Abstract:
In my prior project, I developed an OpenCL path tracer from scratch. The pathtracer could render primitives and simple triangles. However, the product had several weak points. Firstly, it couldn’t render more than a very small set of triangles. Secondly, the rendering of these triangles was extremely slow in that the path tracer had to test for intersections with every single triangle in the scene. Finally, the code did not make the most of the GPU’s capabilities, with extended branching and an inability to contend with paths that were significantly shorter or longer than the other in its group.

Over the course of the semester, I made several optimizations to improve time and performance, broadly categorized into three stages. First, I used the texture memory built into my GPU to store the triangles. Secondly, I built a bounding volume hierarchy off the pbrt codebase, and adapted it for use with the GPU. Thirdly, I refactored the code to more tightly conform to the unique advantages provided by the GPU, resulting in speedups of rendering.

Porting Triangles to Texture Memory:
In the prior version of my path tracer, I stored all my triangles in local memory; this involved copying all triangles into local memory, which were then shared by the work-items in a work-group. While ensuring faster access than Global memory, Local memory is exceedingly limited.

My graphics card is the default that comes with mid-2012 Macbook Pros, the Intel HD Graphics 4000, which allocates 65536 bytes for local memory. My current triangle structure consists of 3 float4s; with each float occupying 4 bytes (Gupta), the entire structure sits at 48 bytes, leaving room for only around 1300 triangles—and that isn’t counting the extra space needed for materials, other primitives, and an acceleration structure.

Given this limitation, I ported my triangle storage to texture memory, accessible through the OpenCL interface with image objects. “Texture memory is another
variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Although texture memory was originally designed for traditional graphics applications, it can also be used quite effectively in some GPU computing applications” (Gupta).

In other words, while texture memory is optimized for image-specific localized data access, I can easily be ported for other uses. I created a one-dimensional image buffer designed to store color in RGBA, float format. Since each variable in my struct is a float4, this makes for extremely easy data access. Although I have been unable to find exact documentation on the size of the texture memory in the Intel HD 4000 card, that it is optimized for image storage indicates that it is orders larger than the local memory and I have never run out of space at any point.

**Implementation of the Bounding Volume Hierarchy Acceleration Structure:**
A bounding volume hierarchy spatially organizes triangles within a scene to reduce unnecessary intersections. I implemented the bounding volume hierarchy detailed in *Physically Based Rendering: from Theory to Implementation*, written by Matt Pharr, adapting their code for use with my GPU path tracer’s codebase.

The bvh implementation first constructs a top-down tree with pointers, and then flattens the tree into an array of nodes which can be ported to the GPU with the aforementioned texture memory.

First, I recursively built the tree using the vector of triangles lifted from the .obj file. The bvh algorithm prescribed by Pharr re-orders the array of triangles themselves, eliminating pointers and instead using indices to locate the triangles contained in each node. This works particularly well in a GPU implementation which is naturally averse to pointers; the memory makeup changes when moving from host to processor.

I split the triangles with the surface area heuristic, which, by calculating the surface area of the bounding boxes of the children, attempts to choose the optimal split plane along which to arrange the children. In the words of Matt Pharr: “The SAH model estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the
time spent on ray-primitive intersection tests for a particular partitioning of primitives.”

When the triangles are arranged in a tree, I flattened the tree of nodes into a compact array. This was done using a simple in-order traversal. These nodes are OpenCL compliant structures that are then passed to the main pathtracing kernel. As with the triangles before it, I stored the nodes in texture memory as 32 bit integers in RGBA format, which, like the triangles, aligned nicely with the structure’s layout. I separate the intersection of primitives and triangles—however, I store the t-value and the intersection information in a separate structure, so both types of intersection are ultimately processed the same in code. Of course, the code still supports a BVH-less mode, which allows for testing the BVH.

It should be said that I ran into issues in every step of the way. The complexity of the bounding volume hierarchy, and the number of steps necessary to get it properly configured on the GPU were a massive logistical nightmare. Allocating the right amount of memory for the structures and ensuring the math with all the bounding boxes was correct were among the challenges in coding this bv h. However, I have coded a kd-tree from scratch before, and was fairly familiar with spatial partitioning concepts.

As expected, the speedups were extremely significant, and allowed for the quick rendering of more complex scenes than was previously. This was evident by observation. However, while they passed the eye test, I could not substantiate that claim with data from the timer. Unfortunately, the C++ timer does not work well with OpenCL’s multithreaded, asynchronous construction. Therefore, I dispatched an empty “event” to the Pathtracing kernel and profiled it for its running time upon its return. This worked well, and I was able to get high resolution times for each kernel execution

**Teapot Scene : 1024 Triangle Faces**
The average time for 10 iterations (per iteration). Of course this is discounting the constant time of writing the image to the UI, which actually takes a good amount of time in each pass.
With BVH: 23.6619 milliseconds.
Without BVH: 214.825 milliseconds.
Which marks a **9.0789 times** speedup.
Refactoring the Code for Increased Optimizations:
Towards the end of the project, I wanted to make sure that Path Tracer was making the most of the GPU’s capabilities. I learned from a presentation by Takahiro Harada called “Introduction to Monte Carlo Ray Tracing: OpenCL Implementation,” which was made for the Computer Entertainment Developers Conference.

The paper espouses the idea of “splitting computation into multiple kernels.” This way, we can minimize branching—all similar computation should be executing concurrently. The way I had it, with all the path tracing logic was jammed into a single kernel, meant that branching would often happen in the paths of all different length and the array of different. The logic should be split into separate kernels to minimize branching.

I made a second mode which I call “Iterative Rendering,” because it processes by the per-bounce, rather than the per-path. This ensures that even when paths terminate, a new one starts. A major difference between this and the prior version is that state must be stored in between each bounce. For that reason, I had to add functionality storing the current weight, color, bounce, path number, direction, and origin point.

Each bounce in the teapot scene takes, on average, 8.83692 milliseconds. Because the teapot scene is pretty simple, most paths should conclude in one or two bounces, which would make this faster than the 24 milliseconds for a single path in the unified kernel. Meanwhile, the reconstruct times are typically less than a tenth of a millisecond—really negligible. However, it’s hard to distinguish a difference between the two renders with the eye test. This would be the starting point for future work, discerning the difference between the iterative approach and the all-in-one kernel approach.

OBJ Files
http://people.sc.fsu.edu/~jburkardt/
https://qt-am-model-tool.googlecode.com/files/monkey.obj

Acknowledgements:
Thank you to Professor Rushmeier for advising me (again) on this graphics project. It’s been a ton of fun.
Some Results:

*Monkey In Green*
The Glass Gourd
The Teapot Scene
The Lonely Glass Man
Works Cited