Accelerating Quadratic-Based Ray Casting Using an FPGA

Joseph Cryer[1]

[1]Department of Computer Science, University of Bath

Abstract

Ray casting is a widely understood computer graphics technique. It is more computationally intensive than the common approach of per-primitive triangle-based rasterisation, but allows for the simulation of more complex optical effects, such as reflection, refraction, detailed shadows, and sub-surface scattering of light. These features consistently result in images that are deemed better or more aesthetically appealing. This paper aims to develop an effective hardware accelerator of ray-quadratic intersection tests for ray casting purposes. Experiments performed during this project show that the hardware design proposed, designed, and simulated achieves a 6.57x speedup compared to an equivalent Python ray-quadratic intersection tester. It is also shown that there is the capacity for significantly improved performance, given further work on data transfer techniques.

Acknowledgements

Throughout my research project I have received a great deal of support. A massive thank you to my project supervisor, Ken Cameron. Being able to work off of your dissertation project was fascinating, and I never would've had the opportunity to learn about and experiment with such interesting technologies without your ideas.

Introduction

This project takes inspiration from Ken Cameron's 4th Year research project at the University of Edinburgh, entitled "Acceleration of CSG Using a Custom VSLI Device". It proposes, designs, and simulates a novel integrated circuit for the sole purpose of "accelerating the creation of images in a Computer-Aided Design (CAD) system based on Constructive Solid Geometry" [1].

This form of ray casting acceleration is different to typical modern approaches in that it is specifically designed for use with quadratic surfaces. This allows for constructive solid geometry (henceforth referred to as CSG) to take place to form complex shapes out of combining primitives that CAD systems use, rather than forming a new polygonal representation of the shape. This CSG approach to object formation allows for higher rendering quality at the expense of computational complexity, but with a custom circuit designed to perform hardware acceleration, this may become a viable option.

In existing widely available graphics hardware, a rasterisation step takes place within the graphics pipeline that determines the pixels covered by a primitive in a given scene, in order to determine what to display. The majority of existing graphics hardware is optimised solely for use with triangle-based models, as it is an efficient shape to rasterise and calculate on. By contrast, ray casting and ray tracing works by shooting rays into the scene for each pixel that needs to be displayed, in order to find out what that pixel's colour should be. This per-pixel rather than per-primitive approach allows for the simulation of much more complex optical effects, such as reflection, refraction, more detailed shadows, and sub-surface scattering of light. This approach comes at a very high cost computationally compared to the more common approach, but consistently results in images that are deemed better or more aesthetically appealing.

With modern advances in Field-Programmable Gate Array technology (FPGAs), it has become significantly cheaper and simpler to design hardware and essentially model the performance of a software-designed chip in real-time. Various system trials and design iterations can be performed without each chip having to be sent off to be made (at great cost), meaning that a much more flexible development process can be achieved.

In this work, we aim to use an FPGA board to develop an effective accelerator for ray-quadratic intersections for ray casting purposes. The contributions of this paper are as follows:

- We design and implement a ray-quadratic intersection tester IP (Intellectual Property) Core in C++ using Vitis HLS.
- We integrate the designed IP Core into the PYNQ-Z2 subsystem using Vivado, and this is exported to a PYNQ-compatible Overlay.
- The designed Overlay is then used to accelerate ray-quadratic intersection tests within a Python

ray tracing program running on the PYNQ-Z2 FPGA board.
- We evaluate the effectiveness of the FPGA design using this Python ray tracing program.

Background

Ray Casting

The process of ray casting is as follows: rays from an 'eye' (i.e. the camera location) are sent through the 'screen' (a square pixel array) - one ray per pixel. For each ray, the closest object to the 'eye' blocking the path of that ray is found. A pixel value can then be determined using a given shading model.

One of the most common shading models to use is the Phong Shading model [2], which takes the material properties of the object being hit, the positions and properties of any lights in the scene, and the calculated normal of the object surface at the intersection point (see Figure 1 for an example). Phong shading provides a better approximation of the shading of a smooth surface compared to other methods such as Gouraud shading [3], though is more expensive as a result.

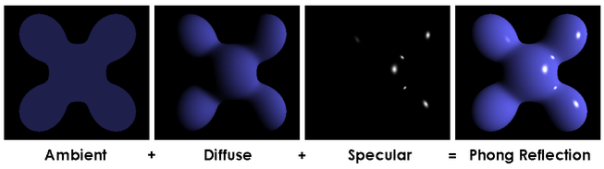

Figure 1: *Diagram showing the different components of Phong shading [4]*

Recursive ray tracing expands the simple ray casting model in order to add realism to ray traced images. Three additional kinds of rays may be shot by the ray tracing engine in certain situations. Reflection rays may be calculated if an initial ray hits a 'reflective' surface - in these situations, a new ray is shot from the intersection point, and the same process as initially described is performed again and again until either a non-reflective surface is hit, or a set maximum recursion depth is reached. At this point, the pixel value is calculated from the last object hit. Similarly, refraction rays will be calculated if an initial ray hits a refractive surface - this allows for transparent and translucent materials [5]. Finally, shadow rays can be calculated, in order to determine whether an opaque object is blocking light reaching a given point. Here, a ray is shot from the ray-object intersection point found by the initial ray towards each light in the scene, and if an object is found to be blocking the light, this is included in the lighting calculation for that pixel.

This project will be performing basic ray casting and using the Phong shading model, but will not be implementing recursive ray tracing. This is because the focus of the project is on accelerating ray-quadratic intersections rather than any other aspect of the computer graphics pipeline.

Constructive Solid Geometry & Quadratic Objects

A typical ray casting algorithm uses the triangle as its primitive object, building up complex polygons using hundreds or thousands of triangles. However, other approaches to displaying complex objects are possible. One such approach is constructive solid geometry (henceforth referred to as CSG), using quadratic objects. In this, complex objects can be constructed by combining quadratic objects with simple operations: these being Union, Difference, and Intersection (as shown in Figure 2). As such, complex shapes that would take hundreds of triangles to render can be modelled using comparatively few quadratic objects.

The ray-quadratic intersection test is more computationally intensive than the ray-triangle intersection test, and the evaluation cost of a CSG structure can be significant, however there are benefits to this approach. Firstly, the number of primitive objects in a given scene will likely be significantly lower when using quadratic objects with CSG compared to triangles, for the same level of fidelity. Secondly, as a model being rendered using quadratic objects directly, the resultant images are more true to models created in CAD software compared to exported polygonal versions of these models. Indeed, modern cutting-edge software such as Autodesk's "Arnold" has begun implementing curve primitive ray tracing in order to better simulate complex features such as hair and fur [6].

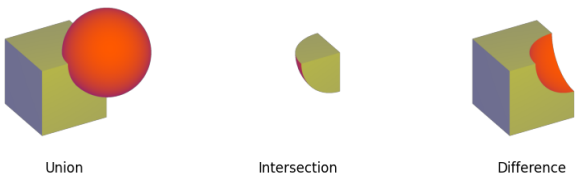

Figure 2: *Diagram showing the three CSG operations [7]*

Ray-Quadratic Intersection Test

The equation of a quadratic surface in 3D space can be defined with the following equation:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

Expanding out these matrix multiplications, we get:

$$ax^2 + 2bxy + 2cxz + 2dx + ey^2 + 2fyz + 2gy + hz^2 + 2iz + j = 0$$

We define a ray by an origin vector P and direction D, with the following equation:

$$R(t) = P + D(t)$$

This can be expanded to its three component parts:

$$x = P_x + tD_x$$

$$y = P_y + tD_y$$

$$z = P_z + tD_z$$

The intersection of the ray and the quadratic surface can be found by substituting these three ray intersections into the quadratic equation previously expanded:

$$a(P_x + tD_x)^2 + 2b(P_x + tD_x)(P_y + tD_y)$$

$$+2c(P_x + tD_x)(P_z + tD_z) + 2d(P_x + tD_x)$$

$$+e(P_y + tD_y)^2 + 2f(P_y + tD_y)(P_z + tD_z)$$

$$+2g(P_y + tD_y) + h(P_z + tD_z)^2 + 2i(P_z + tD_z) + j = 0$$

Like terms can be collected, in order to rewrite the equation as:

$$A_q t^2 + B_q t + C_q = 0$$

Where A_q, B_q, and C_q are defined as:

$$A_q = aD_x^2 + 2bD_xD_y + 2cD_xD_z + eD_y^2 + 2fD_yD_z + hD_z^2$$

$$B_q = 2(aP_xD_x + b(P_xD_y + D_xP_y) + c(P_xD_z + D_xP_z) + dD_x + eP_yD_y + f(P_yD_z + D_yP_z) + gD_y + hP_zD_z + iD_z)$$

$$C_q = aP_x^2 + 2bP_xP_y + 2cP_xP_z + 2dP_x + eP_y^2 + 2fP_yP_z + 2gP_y + hP_z^2 + 2iP_z + j$$

We can calculate the roots of the quadratic in order to find the intersection points between the ray and quadratic object:

$$t = \frac{-B_q \pm \sqrt{B_q^2 - 4A_qC_q}}{2A_q}$$

From here, we can use the smallest t value found to calculate the intersection point between the ray and quadratic, and the normal between the ray and the quadratic at that point.

