6.375 Ray Tracing Hardware Accelerator

Chun Fai Cheung, Sabrina Neuman, Michael Poon

May 13, 2010

Abstract

This report describes the design and implementation of a hardware accelerator for software ray tracing using Bluespec System Verilog compiled onto an FPGA. The hardware accelerator is substituted for the intersection test function in a ray tracing software program called POV-Ray. Sce-Mi, written in C++, is used to interface the POV-Ray software with the FPGA hardware accelerator. Scenes with multiple shapes and multiple types of shapes were successfully rendered, and performance estimates from preliminary testing results indicated that the hardware accelerator could indeed accomplish its task faster than the purely software implementation.

1 Background

In real life, rays of light start at a light source and refract or reflect off of objects in their path. Some of these rays eventually terminate at the eyes of a viewer, whose image of the scene is assembled from those rays. Recreating this phenomena to render a virtual image is very inefficient, because many ray paths do not return to the viewer at all. The much more efficient solution is a backwards approach, known as ray tracing.

Ray tracing is a technique in image rendering where the path of individual rays of light are traced, starting from the viewer, to each pixel in the scene, refracting or reflecting off of objects in the image and terminating at the light source. Although this yields very high quality photorealistic images, ray tracing is not currently in widespread use for real-time rendering applications such as modeling software and video games because it is computationally intensive when written in software, and frames will not render quickly enough at high resolutions.

2 Project Objective

Our goal is to implement a hardware accelerator on an FPGA to work in conjunction with ray tracing software, to speed up the computation necessary for ray traced rendering of images. We use an open source software ray tracing program, called POV-Ray, and an FPGA to implement our hardware accelerator for the software.

There are several benefits to using an FPGA for this application. First, the ease-of-use that working with an FPGA affords makes hardware design, development, and prototyping fast and low-risk. Second, the FPGA's performance is a conservative performance indicator when compared to custom hardware. A hardware accelerator design implemented on an FPGA that meets the necessary timing specifications for high performance ray tracing procedures will likely demonstrate significantly more impressive performance when implemented in custom hardware.

We want a single hardware accelerator to alleviate the computational burden of the software ray tracing algorithm, and successfully interface it with the software so that images render correctly. We chose to accelerate ray-shape intersections because they are the bulk of computations performed and because they lend themselves better to be implemented in hardware compared with another computationally expensive operation, bounding box calculations.

Once we had a basic implementation of our main goal, we made several refinements to our hardware accelerator for higher performance. One performance refinement was to pipeline the ray intersection detection test module, to achieve a higher clock speed and greater throughput. Another performance refinement was to add an additional hardware module that loaded the scene objects to be rendered onto the FPGA. By doing this, we save the communication burden of having the software constantly sending all of the objects as well as all of the rays to be tested to the hardware. Now instead, the objects in a scene are sent to the FPGA only once, where they are loaded into memory storage. The objects are then iterated through on the hardware end, and the software is only responsible for sending rays to be tested to the hardware.

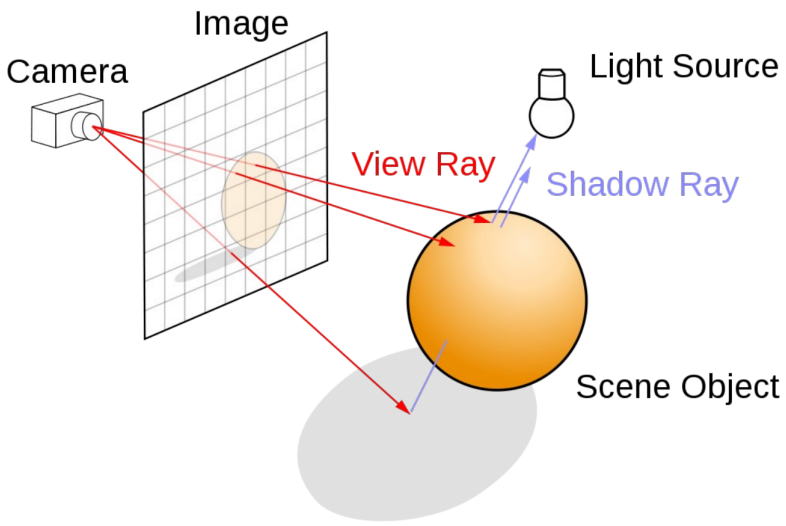

Image Source: http://upload.wikimedia.org/wikipedia/commons/thumb/8/83/Ray_trace_diagram.svg/1000px-Ray_trace_diagram.svg.png

