1 Introduction

Ray traced scenes generally appear more realistic compared to other rendering techniques. The problem is that even on very fast hardware by today's standards, the performance of ray tracing engines lags significantly behind rasterization. The performance gap has until recently been magnified by the ubiquity of dedicated acceleration hardware for polygon based rasterization, and the lack of any such hardware for ray tracing.

This project explores the applicability of recent programmable graphics processing units to ray tracing. The goal is to gain an understanding of the components and techniques necessary to implement a GPU accelerated ray tracer, and to identify algorithmic and computational optimizations that can ultimately lead to an implementation with interactive frame rates while preserving those characteristics that make ray traced images compelling. An implementation of a ray tracer that runs entirely on the GPU using OpenCL is presented in the following sections, along with comparisons with similar CPU-based implementations.

2 Background

Parker et al. [1999] achieved interactive rates for relatively simple static scenes at 512x512 pixels by distributing the work among multiple (up to 64) processors on a single system. Shortly thereafter, developments in graphics acceleration hardware made it feasible to implement a ray tracer that took advantage of some of the hardware previously dedicated to raster graphics. Early attempts produced disappointing results that were not competitive with CPU based implementations. As GPUs continued to grow in flexibility and programmability, many algorithmic optimizations were adapted for efficient use on the graphics hardware, bringing the performance to the same level as the fastest CPU based approaches [Foley and Sugerman 2005], [Horn et al. 2007], [Zhou et al. 2008], [Gunther et al. 2007], and many others. At the same time it was also becoming possible to represent more complex scenes on graphics hardware (given memory size and access constraints), improving the quality of GPU ray traced images.

3 Approach

Ray tracing is well suited to the data-parallel model of modern GPU architectures. There is a strong likelihood that ray tracing performance will improve faster on graphics hardware than on the CPU in the near future, given recent trends in GPU core counts and CPU clock rates. Most recent GPU based implementations have used NVidia's CUDA, or ATI's Stream SDKs. OpenCL promises to support the major of the features of these SDKs, while allowing programs to be portable between different OpenCL implementations on different platforms. OpenCL was used as the development platform for this reason.

In order to perform efficiently at the tasks they were designed for, GPU architectures impose certain constraints on programs written for them. Several of these had a significant influence on the selection of a ray tracing algorithm. In particular, due to the limited amounts of fast on-chip local memory, relatively slow access to global (off-chip) memory, and a lack of function recursion (there is no call stack), distribution ray tracing would be difficult at best. Whitted ray tracing does not produce realistic looking images by today's standards, so Path Tracing is used. The only major drawback to path tracing on the GPU is the quick degradation of ray coherence after the first bounce. Otherwise, it maps very well to the GPU model, the ray state consists of only one ray per active path so it can easily be implemented with no recursion.

The path tracer was first developed in Java to gain sufficient understanding of the algorithm and to provide a benchmark for rendering quality and performance comparisons without the constraints of the GPU model. All of the path tracing capabilities of the Java implementation were ported to OpenCL, optimizing for that platform as much as possible.

4 Implementation

The path tracer application consists of the path tracer implementation in OpenCL C code, and a small amount of C++ code to manage ray-tracing settings, the OpenCL environment and to draw the ray-traced image to the screen.

4.1 Task Decomposition

GPUs can be thought of as collection of 16 or 32 wide SIMD processors, where each thread running in a group executes the same instruction on thread-specific data. The reality is somewhat more complex than this, but there is effectively one instruction stream for 32 threads. This makes it desirable to decompose the path tracing in a manner that minimizes divergent instruction branching within a group since this introduces multiple instruction streams preventing full use of the SIMD processor. The other driving factor for finding the best decomposition is memory access. There are a variety of rules regarding optimal global and local memory access patterns for the threads in a group. For example, reading data from global memory can be 16 times faster if the access is aligned to a 64 byte boundary, the region is contiguous, and each thread in the group reads 4 bytes at a time.

For these and other reasons, the path tracer is decomposed into threads that each process a single pixel. Threads are grouped into 32x6 pixel blocks. The multiple of 16 makes it relatively easy to align per-thread data (such as random number seeds) to a 64 byte boundary. The 6 was chosen because even though NVidia hardware supports up to 512 threads in a group, the path tracer kernel requires too many registers to execute that many (threads in a group share a register file). It is often beneficial to have the largest work-group size possible because the hardware can hide certain memory access latencies by switching to other threads in a group after data has been requested.

4.2 Path Tracing

The OpenCL path tracer is based on the seminal work by Kajiya [1986]. The basic procedure is as follows:

```
for each sample
   ray := generate eye ray
   while path not absorbed
      find intersection Ri of ray with scene
      compute contribution of direct illumination of Ri to sample
      compute next ray in path by sampling BxDF at Ri
```

Path tracing, being a Monte Carlo method, requires a source of uniform random numbers.