PYNQ-Z2 FPGA

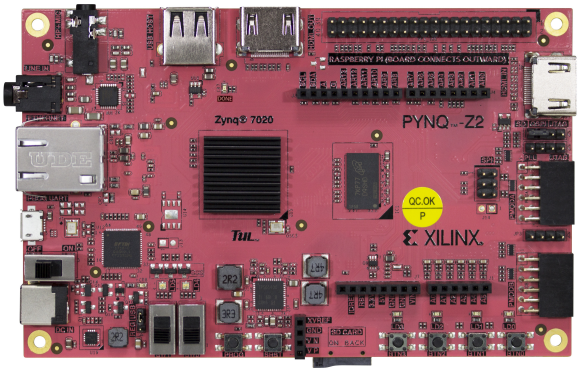

Figure 3: *Image showing the PYNQ-Z2 Development Board [8]*

An FPGA is a "Field Programmable Gate Array", which is an integrated circuit that can be reconfigured to meet specific requirements after manufacture. There are two main use-cases for FPGAs: hardware design prototyping prior to chip manufacture, and hardware acceleration to speed-up specific parts of algorithms [9]. This project aims to leverage FPGAs for both of these use-cases, as we are incrementally prototyping a hardware design on an FPGA with the end goal of accelerating ray-quadratic ray tracing.

An FPGA overlay is a programmable, reconfigurable architecture that encapsulated an underlying FPGA bitstream and provides an interface to any program that is leveraging it. It acts as a hot-swappable interface to the physical FPGA fabric, and allows for much faster development times while also reducing the complexity of the toolchain needed to work with FPGAs [10].

PYNQ is an open-source project from AMD Xilinx, designed for "rapid prototyping and development" of Zynq FPGA designs [11]. It is also intended to be used by software developers without significant knowledge of hardware design, who wish to leverage existing designs to accelerate their projects. The PYNQ framework allows for relatively simple loading of hardware overlays into the on-board FPGA chip, alongside an intuitive Python-based interface

design to allow a software developer to interact with the on-board hardware. The PYNQ-Z2 is the recommended board for getting started with PYNQ, as it is a low-cost, high feature development board. It was designed for the AMD University Program, with the intention of supporting teaching and research [8]. As such, this is the board that was purchased for this project by the University of Bath.

Vitis HLS & Vivado

There are a few different approaches to designing hardware for use on an FPGA board. The most commonly used method in industry is to design hardware in a Hardware Description Language (or HDL), such as VHDL or Verilog. However, these both require significant up-front knowledge of hardware design and optimisation, as they are very low-level. An alternative approach is to design hardware in C++ or SystemC, and then use a synthesis tool to convert this code to a lower level HDL. This still requires significant understanding of the underlying processes occurring, as well as a more complex software toolchain for development, but can allow for faster and easier prototyping by a software developer [12]. As such, we opted to use Xilinx Vitis HLS [13] in order to synthesise C++ code with HLS pragmas into a Verilog Intellectual Property (IP) block.

The Xilinx Vivado tooling was used to integrate this custom IP Block into the Zynq processing subsystem, before exporting it all as a bitstream to be loaded into the PYNQ-Z2 FPGA board. The Xilinx toolchain was decided upon due to them being the manufacturer of the Zynq-7000 FPGA chip embedded into the PYNQ-Z2 board, as well as the majority of the tutorials and documentation using these tools.

Design and Implementation

Numerical Value Representation

When designing the ray-quadratic intersection tester, one of the first design key decisions was the way that numbers should be represented throughout the ray tracer. There were two clear options for this: fixed point representation, and floating point.

Fixed point initially felt like the intuitive way to manage values throughout the ray tracer, as there are significant potential hardware optimisations - when the range of values needing to be represented is sufficiently limited, fixed-point operations can take up a smaller footprint within hardware design. However, similarly to what was discussed in Ken's prior work [1], we found that for the purposes of ray tracing, a "considerable number" of both integer and fractional

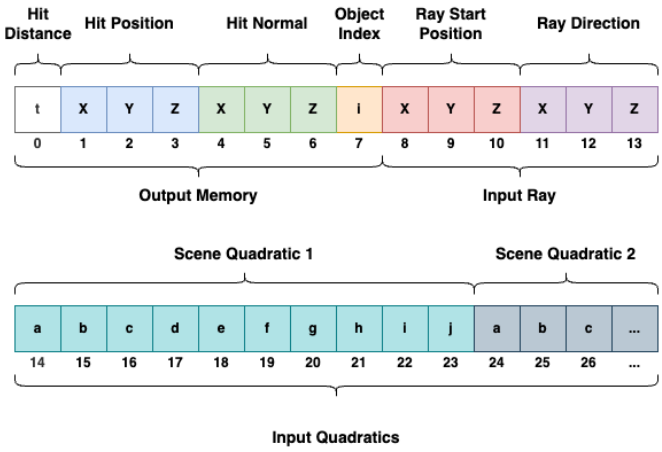

Figure 4: *Diagram showing the structure of the contiguous memory passed into FPGA.*

bits are required in order to adequately operate the environment. As such, there was not any real gain to be had from using fixed point representations of numbers. In addition, Python does not natively support fixed point numbers, and as such additional (potentially slow) frameworks would need to be constructed to handle these numbers being returned from the FPGA board. As such, 32-bit floating point numbers were used throughout the project. This allowed for increased FPGA development time, and simplified the Python interface to the design.

To minimise the number of variables being passed to the FPGA, the final solution simply passes an integer stating the number of quadratic objects in the buffer, and a pointer to the contiguous memory location containing all data needed for the calculations. Figure 4 shows the exact structure of this contiguous memory: memory locations 0-7 are the output locations and will filled at the end of the ray-scene calculation, locations 8-13 are where the current ray being tested is defined, and locations 14 and above define each quadratic object in terms of ten numbers. For ease of use, all values in the memory location take up the space of one float, but the object index is cast to an integer with Python upon being returned.

Data Transfer

Resource	Used	Total	Used (%)
Digital Signal Processor	200	220	90
Flip-Flop	20045	106400	18
Look-up Table	29224	53200	54

Figure 5: *Table showing resource utilisation for IP block design*

Data transfer to, from, and within the FPGA is

performed using what is known as the Advanced eXtensible Interface (AXI). It is defined as an "on-chip communication bus protocol" [14]. Within the PYNQ-Z2, there are various ways of transferring data across the Python-FPGA interface. The simplest (and slowest) is AXI-lite, which implements a register-like AXI bus primarily designed to transfer control signals to and from the FPGA. AXI Stream is designed for transferring stream data around within a chip, though can be used to interact with the PYNQ framework. Finally, AXI Master is most effective in situations where IP cores want to access external addressable memory locations (for example, in RAM).