Figure 1: The ray tracing algorithm builds an image by extending rays into a scene

3 High Level Hardware Design

Our basic hardware accelerator is an implementation of intersection testing. The intersection hardware accelerator takes as input rays and objects provided by the software, and returns intersection results, including the point of intersection and the depth along the ray where the point is found. In cases where a single ray intersects multiple objects, the intersection for the object with the minimum depth along the ray is returned.

The high level design of our intersection hardware accelerator is two main blocks– one that performs intersection tests between rays and objects in a scene, and one that iterates through the objects to be tested for intersections. The object iteration block loads all of the objects in a scene from the software once at the beginning of rendering, and then iterates through the stored objects and feeds them to the intersect test block for testing. This block also receives the intersection data output from the intersect test block, and keeps track of the current minimum depth intersection for every ray, in order to return that as a result to the software. The intersect test block receives the rays and objects to test as input, and then checks for intersections. It outputs whether there was an intersection or not, the calculated point of intersection, the depth along the ray that the intersection occurs, and a tag signifying what type of shape was intersected.

4 Microarchitecture Description

The hardware accelerator has three main parts: the software-hardware interface layers, the shape loading and iteration hardware block, and the hardware intersect test block. The following sections are an in depth look into the designs of each of these parts.

4.1 Software-Hardware Interface Layers

The software-hardware interface consists of three distinct parts, the POV-Ray software, a testbench layer and the Sce-Mi bridge. The POV-Ray software's ray-shape intersection function was modified to off-load the ray-shape intersection tests to our hardware accelerator through pipes. A request pipe is used to send the ray-shape intersection test request messages and shape messages to the C++ testbench and a response pipe is used to receive the ray-shape intersection result message from the C++ testbench.

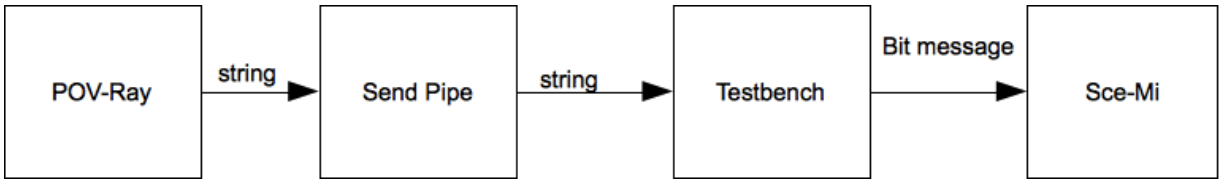

Figure 2: The request flow from POV-Ray to Sce-Mi bridge

The C++ testbench acts as a translation layer that converts the message data sent from POV-Ray into data bits understood by the Sce-Mi bridge. The testbench is able to handle two types of messages, shape messages and ray messages. The shape messages are used to load the objects into the FPGA, and the ray messages are used to request the intersection result for the ray provided in the ray message. When appropriate, the message data parsed from the POV-Ray request pipe is converted into floating point numeric types through the use of Boost's `lexical_cast` function. The translation layer converts the floating point data into fixed point and then bit packs into bits that can be sent over the Sce-Mi bridge. If the message is a shape message, the bit message would be sent to the shape port, or if it was a ray message, it would be sent to the request port.