4.3 Materials

OpenCL does not support function pointers, which poses a significant challenge for designing an extensible material model. This implementation permits probabilistic combinations of 4 parametric material types:

- Lambert diffuse reflectance (reflection color)
- Modified Phong specular reflection (exponent parameter to control shininess)
- Emission (emission color/intensity)
- Refractive material based on Snell's law with extinction based on Beer's law.

It may be beneficial to investigate a more general material model (for example spherical harmonic or wavelet based BRDF representations) in the future.

The contribution of direct illumination is only sampled for diffuse materials. Specular materials have either a very strong response to illumination when irradiated from the direction of mirror reflection of the sample ray, or a very low response when irradiated from any other direction. All materials are importance sampled to determine the reflection/transmission direction, which causes bounce rays to tend toward light sources at the appropriate locations on specular surfaces. Sampling direct illumination for the specular surfaces would provide little benefit for this reason. Diffuse surfaces, on the other hand, have a uniform BRDF, and the bounce rays are relatively unlikely to hit a light source without an extremely large number of samples. Sampling the lights directly drastically reduces the number of samples needed.

A modulation color is maintained to track how much light reaches the image plane for each path. After each bounce ray is calculated, this color is scaled by the BxDF of the intersection point, given the original sample direction, and the bounce direction.

4.4 Scene Model

For simplicity, the scene consists of spheres inside a fixed-size diffuse cube. The scene is configured in C++ code (which can easily be modified, but there is no user interface to modify the scene content), which allows specification of any number of spheres, and their associated material and positional properties. Spheres also double as light sources, by defining a non-zero probability of emission, and an emission color.

More sophisticated geometry can be implemented, however without function pointers to facilitate polymorphic intersection testing, this would require conditional logic in the intersection testing that could significantly hurt performance. The best approach may be to use a more generic geometric primitive such as a triangle mesh, however this would likely need a spatial partitioning data structure to manage the large number of primitives required to represent complex shapes..

4.5 Application

The OpenCL path tracer consists of a single kernel function (with several supporting auxiliary functions and data structures), with parameters specifying the scene content, camera position, image size, and an output buffer for the pixel data.

This function is invoked though the C++ host application. The application provides a simple API for managing the ray tracer settings, scene content, and camera position. It also sets up the OpenCL environment, causing compilation of the kernel function for the device it will run on, and binds parameters for kernel invocation. Using GLUT, the application creates an OpenGL context and a Pixel Buffer Object. OpenCL provides a CL/GL compatibility API, which allows a data buffer to be shared between the two libraries. Where supported, the application uses this to avoid transferring the image data over the system's PCI bus twice per frame (once to read from OpenCL, once to write to OpenGL).

The application also manages progressive refinement of the ray traced image. For prompt visual

feedback regarding the rendering status, and to avoid locking up the system (see section A.C), only a few (4 by default) paths are traced for each pixel between each redraw to the screen. This slows the overall process, and requires pixel data to be stored in a 32bit/component buffer to avoid truncation/rounding problems, but provides a better experience. Ideally the ray tracer would converge to a high quality image fast enough so this was not necessary.

5 Results

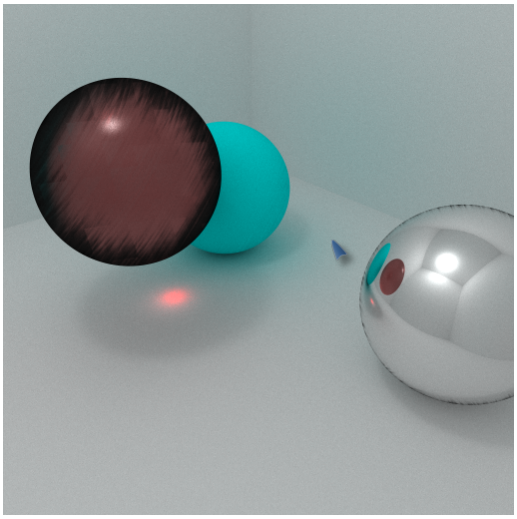

Illustration 1: Scene Path-Traced on an NVidia QuadroFX 5800

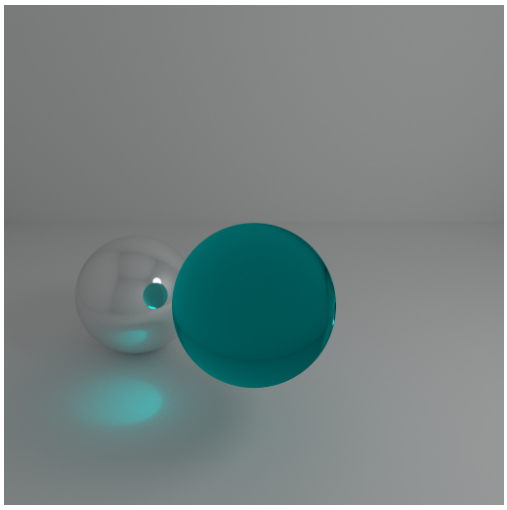

Illustration 2: Scene Path-Traced with Java based implementation.