Data transfer speeds between the intersection testing hardware and the Python ray tracer was identified as a critical bottleneck in this project, as it tends to be for FPGA-based projects [15]. As such, various approaches were tested, before one was decided upon. The initial approach to test the Overlay was to pass every value for one ray-quadratic intersection test in using 16 AXI-lite registers as input (6 values to define a ray, 10 values to define a quadratic) and 7 AXI-lite registers as output. This proved the functionality of the designed hardware, though was neither scaleable nor fast. An AXI Stream interface was then experimented with, but this was not well implemented within the PYNQ framework and had significant timing issues. Finally, the AXI Master approach was tested, using Direct Memory Access (DMA). In this, Numpy was used to initialise a contiguous memory location within RAM and fill it with the quadratic object definitions. For each ray to be tested against this object set, a ray was loaded into another memory location, and these memory locations were passed to the FPGA Overlay via AXI-lite. Outputs were provided in a third static memory location, and the process is repeated. This minimised the number of read/write operations performed by Python, which proved to be the main bottleneck.

As the FPGA has limited memory within it, there is an upper limit to the number of quadratics that can be loaded into it in one batch. Given the hardware limitations and the design size, the FPGA design could only hold 50 quadratic objects at a time. So, a part of the Python interface was dedicated to a batching system, which stores batches of up to 50 quadratic objects in various contiguous memory arrays, and changes the pointer being provided to the FPGA design as necessary to hand off new batches to the board.

IP Block Construction

A ray-quadratic intersection tester was implemented in C++, within Vitis HLS. This tester iterates through each quadratic in the buffer and finds the nearest intersection point with the supplied ray, if there is one. In order to leverage the speed-up potential of the FPGA, this loop was fully pipelined with an Initiation Interval (II) of 1, meaning that the processing of a new input is started at each clock cycle. With a single ray-quadratic intersection taking 93 clock cycles, 100 ray-quadratic intersections would take 192 clock cycles - only one additional clock cycle per quadratic in the scene. This was only possible by defining an output buffer for all t-values calculated and having each input of the pipeline write its values to a different index within the output buffer.

```
float calcInvSqRoot(float n) {
    const float threehalfs = 1.5F;
    float y = n;
    long i = * (long *) &y;
    i = 0x5f3759df - (i >> 1);
    y = * (float *) &i;
    y = y * (threehalfs - ((n * 0.5F) * y * y));
    return y;
}
```

Figure 6: *An implementation of the Fast Inverse Square Root algorithm, from [16]*

Once the pipeline completes and all t-values have been calculated and stored, we try to find the smallest t-value in the array. This is an $\mathcal{O}(n)$ operation as currently implemented - it is programmed as a tight loop that minimises the number of operations per iteration, as it is the primary bottleneck within the chip design. Once the minimum t-value has been calculated, the ray-quadratic hit position and normal can be calculated and stored in the output section of the buffer, along with the index of the object that has been hit. The full IP Block C++ code can be found in the Appendix.

```
float mod = calcInvSqRoot(
        (n_x * n_x) +
        (n_y * n_y) +
        (n_z * n_z)
    );

buff[4] = n_x * mod;
buff[5] = n_y * mod;
buff[6] = n_z * mod;
```

Figure 7: *Normalisation of the calculated hit normal vector, using the Fast Inverse Square Root*

When normalising the ray-quadratic hit normal vector calculated, the inverse square root was calculated using a variant of the "Fast Inverse Square Root" code originally theorised by William Kahan and K.C. Ng and made famous by its use in the video game Quake III Arena [17]. An implementation of this can

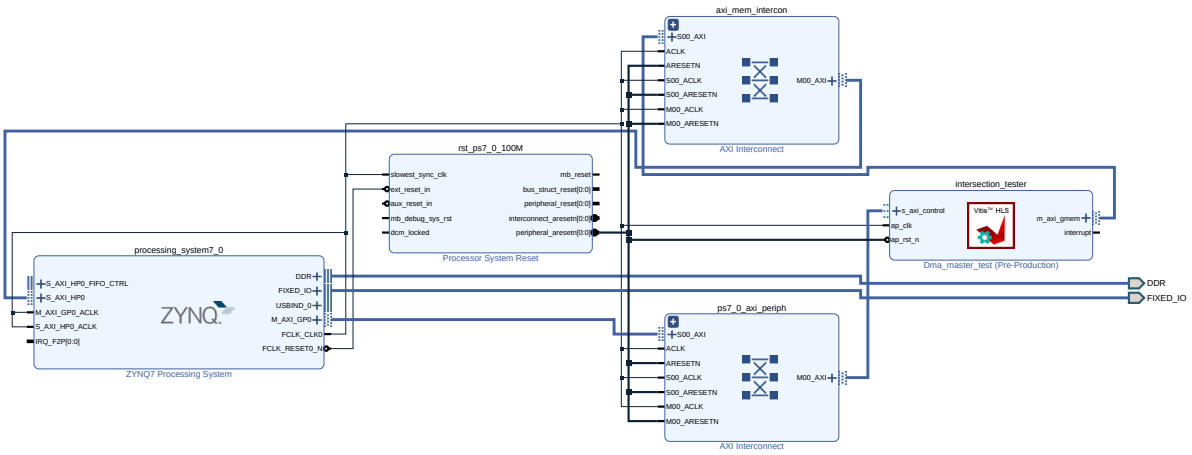

Figure 8: *Diagram showing the block design view of the completed FPGA design*

be found in Figure 6, and its usage is shown in Figure 7. This method results in 10 fewer clock cycles compared to the alternative (a square root operation followed by three parallel divisions).

The ray-quadratic intersection test equation defined in Section uses 59 multiplications, 25 additions, 3 subtractions, 2 divisions, and a square root. However, this was able to be reduced by storing and reusing repeated values as we went along, in hardware. The table in Figure 9 shows the reduction of operations achieved, thus minimising both the space taken up on the FPGA and the number of clock cycles required to compute an intersection. The majority of these operations were able to be performed in parallel or nearly so, however the final operations (ie. square root followed by division, shown in Section when the t value is being calculated) are inherently sequential, and must be performed as such (see Figure 10).

Operation	Expected	Actual
Multiplication	59	41
Addition	25	22
Subtraction	3	2
Division	2	2
Square Root	1	1

Figure 9: *Table showing number of mathematical operations needed in hardware*

Block Design

The block design was largely performed by Xilinx Vivado, as an automatic process. We simply specified that the custom IP Block generated from Vitis HLS should be integrated into the ZYNQ7 Processing system, and the routing was generated from this instruction (see Figure 8 for full design). Vivado implemented two AXI Interconnects to manage the inputs and outputs specified in the Block design. One acts as the interconnect for AXI-Lite, which the PYNQ Python overlay uses to set the block's parameters and trigger a new run. The other acts as the memory interconnect for AXI Master, to allow the IP Block to perform direct memory access to memory locations within the on-board RAM. This block design is synthesised and exported as a .bit (Bitstream) file and a .hwh (Hardware Handoff) file whenever a change is made within the IP design. These two files are copied over to the PYNQ-Z2 board, where it is picked up as an Overlay by the PYNQ system.

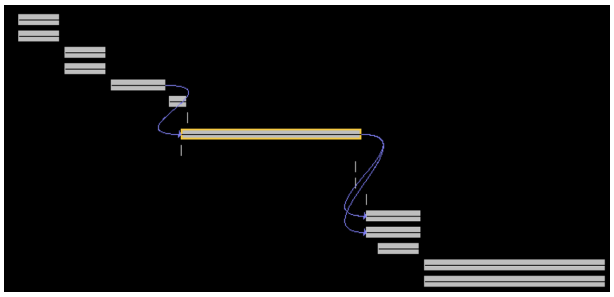

Figure 10: *Schedule Viewer diagram of sequential part of intersection test. Highlighted block is square root operation.*

Python Ray Caster

A basic Python ray caster was implemented, for the purposes of testing the ray-quadratic intersection FPGA design in a "real-life" situation. This ray caster implemented the Phong lighting model [2], but did not include additional features such as reflection, refraction, or shadows. A Python version of the ray-quadratic intersection test was implemented, to act as a baseline to compare the FPGA design against.

The Raytracer class was designed to allow for simple hot-swapping of the intersection test function - this allows for the same image to be created using either Python or the FPGA design, with just a different value being passed into it. PYNQ's Overlay functionality was used to interface with the FPGA chip. Data was stored in Numpy arrays in order to optimise the Python code as much as possible while still maintaining the simplicity of the test setup, as this was not the focus of the project.