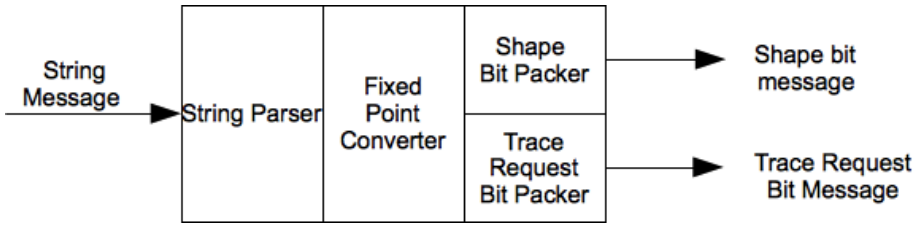

Figure 3: Detailed request flow from the testbench code

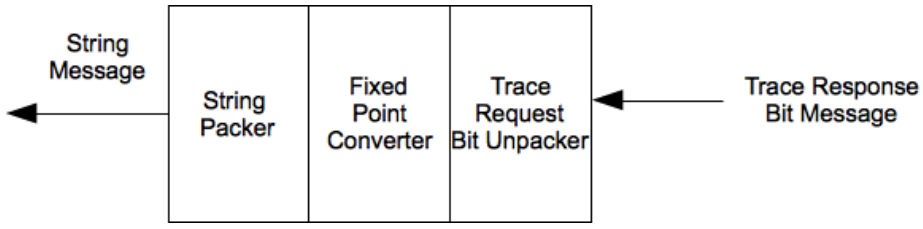

Figure 4: Detailed response flow from the testbench code

After sending a request to the FPGA, the testbench blocks and waits until a response is received. When the bit response from the Sce-Mi bridge arrives, the response is converted into C++ string types and numeric type data is converted from bits to fixed point and then converted to strings to be sent back the POV-Ray through the response pipe.

When POV-Ray receives the intersection result from the response pipe, the appropriate shape intersection result object is constructed and returned from the `FindIntersection` function. The sphere and the plane intersection result objects are supported in our design. If additional shapes are to be added, the POV-Ray intersection result objects must be created for these before POV-Ray can properly render an image. These intersection result objects must be used because in addition to the intersection coordinates and depth, they also contain important flags that POV-Ray depends upon for other stages of the rendering process.

4.2 Intersection Accelerator Block

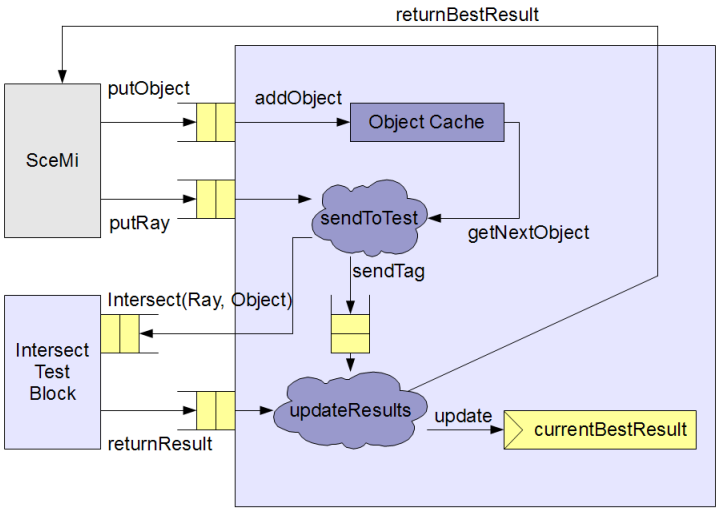

Figure 5: Block diagram of the intersection accelerator

The intersection accelerator block directly interacts with Sce-Mi by loading shapes and then iterating through them. The function of this block is to receive all objects in a scene, and take all requests for ray-shape intersection testing. The block delegates the testing to the intersect test block and returns an intersection result back to the software through Sce-Mi.

First, SceMi must send all objects in the scene and the block loads them all into an object cache. This cache stores all of the objects in a register file (currently it stores up to 256 objects, but this is flexible). Once all of the objects are loaded into the cache, the block can call on the cache to send an object. Every time the block calls `getNextObject`, the cache internally updates an index, so that next time `getNextObject` is called, a new object is returned. The object cache also has a `reset` method, where the internal index is reset so that `getNextObject` will return the first object in the cache.

Once the object cache is fully initialized, the Sce-Mi can now send a ray to the block for testing. The block receives this ray, calls `getNextObject` and sends both to the intersect test block. Since the intersect test block is multicycle, the intersection accelerator block enqueues the object's ID in a FIFO, to be later retrieved when the intersect test block returns a result. This allows the intersection accelerator block to decouple its stages and achieve some more performance from pipelining.