As shown in Illustration 1, the path tracer produces images similar in quality to the Java based renderer (Illustration 2) excepting one problem; there is a small bug in the OpenCL path tracer that manifests as dark streaks near the edges of the specular and refractive surfaces on the primary eye ray hit. This is clear in Illustration 1, and you can see that the problem does not appear in the reflection of the red sphere off of the specular surfaces.

The GPU rendered image in Illustration 1 took about 1 minute, while the Java based implementation took longer than 2 hours for a scene with similar complexity. The Java code has been heavily optimized (being a byte-compiled language means that many low-level optimizations are not controllable by the programmer), while there is still significant room for improvement in the OpenCL version. The OpenCL version's performance was also reduced by using progressive refinement (Illustration 3, Illustration 4, Illustration 5) rendering, rather than rendering the final image in one pass as the Java version did.

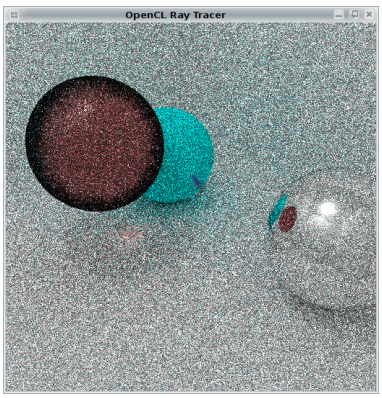

Illustration 3: Progressive Refinement - 1 Pass

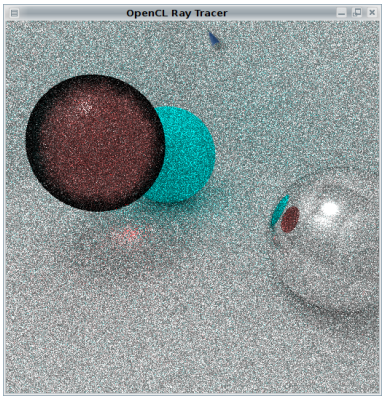

Illustration 4: Progressive Refinement - 2 Passes

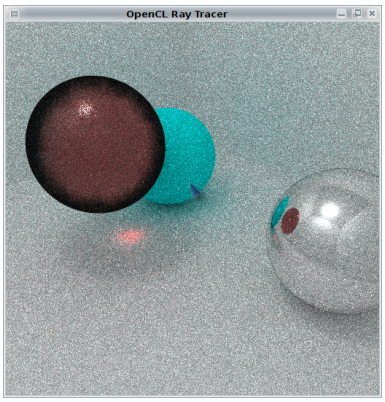

Illustration 5: Progressive Refinement - 10 Passes

The OpenCL ray tracer was also tested on a dual-core 1.8GHz 64-bit CPU using ATI's Stream SDK. Each refinement pass typically took 30-40 times longer than on the 240-core 1GHz QuadroFX 5800. ATI's Stream SDK utilizes the 4-wide SIMD floating point instructions where possible, while each of the 240 GPU cores are scalar processors. The GPU also has greater memory bandwidth, lower (effective) access latencies, and other features that improve performance beyond just the number of processing units.

6 Conclusion

A lot more work is necessary to achieve interactive or real-time ray tracing in this implementation, however the performance improvements gained by executing this on the GPU are promising. There are many known improvements that can be made to the path tracer that can increase performance further, by orders of magnitude. Modern GPU architectures still present some challenges that make a general ray tracer implementation difficult, but these are not insurmountable, and the benefits are worth the extra effort.

7 Future Enhancements

There has been a lot of work in the area of GPU based ray tracing over the past decade. Many of the algorithmic, data structure, and low-level enhancements can improve performance by orders of magnitude, by:

- Reducing the number of intersection tests (spatial partitioning, packet tracing).
- Reducing the number of samples necessary to produce a good image (photon mapping, bi-directional path tracing)
- Improving memory access coherence (data organization, packet tracing, decomposing the OpenCL kerenel differently.

Other interesting possibilities include evaluating the Metropolis Light Transport (MLT) algorithm for GPU suitability, and hybrid rasterization/ray tracing. MLT generally produces good image results much faster than traditional path tracing, but to my knowledge has not been ported to the GPU. Rasterization can be used to generate the initial eye-ray intersections (e.g. with OpenGL), and can then pass this information to OpenCL for further tracing. This also provides some flexibility in balancing

performance vs. quality, allowing mixed ray-traced and rasterized entities in a single scene.