Evaluation of practical application

The ray-quadratic intersection tester FPGA design was benchmarked against a varying number of quadratic objects, and compared to the equivalent intersection tester written in Python. The image size being generated was fixed to be 20×20px, in order to minimise testing time while still giving significant enough results. Scenes with everything ranging from 1 object to 500 were tested, and timing data was recorded for the time taken for an entire image to be generated, the average time for each ray to be computed with respect to a scene, and for the average time for each ray-quadratic intersection. This was ran 10 times, and mean averages at each scene size were taken. The raw data from these experiments can be found in Appendix 1. This was then graphed in order to better understand the speedups being achieved by the FPGA design.

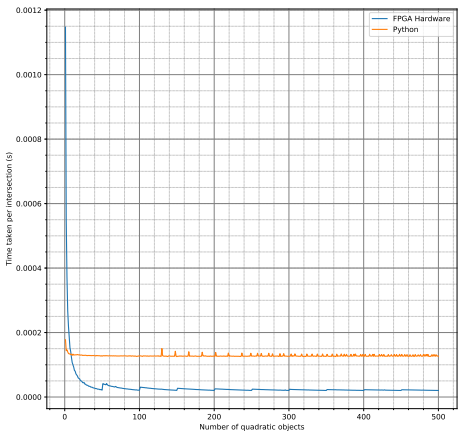

Figure 11: *Graph showing time taken per ray-quadratic intersection test for both Python and the FPGA design, as the number of quadratic objects in the scene increases*

The graph in Figure 11 shows the total time taken for either the Python ray caster or the hardware-accelerated ray caster as the number of quadratic objects in the scene increases. We can clearly see that the time taken by the Python ray caster increases linearly as the number of quadratic objects increases. This is as expected without any optimisations, as every quadratic in the scene has to be checked against each ray in order to find the closest hit. There are minor anomalous spikes in this data that occur with increasing likelihood as the number of quadratic objects in the scene increases - this is most likely due to Python's garbage collection cycles, as the tests continue running.

It takes on average 0.059005s for the Python ray caster to compute an image with 1 quadratic object. To compare, it takes it 6.429039s on average to compute an image with 500 quadratic objects. Using the gradient of the straight line of best fit of our data, we calculate the per-quadratic time increase to be 0.012740s, with a fixed overhead of 0.046266s.

Figure 12: *Generated image of quadratic object using FPGA design*

Our hardware accelerated ray caster does not increase linearly with the number of quadratic objects in the scene, but rather jumps in time taken at every interval of 50 quadratic objects. This is due to the hardware chip being able to compute a maximum of 50 ray-quadratic intersection tests at a time, and the Python interface batching jobs into this size as a result.

It takes on average 0.177155s for the hardware accelerated ray caster to compute an image with 1 quadratic object. To compare, it takes it 1.146608s on average to compute an image with 500 quadratic objects. There is a fixed initial overhead of 0.080209s, with each multiple of 50 quadratics taking an additional 0.096945s total. This gives us a per-quadratic time increase of 0.001939s - this is a speedup of 6.57x using the hardware acceleration. It is worth noting

Method Used	Overhead (s)	Avg Time Per Quadratic (s)
Python	0.046266	0.012740
FPGA	0.080209	0.001939
Speed-up (×)		**6.57**

Figure 13: *Performance evaluation of Intersection Testing IP Core*

that it takes essentially the same amount of time to render 1 quadratic object as it does 50, to within the uncertainty of the measurements being taken. This is because the time increase per quadratic in the scene only occurs when the hardware buffer size maxes out and an additional batch is needed per ray.

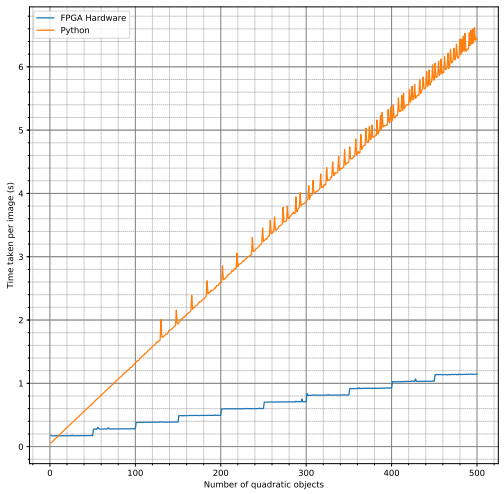

Figure 14: *Graph showing total image generation time for both Python and the FPGA design, as the number of quadratic objects in the scene increases*

The clock frequency of the FPGA board is 100MHz, meaning that each clock cycle occurs in 10ns. The final FPGA board design states a minimum latency of 557 cycles and a maximum latency of 870 cycles, equating to between 5.57μs and 8.7μs per batch job, depending on how many quadratics are in that batch of the scene. This allows for a theoretical minimum average execution time of 174ns per quadratic. It is evident, therefore, that the limiting factors in our design are the data transfer between the CPU and FPGA board, and the speed of the Python software aspects of the ray caster, as they are orders of magnitude slower. The hardware-software co-design approach utilised in this project allows for a much simpler development experience at the expense of reduced performance compared to implementing all required functionality directly into hardware.

Conclusion and Further Work

In this research paper, we have discussed, designed, implemented and evaluated a ray-quadratic intersection tester IP core. We used Xilinx Vivado to synthesise this design to be compatible with the PYNQ-Z2 FPGA board, and wrote a Python Overlay interface with the FPGA design. We have shown that the hardware accelerated version of the intersection tester outperforms the software-only version by a factor of at least 6.5 times when using it to generate a ray casted image.

There are various potential improvements that could be made to the design. To further increase the performance of the software-hardware co-design, more attention should be paid to the interface between Python and the FPGA IP core. An AXI-Stream interface would likely allow for the quadratic objects to be loaded into the FPGA, and for rays to be streamed through it much faster than currently available with the batch-job method. With this approach, each ray-quadratic intersection pair would need to be labelled in some way, so that as results are continuously returned to Python, they can be distinguished.

On the software side of this project, some level of multi-threading feels necessary to handle the sending and receiving of the data to and from the FPGA IP core. This could be implemented as a worker thread, either in Python or in C++. An alternative is for the entire interface to be rewritten in C++ and wrapped as a Python library, in order to achieve higher performance when reading and writing to memory.

References

[1] Kenneth M Cameron. *Acceleration of CSG Using a Custom VLSI Device*. May 1993.

[2] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: https://doi.org/10.1145/360825.360839.

[3] H. Gouraud. "Continuous Shading of Curved Surfaces". In: *IEEE Transactions on Computers* C-20.6 (1971), pp. 623–629. DOI: 10.1109/T-C.1971.223313.

[4] Jan. 2023. URL: https://en.wikipedia.org/wiki/Phong_reflection_model.

[5] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.

[6] *Arnold features: 2024, 2023, 2022 features: Autodesk UK*. Apr. 2023. URL: https://www.autodesk.co.uk/products/arnold/features.

[7] Ryan Frazier. *Rendering constructive solid geometry with python*. Mar. 2021. URL: https://www.fotonixx.com/posts/efficient-csg/.

[8] *TUL PYNQ-Z2 board*. URL: https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html.

[9] *What is an FPGA?* URL: https://www.arm.com/glossary/fpga.

[10] Hayden Kwok-Hay So and Cheng Liu. "FPGA Overlays". In: *FPGAs for Software Programmers*. Ed. by Dirk Koch, Frank Hannig, and Daniel Ziener. Cham: Springer International Publishing, 2016, pp. 285–305. ISBN: 978-3-319-26408-0. DOI: 10.1007/978-3-319-26408-0_16. URL: https://doi.org/10.1007/978-3-319-26408-0_16.

[11] *Python productivity for Zynq*. URL: http://www.pynq.io/home.html.

[12] Roberto Millon, Emmanuel Frati, and Enzo Rucci. "A Comparative Study between HLS and HDL on SoC for Image Processing Applications". In: *CoRR* abs/2012.08320 (2020). arXiv: 2012.08320. URL: https://arxiv.org/abs/2012.08320.

[13] *Vitis HLS*. May 2022. URL: https://xilinx.github.io/Vitis-Tutorials/2022-1/build/html/docs/Getting_Started/Vitis_HLS/Getting_Started_Vitis_HLS.html.

[14] *AXI Protocol Specification*. URL: https://developer.arm.com/documentation/ihi0022/e/.