When the intersect test block returns a result, it is matched up with its ID and compared against a register state that keeps a copy of the current best result (defined as the intersection result that has the least amount of depth). If the result is better than the best result, the register is updated with the new one. Once we've exhausted all of the objects to check from the object cache, we return the best result back to Sce-Mi.

4.3 Intersect Test Block

The intersect test block takes as input a ray and an object, and returns intersection data including whether there was an intersection or not, the coordinates of the intersection point, and the depth along the ray that the intersection occurs at. Our implementation of the intersect test block supports two object types, planes and spheres, but is highly modularized such that it can be easily expanded to support many more shape types. The geometric intersection test algorithms used for planes and spheres are adapted from pseudo-code algorithms described in Chapter 5.3 of Ericson's Real-Time Collision Detection [1].

For the ray-plane test, the input information is the ray origin point and unit direction vector, and the plane unit normal vector and scalar distance from the origin.

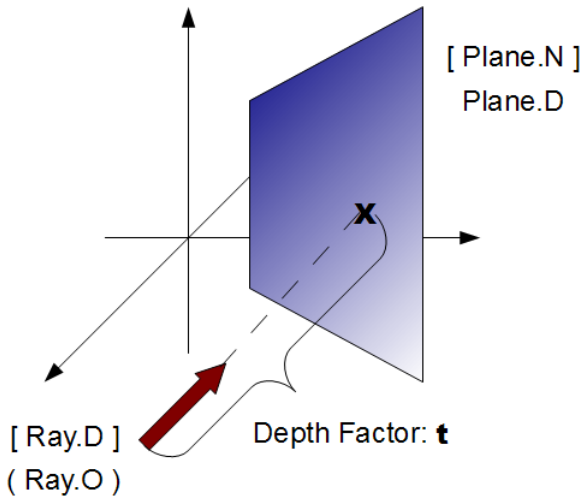

Figure 6: Ray and plane

Given this information, the ray-plane intersection can be tested with the following algorithm:

```
Input:   Ray origin = Ray.O
         Ray unit direction vector = Ray.D

         Plane unit normal vector = Plane.N
         Plane distance from origin = Plane.D

Output:  t = (Plane.D − Plane.N • Ray.O) / (Plane.N • Ray.D)
         if t < 0:
              return MISS
         else:
              pt = Ray.O + t(Ray.D)
              return HIT, t, pt
```

A block diagram of the algorithm, emphasizing the computational blocks necessary, is shown below:

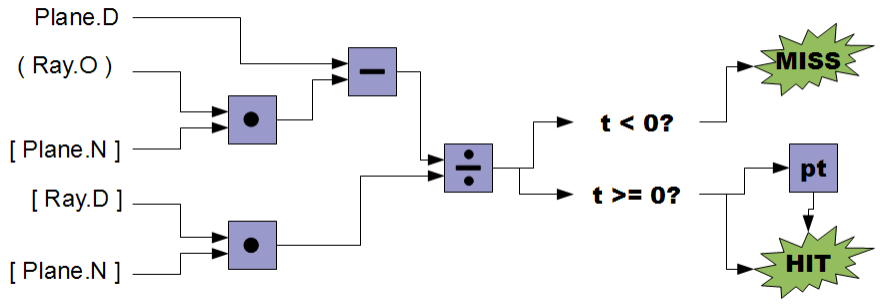

Figure 7: Block diagram of ray-plane test algorithm

For the ray-sphere test, the input information is the ray origin point and unit direction vector, and the sphere center point and scalar radius.

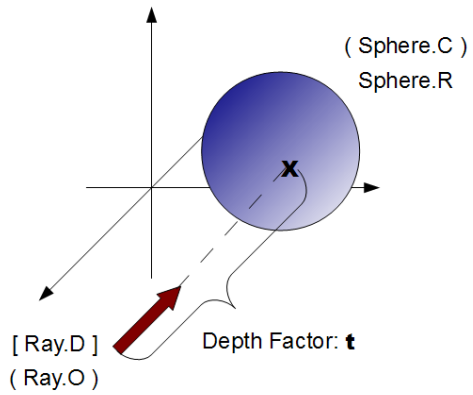

Figure 8: Ray and Sphere

Given this information, the ray-sphere intersection can be tested with the following algorithm:

```
Input:   Ray origin = Ray.O
         Ray unit direction vector = Ray.D

         Sphere center = Sphere.C
         Sphere radius = Sphere.R

Output:  m = Ray.O - Sphere.C
         b = m • Ray.D
         c = (m • m)(Sphere.R • Sphere.R)
         d = b^2 - c
         if (c > 0 && b > 0) || d < 0:
              return MISS
         else:
              t = -b - √d
              pt = Ray.O + t(Ray.D)
              return HIT, t, pt
```