References

Foley, Tim, and Jeremy Sugerman. 2005. KD-tree acceleration structures for a GPU raytracer. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS '05*, no. July: 15.

Gunther, J., Popov, S., Seidel, H., & Slusallek, P. (2007). Realtime Ray Tracing on GPU with BVH-based Packet Traversal. *2007 IEEE Symposium on Interactive Ray Tracing*, 113-118.

Horn, Daniel Reiter, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. 2007. Interactive k-d tree GPU raytracing. *Proceedings of the 2007 symposium on Interactive 3D graphics and games - I3D '07*: 167.

Kajiya, J. (1986). The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (p. 150). ACM. Retrieved from http://portal.acm.org/citation.cfm?id=15922.15902.

Lafortune, E., & Leuven, K. U. (n.d.). Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering.

NVIDIA OpenCL Best Practices Guide Version 1.0. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf (accessed January 12, 2010)

Parker, S., Shirley, P., Livnat, Y., Hansen, C., & Sloan, P. (1999). Interactive Ray Tracing. *In Proceedings of Interactive 3D Graphics*, *pages*, 119-126.

Shirley, P., Wang, C., & Zimmerman, K. (1996). Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, *15*(1), 1-36.

Veach, E., & Guibas, L. J. (1997). Metropolis light transport. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*, 65-76. New York, New York, USA: ACM Press.

Wald, I., Ize, T., Kensler, A., Knoll, A., & Parker, S. G. (2006). Ray tracing animated scenes using coherent grid traversal. *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06*, *1*(212), 485. New York, New York, USA: ACM Press.

Wang, R., Zhou, K., Pan, M., & Bao, H. (2009). An efficient GPU-based approach for interactive global illumination. *ACM Transactions on Graphics*, *28*(3), 1.

Zhou, K., Hou, Q., Wang, R., Guo, B., & Asia, M. R. (2008). Real-Time KD-Tree Construction on Graphics Hardware. *Construction*, *27*(5).

A Application Information

A.A System/Platform Requirements

Architecture: x86_64, x86 (binaries included for x86_64)

Platform: Linux (tested on Debian unstable and Ubuntu Karmic Koala)

Libraries:

OpenCL: A driver supporting OpenCL 1.0. is required for the path tracer. There are implementations supporting recent NVidia and ATI graphics hardware, as well as any x86 CPU supporting SIMD instructions. The application was tested using NVidia's 195.30 linux beta drivers on a Quadro FX5800 card, as well as ATI's Stream SDK 2.01 on a dual-core AMD Turion CPU. The development header files are required to build the application from source, and should be installed on the standard system include path, or in the include/ directory of the ray-tracer sources.

OpenGL: OpenGL 2.1 is required to draw the ray traced images. Your driver must also support 32-bits per component pixel buffers. Development headers are necessary to compile the application from source. Tested with NVidia 195.30 drivers and open source radeon driver included with X.Org 7.5.

GLUT: Glut is used to manage the OpenGL environment, and the render loop. Tested with freeglut3.

A.B Compiling

The included *Makefile* at the top level of the source tree can be used to compile the application on Linux. Run *make all* to compile the application into the *bin/* subdirectory, *make doc* to build the documentation (requires doxygen) into the *doc/* directory and *make clean* to remove any compiled object files. This has been tested using GCC/G++ version 4.4.1 and GNU Make 3.81.

A.C Running

Run the *bin/ray-tracer* application included with the distribution to launch the application. The application will look in the working directory for a *raytracer.cl* (which can be found in the bin/ directory as well) that contains the OpenCL source code. Upon successful launch, you will see a window containing a 512x512 pixel path traced image. The image will continually refine until it reaches a (large) maximum number of iterations. The scene is statically configured in C++ code (main.cpp), and can not be manipulated via the user interface.

Warning!

The application will run the OpenCL kernel on the first device it finds (preferring GPU over CPU if both are present and supported by your OpenCL driver). If this happens to be the GPU that is driving your display, you may experience apparent lock-ups. This is due to the fact that the GPU can not be preemptively scheduled. While the ray tracer is actively running the GPU can not work on updating your display. The default pixel sample rate per pass is small enough so that your system will not become completely unusable.

A.D Porting to other platforms

The entire path tracer implementation is done in cross platform OpenCL, which is automatically compiled from source for your platform at runtime. There is a relatively small amount of C++ code used to manage the various configuration settings, invoke the OpenCL drivers, and draw the ray traced images. This section of the source code is not cross-platform, but it uses standard libraries that should be available on any OpenCL-supporting platform. Porting the C++ code should only require changing some system includes, and these are marked as Linux-specific in the source code.

B Application Source Code

Included in raytracer.cl-krisher.tar.gz.