[15] Thang Huynh. "FPGA-based Acceleration for Convolutional Neural Networks on PYNQ-Z2". In: *International Journal of Computing and Digital Systems* 11 (Jan. 2022), pp. 441–449. DOI: 10.12785/ijcds/110136.

[16] Sudhir Sharma. *Fast inverse square root in c*. URL: https://www.tutorialspoint.com/fast-inverse-square-root-in-cplusplus.

[17] Chris Lomont. "Fast Inverse Square Root". In: (Feb. 2003). URL: http://www.matrix67.com/data/InvSqrt.pdf.

Appendix

Raw Experimental Data

Table 1: *Ray-Quadratic experimental data*

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
1	0.001147	0.001492	0.177155	0.000178	0.000309	0.059006
2	0.000550	0.001444	0.170764	0.000144	0.000414	0.066300
3	0.000373	0.001463	0.172442	0.000146	0.000566	0.081590
4	0.000268	0.001411	0.167128	0.000140	0.000688	0.093846
5	0.000220	0.001442	0.170979	0.000135	0.000803	0.105339
6	0.000188	0.001473	0.173815	0.000134	0.000930	0.119726
7	0.000153	0.001413	0.168072	0.000134	0.001066	0.134890
8	0.000141	0.001469	0.173878	0.000132	0.001182	0.143682
9	0.000123	0.001447	0.171028	0.000133	0.001328	0.160837
10	0.000107	0.001412	0.167881	0.000129	0.001421	0.168165
11	0.000100	0.001443	0.171223	0.000133	0.001594	0.185721
12	0.000093	0.001460	0.172485	0.000130	0.001689	0.195327
13	0.000083	0.001417	0.168555	0.000131	0.001833	0.210958
14	0.000080	0.001463	0.173933	0.000131	0.001957	0.222466
15	0.000072	0.001422	0.170992	0.000131	0.002096	0.236910
16	0.000068	0.001436	0.171656	0.000131	0.002224	0.249834
17	0.000065	0.001454	0.172436	0.000132	0.002371	0.263644
18	0.000059	0.001412	0.169001	0.000131	0.002490	0.276772
19	0.000058	0.001440	0.172034	0.000131	0.002612	0.288513
20	0.000056	0.001461	0.173703	0.000131	0.002745	0.301713
21	0.000051	0.001412	0.169378	0.000130	0.002866	0.314255
22	0.000050	0.001440	0.172395	0.000130	0.003002	0.327961
23	0.000048	0.001455	0.173131	0.000130	0.003126	0.339860
24	0.000045	0.001410	0.169251	0.000129	0.003224	0.352251
25	0.000044	0.001446	0.173464	0.000129	0.003382	0.366611
26	0.000045	0.001529	0.181834	0.000129	0.003494	0.378102
27	0.000040	0.001420	0.170713	0.000129	0.003624	0.390527
28	0.000040	0.001451	0.172933	0.000130	0.003761	0.404627
29	0.000037	0.001415	0.170426	0.000129	0.003874	0.416937
30	0.000037	0.001440	0.172923	0.000128	0.003980	0.427946
31	0.000036	0.001454	0.173545	0.000129	0.004128	0.441003
32	0.000033	0.001410	0.169991	0.000129	0.004253	0.454372
33	0.000033	0.001440	0.173337	0.000129	0.004385	0.467643
34	0.000033	0.001457	0.174136	0.000128	0.004497	0.477950
35	0.000031	0.001412	0.170483	0.000129	0.004644	0.494202
36	0.000031	0.001443	0.173940	0.000128	0.004769	0.505746
37	0.000030	0.001455	0.174834	0.000129	0.004893	0.518929
38	0.000028	0.001415	0.171000	0.000129	0.005028	0.531563
39	0.000028	0.001440	0.173668	0.000129	0.005148	0.544745
40	0.000028	0.001451	0.174065	0.000128	0.005256	0.555132
41	0.000026	0.001418	0.171479	0.000127	0.005362	0.566664
42	0.000026	0.001444	0.174387	0.000129	0.005555	0.588059
43	0.000026	0.001448	0.174132	0.000128	0.005633	0.593199
44	0.000024	0.001415	0.171547	0.000129	0.005791	0.609165
45	0.000024	0.001441	0.174410	0.000128	0.005901	0.620642
46	0.000024	0.001458	0.175533	0.000128	0.006028	0.633307

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
47	0.000023	0.001415	0.171667	0.000128	0.006157	0.647011
48	0.000023	0.001473	0.178003	0.000127	0.006237	0.654337
49	0.000023	0.001452	0.175107	0.000128	0.006406	0.671907
50	0.000021	0.001412	0.171652	0.000127	0.006472	0.677768
51	0.000041	0.002455	0.277285	0.000128	0.006661	0.697886
52	0.000041	0.002459	0.277065	0.000128	0.006797	0.711223
53	0.000039	0.002418	0.273439	0.000128	0.006902	0.722101
54	0.000039	0.002460	0.277612	0.000128	0.007068	0.739022
55	0.000038	0.002424	0.274622	0.000128	0.007158	0.747857
56	0.000042	0.002697	0.305125	0.000128	0.007315	0.763863
57	0.000038	0.002550	0.288570	0.000128	0.007427	0.775934
58	0.000036	0.002428	0.277631	0.000128	0.007542	0.786524
59	0.000035	0.002419	0.276336	0.000127	0.007652	0.797566
60	0.000035	0.002442	0.279083	0.000128	0.007822	0.814460
61	0.000034	0.002424	0.277225	0.000127	0.007906	0.822832
62	0.000034	0.002421	0.277297	0.000128	0.008067	0.838805
63	0.000033	0.002419	0.277315	0.000128	0.008184	0.851477
64	0.000033	0.002439	0.279679	0.000127	0.008287	0.860447
65	0.000032	0.002425	0.278176	0.000128	0.008449	0.877731
66	0.000032	0.002426	0.278292	0.000127	0.008544	0.887608
67	0.000031	0.002423	0.277747	0.000128	0.008680	0.901175
68	0.000033	0.002609	0.298661	0.000128	0.008818	0.915195
69	0.000031	0.002521	0.288852	0.000127	0.008919	0.927174
70	0.000030	0.002423	0.277802	0.000127	0.009048	0.937950
71	0.000029	0.002424	0.278296	0.000127	0.009172	0.950009
72	0.000029	0.002442	0.279933	0.000127	0.009295	0.962964
73	0.000028	0.002421	0.278454	0.000127	0.009418	0.975365
74	0.000028	0.002423	0.279068	0.000128	0.009571	0.990827
75	0.000028	0.002426	0.279178	0.000128	0.009708	1.004571
76	0.000028	0.002449	0.281250	0.000128	0.009830	1.017028
77	0.000027	0.002423	0.278626	0.000127	0.009939	1.028169
78	0.000027	0.002428	0.279053	0.000127	0.010070	1.041186
79	0.000026	0.002431	0.280106	0.000128	0.010222	1.056676
80	0.000026	0.002443	0.281908	0.000127	0.010332	1.067569
81	0.000026	0.002427	0.279346	0.000128	0.010464	1.080966
82	0.000025	0.002424	0.279581	0.000128	0.010597	1.094182
83	0.000025	0.002422	0.279125	0.000127	0.010675	1.102242
84	0.000025	0.002447	0.282192	0.000127	0.010789	1.113486
85	0.000024	0.002425	0.279461	0.000128	0.010985	1.133459
86	0.000024	0.002438	0.281027	0.000127	0.011058	1.140697
87	0.000024	0.002424	0.280081	0.000128	0.011256	1.159490
88	0.000024	0.002445	0.281415	0.000128	0.011376	1.172736
89	0.000024	0.002438	0.281245	0.000126	0.011388	1.173496
90	0.000023	0.002429	0.280841	0.000128	0.011612	1.196396
91	0.000023	0.002425	0.279697	0.000127	0.011732	1.208283
92	0.000023	0.002438	0.281726	0.000127	0.011820	1.217457
93	0.000022	0.002431	0.281507	0.000127	0.011901	1.225863
94	0.000022	0.002426	0.280702	0.000127	0.012111	1.247020
95	0.000022	0.002425	0.280415	0.000127	0.012163	1.252226
96	0.000022	0.002473	0.285677	0.000126	0.012226	1.257275
97	0.000021	0.002424	0.281851	0.000126	0.012378	1.273894