A block diagram of the algorithm, emphasizing the computational blocks necessary, is shown below:

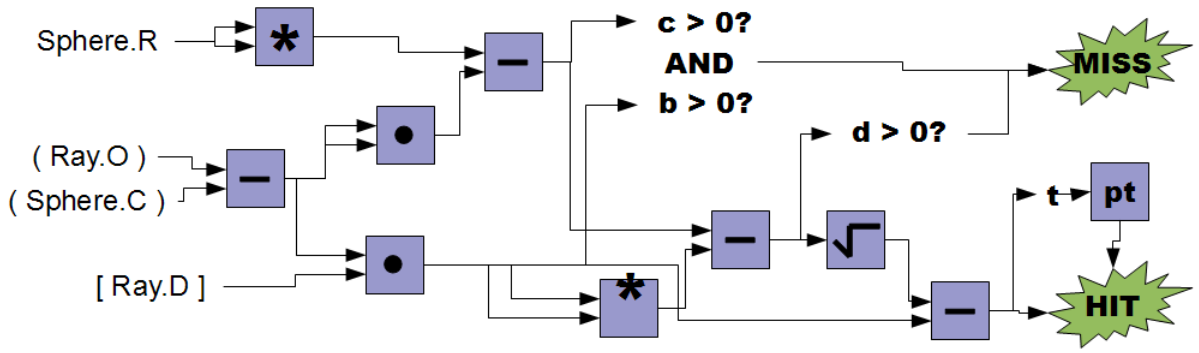

Figure 9: Block Diagram of Ray-Sphere Test Algorithm

To implement these test algorithms in hardware, a five-stage pipeline was created, with stages separated by multi-cycle computational blocks, such as dot products, dividers, and square root calculations. The pipeline has two main paths through it, one for the ray-plane algorithm and one for the ray-sphere algorithm. These paths branch in two when it is determined whether the test will return a hit or a miss. The possible paths for a single ray-object test to travel through the pipeline are pictured below.

