensure the GPU was operating on valid data.

Throughout the process of porting the ray-trace class code to the kernel code, we found critical mishandlings of memory allocation within loops that would frequently overflow the stack size of the GPU. Since the GPU doesn't experience the luxury of memory-centric architecture, we had to implement the same functions in a much more memory-frugal way to meet the stack size constraint of the GPU.

B. Future Work

Our project results in a very limited scope of ray-tracing on a GPU. An immediate extension could look at teapots comprised of an increasing number of shapes or reflections. We leave the question of additional rendering procedures open as well, other properties include refraction and Point-Of-View calculations to distort the image. Additionally, our access to an assortment of hardware is limited, so running our code on a multitude of machines could pose an interesting result when comparing speedup across different CPUs and GPUs. Lastly, constructing a complete ray-tracer with the full .nff functionality would complement this project very well and serve as a great foundation for visualizing graphics in motion over several scenes.

V. Conclusion

From feeble beginnings in ray tracing, waiting almost 30 minutes for the generation of the first teapot, we have made incredible leaps in this project decreasing the runtime by over seventy-two times from that of the serial implementation. Our results provide an exciting baseline for a myriad of additional graphics projects and have given us valuable experiences in coding for GPUs and other parallel processing components. It will be exciting for us to revisit this project in the future for our next parallel computation problem arises. Whenever it does, we are prepared.

References

[1] Lapere, S. (2016, November 14). OpenCL path tracing tutorial 2: path tracing spheres. Raytracey. http://raytracey.blogspot.com/2016/11/opencl-path-tracing-tutorial-2-path.html

[2] Bargteil, A. (2018, September 24). CMSC 435/634 Assignment 2: Ray Tracing II. CMSC 435/634: Introduction to Computer Graphics. https://www.csee.umbc.edu/~adamb/435/proj2.html