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
98	0.000021	0.002423	0.281006	0.000127	0.012569	1.292712
99	0.000021	0.002425	0.281902	0.000127	0.012752	1.311723
100	0.000021	0.002445	0.283320	0.000127	0.012794	1.315872
101	0.000031	0.003434	0.382919	0.000128	0.013031	1.340556
102	0.000030	0.003433	0.382993	0.000128	0.013147	1.351970
103	0.000030	0.003429	0.382586	0.000127	0.013182	1.356038
104	0.000030	0.003436	0.383626	0.000126	0.013232	1.360494
105	0.000029	0.003436	0.382554	0.000128	0.013525	1.390688
106	0.000029	0.003464	0.387101	0.000128	0.013662	1.403879
107	0.000029	0.003439	0.382625	0.000127	0.013745	1.412364
108	0.000029	0.003438	0.381774	0.000127	0.013898	1.427627
109	0.000029	0.003476	0.386891	0.000127	0.014007	1.438987
110	0.000028	0.003440	0.384543	0.000127	0.014058	1.443385
111	0.000028	0.003433	0.382993	0.000128	0.014290	1.466457
112	0.000028	0.003438	0.384109	0.000127	0.014326	1.470770
113	0.000027	0.003442	0.385017	0.000127	0.014432	1.481329
114	0.000027	0.003433	0.385231	0.000127	0.014664	1.504407
115	0.000027	0.003460	0.387017	0.000127	0.014760	1.514567
116	0.000027	0.003422	0.384365	0.000127	0.014870	1.525474
117	0.000026	0.003433	0.384175	0.000127	0.014990	1.537405
118	0.000026	0.003435	0.385428	0.000127	0.015122	1.551011
119	0.000026	0.003430	0.384072	0.000127	0.015245	1.562440
120	0.000026	0.003432	0.384475	0.000128	0.015437	1.582249
121	0.000026	0.003451	0.387408	0.000127	0.015531	1.592069
122	0.000025	0.003436	0.385133	0.000127	0.015612	1.602437
123	0.000025	0.003427	0.384857	0.000127	0.015787	1.617213
124	0.000025	0.003434	0.385100	0.000127	0.015886	1.627832
125	0.000025	0.003427	0.384878	0.000128	0.016132	1.652837
126	0.000024	0.003431	0.384844	0.000126	0.016069	1.646038
127	0.000024	0.003452	0.387079	0.000127	0.016263	1.666322
128	0.000024	0.003442	0.385232	0.000126	0.016326	1.672292
129	0.000024	0.003498	0.392363	0.000126	0.016411	1.681728
130	0.000024	0.003478	0.389920	0.000150	0.019688	2.008983
131	0.000024	0.003439	0.387454	0.000128	0.016906	1.730682
132	0.000023	0.003441	0.386910	0.000127	0.016904	1.730141
133	0.000023	0.003439	0.387006	0.000127	0.016965	1.736385
134	0.000023	0.003432	0.385641	0.000127	0.017102	1.750360
135	0.000023	0.003438	0.388292	0.000127	0.017244	1.764607
136	0.000023	0.003441	0.387193	0.000127	0.017394	1.779784
137	0.000023	0.003436	0.386622	0.000126	0.017354	1.775651
138	0.000022	0.003439	0.386873	0.000126	0.017496	1.789862
139	0.000022	0.003446	0.387640	0.000127	0.017843	1.824906
140	0.000022	0.003443	0.387539	0.000128	0.018002	1.841580
141	0.000022	0.003429	0.386304	0.000127	0.018043	1.845307
142	0.000022	0.003443	0.387987	0.000126	0.018077	1.848994
143	0.000022	0.003444	0.387845	0.000127	0.018232	1.864403
144	0.000021	0.003434	0.387245	0.000126	0.018348	1.875913
145	0.000021	0.003457	0.390236	0.000127	0.018498	1.891173
146	0.000021	0.003448	0.388941	0.000127	0.018608	1.902375
147	0.000021	0.003441	0.387712	0.000126	0.018717	1.913730
148	0.000021	0.003453	0.389861	0.000142	0.021153	2.157005

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
149	0.000021	0.003437	0.387793	0.000127	0.019034	1.945276
150	0.000021	0.003438	0.388586	0.000125	0.018935	1.935371
151	0.000027	0.004472	0.492330	0.000126	0.019210	1.962536
152	0.000027	0.004437	0.488264	0.000127	0.019385	1.981499
153	0.000027	0.004443	0.490314	0.000126	0.019481	1.991196
154	0.000027	0.004461	0.491398	0.000128	0.019772	2.020569
155	0.000026	0.004435	0.485769	0.000127	0.019788	2.022114
156	0.000026	0.004442	0.489246	0.000127	0.019998	2.043187
157	0.000026	0.004445	0.489385	0.000126	0.019968	2.040542
158	0.000026	0.004445	0.489927	0.000127	0.020166	2.060114
159	0.000026	0.004436	0.489124	0.000127	0.020269	2.069587
160	0.000026	0.004442	0.489471	0.000127	0.020448	2.088639
161	0.000025	0.004443	0.490618	0.000127	0.020549	2.098656
162	0.000025	0.004442	0.490207	0.000127	0.020677	2.112028
163	0.000025	0.004447	0.490453	0.000126	0.020710	2.115154
164	0.000025	0.004477	0.493884	0.000126	0.020857	2.129987
165	0.000025	0.004447	0.491136	0.000127	0.021033	2.147777
166	0.000025	0.004446	0.490833	0.000140	0.023448	2.389387
167	0.000025	0.004450	0.491287	0.000126	0.021256	2.170404
168	0.000024	0.004443	0.492071	0.000127	0.021500	2.194947
169	0.000024	0.004458	0.492835	0.000127	0.021567	2.201116
170	0.000024	0.004452	0.491746	0.000127	0.021692	2.214147
171	0.000024	0.004449	0.491999	0.000127	0.021839	2.228929
172	0.000024	0.004439	0.492046	0.000126	0.021880	2.233137
173	0.000024	0.004461	0.492872	0.000127	0.022024	2.247423
174	0.000023	0.004432	0.489960	0.000127	0.022217	2.267786
175	0.000023	0.004444	0.490921	0.000127	0.022435	2.289009
176	0.000023	0.004457	0.493681	0.000127	0.022423	2.288138
177	0.000023	0.004444	0.491725	0.000126	0.022447	2.290647
178	0.000023	0.004450	0.492581	0.000126	0.022630	2.308011
179	0.000023	0.004464	0.493556	0.000126	0.022762	2.321924
180	0.000023	0.004446	0.493899	0.000126	0.022845	2.330889
181	0.000023	0.004445	0.492223	0.000126	0.023006	2.345947
182	0.000023	0.004449	0.492686	0.000127	0.023161	2.361421
183	0.000022	0.004436	0.491783	0.000127	0.023297	2.376261
184	0.000022	0.004448	0.493403	0.000139	0.025730	2.619267
185	0.000022	0.004444	0.492646	0.000127	0.023692	2.415628
186	0.000022	0.004446	0.493681	0.000127	0.023849	2.431406
187	0.000022	0.004449	0.493735	0.000127	0.023867	2.433536
188	0.000022	0.004447	0.493189	0.000128	0.024183	2.465462
189	0.000022	0.004435	0.492703	0.000127	0.024096	2.455674
190	0.000022	0.004442	0.492540	0.000127	0.024223	2.469575
191	0.000022	0.004459	0.494622	0.000126	0.024269	2.474112
192	0.000021	0.004445	0.493513	0.000126	0.024383	2.485360
193	0.000021	0.004438	0.493746	0.000127	0.024609	2.508076
194	0.000021	0.004512	0.501215	0.000126	0.024572	2.504781
195	0.000021	0.004434	0.492228	0.000127	0.024965	2.545073
196	0.000021	0.004446	0.493019	0.000127	0.024950	2.542957
197	0.000021	0.004465	0.496863	0.000126	0.024974	2.545061
198	0.000021	0.004442	0.492466	0.000127	0.025216	2.569539
199	0.000021	0.004430	0.492186	0.000127	0.025374	2.585912