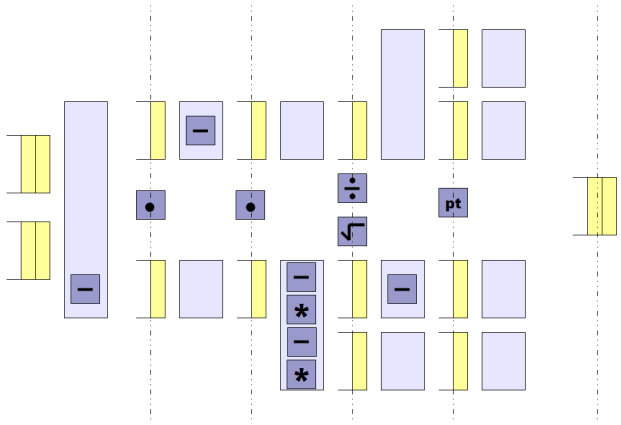

Figure 10: Five stage intersect test pipeline

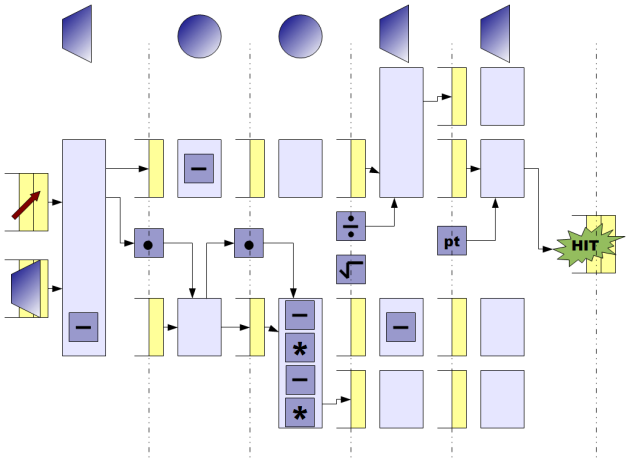

Figure 11: Intersect test pipeline with multiple simulatenous ray-object tests

Because the pipeline stages are isolated from one another by FIFO buffers, five different tests can be progressing through the pipeline at the same time, one inhabiting each stage. These tests can be for either object. The critical path through the intersect test block (and for the entire hardware accelerator) is currently in the ray-sphere path, in the third stage of the pipeline, where there are a number of computationally intensive operations that all need to happen. Splitting that third stage into two pipeline stages would be wise, if further work on this project were to be done.

5 Implementation Evaluation and Performance

The design was synthesized on a Xilinx Virtex 5 FPGA. Three distinct implementations were synthesized. The extensive use of dot products in our algorithms required the careful design of the architecture to effectively use the limited multipliers on the FPGA. Our implementation focused on increasing frequency performance while remaining below the multiplier resource limitation. The original naïve functionally correct implementation would not synthesize on the FPGA because all 64 multipliers were instantiated and the place and route engine could not achieve timing closure with these resources being exhausted. The second implementation was re-architected to re-use more of the multiplier resources by sharing multipliers between the shape tests since only one shape test was active at one time. This implementation was reported by the synthesis report to operate at a maximum frequency of 60MHz and used 32 multipliers. The third implementation introduced a deeper pipeline and the ability to handle two distinct shape intersection tests in parallel and this was successfully synthesized operated at 64MHz and used 44 multipliers.

The Intel Pentium D 2.8GHz on the `cs02.mit.edu` machine using POV-Ray to render a 320x240 scene with a plane and a sphere required 0.149 seconds to perform 280,916 intersection tests, which evaluates to an average of 525 ns per intersection test spent in hardware. The same scene was calculated to have spent 363 ns per intersection test in the FPGA. The cycles per intersection test in simulation was multiplied with the maximum frequency reported by the synthesis results to obtain this number.

6 Design Exploration

Although we've successfully implemented an accelerator and hooked it into a widely used software ray tracer, many more options exist to expand upon this work. A few considerations limited our performance: Sce-Mi and fixed point arithmetic. Ideally, an FPGA can use the full bandwidth offered by its I/O bus, but by using Sce-Mi (since we did not have enough time to implement a new, more lean interface) we lowered our PCIe bus performance to under 5% of it's maximum bandwidth capacity. Given more time to develop quicker interfacing between the test bench and the FPGA, we could see huge I/O performance gains. Secondly, we were limited by Bluespec's lack of floating point arithmetic. Because we neither had it readily availalbe as a Bluespec library, nor decided it would be a good idea to drop in a Verilog module after Bluespec compilation, our implementation was forced to use fixed point arithmetic for all of its calculations. This heavily limited our precision, and looking at the scenes we output, almost all of the errors generated (lack of smooth edges, dots, etc) were due to fixed point precision errors.

Further work could be done by implementing a spatial tree structure for storing the nodes, which would give a good speedup for scenes with a lot of images. Of course, more shapes can be implemented in the hardware, along with possible specialized support for advanced ray tracing features such as radiosity calculations. Another promising area of improvement is moving all of the primary ray generation onto the FPGA. This way we can minimize I/O operations to just a constant cost of sending in scene data. Once sending in the scene data, all necessary calculations are done completely on hardware, possibly leading to great performance improvements.

7 Acknowledgements

The authors gratefully acknowledge Richard Stephen Uhler and Abhinav Agarwal for their help, advice, and support in the course of this project. The authors also acknowledge Professor Arvind for his guidance, and the Bluespec Inc. for allowing use of their software package.

8 References

[1] C Erickson. *Real-time Collision Detection.* Morgan Kaufmann, 2005.

[2] A Keller, F Slomka. *Fixed Point Hardware Ray Tracing.* Universität Ulm, 2007.

[3] M Pharr, G Humphreys. *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2004.