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
200	0.000021	0.004453	0.495283	0.000126	0.025400	2.588382
201	0.000025	0.005436	0.594653	0.000126	0.025458	2.594904
202	0.000025	0.005455	0.595923	0.000138	0.028068	2.856127
203	0.000025	0.005459	0.596719	0.000127	0.025880	2.637598
204	0.000025	0.005461	0.597588	0.000126	0.025910	2.640202
205	0.000025	0.005473	0.598073	0.000127	0.026129	2.662258
206	0.000025	0.005451	0.597326	0.000126	0.026149	2.664277
207	0.000025	0.005462	0.597224	0.000126	0.026315	2.681149
208	0.000025	0.005451	0.596704	0.000127	0.026565	2.706565
209	0.000025	0.005479	0.599092	0.000127	0.026685	2.717383
210	0.000024	0.005462	0.599366	0.000127	0.026713	2.721579
211	0.000024	0.005474	0.598562	0.000127	0.026862	2.736475
212	0.000024	0.005445	0.596010	0.000126	0.026947	2.744927
213	0.000024	0.005456	0.597433	0.000127	0.027290	2.779625
214	0.000024	0.005442	0.596742	0.000127	0.027209	2.771655
215	0.000024	0.005456	0.597533	0.000126	0.027272	2.778012
216	0.000024	0.005438	0.596218	0.000127	0.027602	2.811066
217	0.000024	0.005453	0.597353	0.000126	0.027529	2.803772
218	0.000024	0.005483	0.600967	0.000126	0.027708	2.821674
219	0.000023	0.005451	0.598191	0.000137	0.030062	3.057377
220	0.000023	0.005462	0.599393	0.000126	0.027919	2.842973
221	0.000023	0.005442	0.596927	0.000128	0.028321	2.882936
222	0.000023	0.005444	0.597674	0.000127	0.028309	2.882725
223	0.000023	0.005448	0.598107	0.000127	0.028389	2.890389
224	0.000023	0.005448	0.598204	0.000126	0.028450	2.896712
225	0.000023	0.005450	0.597935	0.000126	0.028595	2.911061
226	0.000023	0.005439	0.596995	0.000127	0.028825	2.934320
227	0.000022	0.005451	0.598213	0.000126	0.028779	2.930143
228	0.000023	0.005480	0.602330	0.000126	0.028946	2.946747
229	0.000022	0.005462	0.600294	0.000126	0.029095	2.961868
230	0.000022	0.005455	0.600049	0.000126	0.029187	2.970940
231	0.000022	0.005468	0.601511	0.000126	0.029309	2.983293
232	0.000022	0.005471	0.601302	0.000126	0.029466	2.999101
233	0.000022	0.005478	0.599085	0.000126	0.029534	3.005177
234	0.000022	0.005462	0.600532	0.000126	0.029653	3.017948
235	0.000022	0.005461	0.601188	0.000126	0.029708	3.023557
236	0.000022	0.005440	0.598227	0.000126	0.029858	3.038789
237	0.000022	0.005462	0.600431	0.000137	0.032512	3.304999
238	0.000021	0.005456	0.600556	0.000127	0.030322	3.084095
239	0.000021	0.005450	0.599683	0.000126	0.030369	3.089805
240	0.000021	0.005458	0.601263	0.000126	0.030420	3.095664
241	0.000021	0.005437	0.598429	0.000127	0.030693	3.121773
242	0.000021	0.005457	0.600237	0.000127	0.030754	3.128979
243	0.000021	0.005460	0.600845	0.000126	0.030868	3.140966
244	0.000021	0.005450	0.600995	0.000126	0.030967	3.152215
245	0.000021	0.005531	0.608703	0.000127	0.031176	3.171213
246	0.000021	0.005471	0.602611	0.000127	0.031343	3.188699
247	0.000021	0.005450	0.600528	0.000127	0.031417	3.195000
248	0.000021	0.005462	0.601828	0.000127	0.031543	3.208575
249	0.000021	0.005465	0.602572	0.000136	0.033993	3.453543
250	0.000020	0.005455	0.601885	0.000127	0.031851	3.239508

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
251	0.000024	0.006471	0.703307	0.000127	0.032017	3.256640
252	0.000024	0.006464	0.703177	0.000127	0.032104	3.265821
253	0.000024	0.006464	0.704083	0.000127	0.032309	3.286563
254	0.000024	0.006473	0.704387	0.000126	0.032226	3.278337
255	0.000024	0.006454	0.702245	0.000127	0.032517	3.307280
256	0.000024	0.006452	0.702589	0.000127	0.032671	3.323604
257	0.000024	0.006472	0.705305	0.000127	0.032731	3.329482
258	0.000024	0.006472	0.705072	0.000136	0.035212	3.577266
259	0.000024	0.006473	0.706668	0.000127	0.033077	3.363232
260	0.000024	0.006465	0.704878	0.000127	0.033118	3.368814
261	0.000023	0.006449	0.702622	0.000127	0.033354	3.392060
262	0.000023	0.006471	0.705861	0.000127	0.033349	3.391705
263	0.000023	0.006442	0.702697	0.000135	0.035721	3.629053
264	0.000023	0.006467	0.704885	0.000127	0.033645	3.421813
265	0.000023	0.006456	0.704433	0.000126	0.033618	3.418745
266	0.000023	0.006484	0.706262	0.000126	0.033746	3.431950
267	0.000023	0.006452	0.706311	0.000126	0.033879	3.445539
268	0.000023	0.006455	0.704606	0.000127	0.034259	3.483506
269	0.000023	0.006450	0.704514	0.000127	0.034306	3.488526
270	0.000023	0.006472	0.706908	0.000127	0.034378	3.495680
271	0.000023	0.006462	0.705602	0.000127	0.034490	3.506949
272	0.000023	0.006472	0.706323	0.000127	0.034610	3.519018
273	0.000022	0.006444	0.703047	0.000136	0.037232	3.781014
274	0.000022	0.006465	0.706739	0.000127	0.034862	3.544518
275	0.000022	0.006461	0.705670	0.000127	0.034978	3.555864
276	0.000022	0.006470	0.706878	0.000126	0.034971	3.555580
277	0.000022	0.006453	0.705138	0.000126	0.035116	3.569832
278	0.000022	0.006472	0.707290	0.000134	0.037404	3.799203
279	0.000022	0.006473	0.707046	0.000126	0.035431	3.603451
280	0.000022	0.006460	0.706538	0.000126	0.035496	3.608333
281	0.000022	0.006478	0.708075	0.000127	0.035703	3.629537
282	0.000022	0.006483	0.709260	0.000127	0.035874	3.646536
283	0.000022	0.006452	0.705517	0.000127	0.036110	3.670515
284	0.000022	0.006462	0.707477	0.000127	0.036145	3.674017
285	0.000021	0.006467	0.708157	0.000127	0.036440	3.703557
286	0.000021	0.006464	0.708097	0.000126	0.036209	3.680588
287	0.000021	0.006451	0.704495	0.000126	0.036223	3.681838
288	0.000022	0.006562	0.717946	0.000134	0.038864	3.946178
289	0.000021	0.006490	0.709256	0.000126	0.036480	3.707699
290	0.000021	0.006471	0.708908	0.000127	0.036847	3.744706
291	0.000021	0.006480	0.709087	0.000127	0.037038	3.763607
292	0.000021	0.006480	0.709258	0.000127	0.037155	3.775859
293	0.000021	0.006505	0.712502	0.000134	0.039544	4.014432
294	0.000021	0.006458	0.706800	0.000127	0.037421	3.802548
295	0.000022	0.006876	0.751646	0.000127	0.037607	3.820989
296	0.000021	0.006476	0.709088	0.000127	0.037703	3.830899
297	0.000021	0.006445	0.706335	0.000127	0.037778	3.838893
298	0.000021	0.006472	0.709476	0.000126	0.037764	3.836762
299	0.000020	0.006455	0.707727	0.000127	0.038018	3.861991
300	0.000020	0.006467	0.708923	0.000126	0.037996	3.860733
301	0.000024	0.007731	0.838318	0.000127	0.038271	3.889689

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
302	0.000024	0.007476	0.811045	0.000127	0.038494	3.910996
303	0.000023	0.007454	0.809239	0.000134	0.040604	4.122657
304	0.000023	0.007467	0.810036	0.000126	0.038415	3.903628
305	0.000023	0.007460	0.810254	0.000128	0.039075	3.970358
306	0.000023	0.007466	0.810318	0.000127	0.039056	3.968467
307	0.000023	0.007488	0.814110	0.000127	0.039095	3.972460
308	0.000023	0.007517	0.816360	0.000134	0.041416	4.204383
309	0.000023	0.007469	0.812085	0.000127	0.039529	4.016132
310	0.000023	0.007489	0.813121	0.000127	0.039503	4.012397
311	0.000023	0.007490	0.814559	0.000127	0.039564	4.018671
312	0.000023	0.007500	0.815173	0.000127	0.039707	4.034103
313	0.000023	0.007464	0.810923	0.000126	0.039666	4.030007
314	0.000023	0.007463	0.810872	0.000126	0.039797	4.043210
315	0.000023	0.007475	0.813126	0.000126	0.039946	4.058271
316	0.000023	0.007466	0.811634	0.000127	0.040121	4.075969
317	0.000023	0.007481	0.814020	0.000133	0.042429	4.307361
318	0.000022	0.007478	0.813305	0.000126	0.040302	4.093933
319	0.000022	0.007468	0.812728	0.000126	0.040389	4.102833
320	0.000022	0.007477	0.813749	0.000126	0.040322	4.096482
321	0.000022	0.007482	0.815812	0.000126	0.040735	4.137403
322	0.000022	0.007486	0.814426	0.000126	0.040642	4.128706
323	0.000022	0.007478	0.814342	0.000126	0.040820	4.146546
324	0.000022	0.007481	0.814232	0.000134	0.043413	4.406662
325	0.000022	0.007461	0.812311	0.000127	0.041329	4.197902
326	0.000022	0.007547	0.821012	0.000127	0.041463	4.211869
327	0.000022	0.007462	0.812360	0.000126	0.041492	4.213089
328	0.000022	0.007488	0.816186	0.000126	0.041579	4.223416
329	0.000022	0.007459	0.813823	0.000127	0.041901	4.255897
330	0.000022	0.007471	0.813656	0.000127	0.041908	4.256115
331	0.000021	0.007450	0.811872	0.000133	0.044315	4.497408
332	0.000022	0.007490	0.815987	0.000127	0.042155	4.281451
333	0.000021	0.007464	0.813544	0.000126	0.042242	4.290882
334	0.000021	0.007479	0.814697	0.000127	0.042638	4.330337
335	0.000021	0.007472	0.815179	0.000127	0.042567	4.323312
336	0.000021	0.007480	0.812964	0.000126	0.042554	4.321591
337	0.000021	0.007462	0.813380	0.000126	0.042665	4.333215
338	0.000021	0.007469	0.814377	0.000133	0.045235	4.590976
339	0.000021	0.007460	0.814425	0.000126	0.042940	4.360897
340	0.000021	0.007469	0.814548	0.000127	0.043249	4.390730
341	0.000021	0.007471	0.816953	0.000127	0.043290	4.396286
342	0.000021	0.007474	0.815570	0.000126	0.043255	4.391355
343	0.000021	0.007459	0.813812	0.000126	0.043412	4.408617
344	0.000021	0.007481	0.818308	0.000126	0.043538	4.421369
345	0.000021	0.007478	0.816196	0.000134	0.046230	4.690903
346	0.000021	0.007485	0.817374	0.000126	0.043826	4.450021
347	0.000020	0.007444	0.813295	0.000127	0.044052	4.472013
348	0.000020	0.007459	0.814723	0.000126	0.044150	4.482772
349	0.000020	0.007463	0.816169	0.000126	0.044154	4.483281
350	0.000020	0.007468	0.816282	0.000126	0.044329	4.500971
351	0.000023	0.008445	0.915073	0.000133	0.046666	4.735921
352	0.000023	0.008486	0.919072	0.000127	0.044801	4.549059

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
353	0.000023	0.008436	0.914439	0.000126	0.044785	4.548152
354	0.000023	0.008450	0.915461	0.000126	0.044764	4.545397
355	0.000023	0.008467	0.917344	0.000126	0.044972	4.566815
356	0.000023	0.008459	0.917699	0.000127	0.045190	4.587735
357	0.000023	0.008436	0.914159	0.000126	0.045144	4.583225
358	0.000023	0.008475	0.918798	0.000133	0.047903	4.859258
359	0.000023	0.008449	0.916694	0.000126	0.045396	4.609587
360	0.000023	0.008503	0.922812	0.000126	0.045628	4.632745
361	0.000022	0.008443	0.915640	0.000127	0.045902	4.661093
362	0.000023	0.008571	0.929244	0.000126	0.045633	4.633472
363	0.000022	0.008431	0.914386	0.000126	0.045736	4.644177
364	0.000022	0.008497	0.921640	0.000133	0.048595	4.930385
365	0.000022	0.008432	0.914960	0.000126	0.046246	4.695302
366	0.000022	0.008503	0.922072	0.000127	0.046687	4.739979
367	0.000022	0.008458	0.918078	0.000127	0.046786	4.749753
368	0.000022	0.008513	0.923617	0.000127	0.046902	4.760808
369	0.000022	0.008469	0.919278	0.000127	0.046887	4.759777
370	0.000022	0.008520	0.924741	0.000134	0.049599	5.030612
371	0.000022	0.008442	0.916591	0.000127	0.047379	4.809669
372	0.000022	0.008508	0.923551	0.000127	0.047386	4.809993
373	0.000022	0.008472	0.920338	0.000127	0.047332	4.804049
374	0.000022	0.008522	0.925457	0.000133	0.049846	5.056942
375	0.000022	0.008455	0.919008	0.000127	0.047768	4.848732
376	0.000022	0.008461	0.919678	0.000127	0.047840	4.856095
377	0.000022	0.008513	0.924878	0.000132	0.050077	5.080420
378	0.000021	0.008428	0.916237	0.000127	0.048131	4.885716
379	0.000022	0.008502	0.923524	0.000127	0.048241	4.896961
380	0.000021	0.008450	0.918808	0.000127	0.048220	4.894576
381	0.000021	0.008508	0.925598	0.000127	0.048444	4.917191
382	0.000021	0.008437	0.917480	0.000126	0.048410	4.913788
383	0.000021	0.008493	0.923481	0.000132	0.050872	5.160269
384	0.000021	0.008441	0.918834	0.000126	0.048616	4.934912
385	0.000021	0.008496	0.924361	0.000127	0.048984	4.972145
386	0.000021	0.008455	0.920923	0.000127	0.049151	4.988839
387	0.000021	0.008499	0.925152	0.000133	0.051546	5.227948
388	0.000021	0.008455	0.920294	0.000127	0.049359	5.009728
389	0.000021	0.008509	0.926021	0.000133	0.052039	5.278448
390	0.000021	0.008448	0.921152	0.000127	0.049750	5.049006
391	0.000021	0.008501	0.925105	0.000127	0.049901	5.064595
392	0.000021	0.008457	0.920199	0.000127	0.050006	5.075155
393	0.000021	0.008514	0.927041	0.000127	0.049932	5.068111
394	0.000021	0.008522	0.927966	0.000127	0.050076	5.082433
395	0.000021	0.008510	0.926734	0.000126	0.050071	5.082071
396	0.000020	0.008448	0.920740	0.000132	0.052494	5.324216
397	0.000021	0.008522	0.928047	0.000127	0.050557	5.130738
398	0.000020	0.008484	0.924832	0.000126	0.050395	5.114560
399	0.000020	0.008515	0.927593	0.000132	0.052876	5.363320
400	0.000020	0.008459	0.922819	0.000126	0.050643	5.139600
401	0.000023	0.009488	1.025900	0.000127	0.050889	5.165817
402	0.000023	0.009504	1.028595	0.000132	0.053248	5.400453
403	0.000023	0.009443	1.021788	0.000127	0.051189	5.195534

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
404	0.000023	0.009486	1.025995	0.000126	0.051152	5.191787
405	0.000022	0.009457	1.023693	0.000127	0.051390	5.215911
406	0.000022	0.009456	1.023073	0.000127	0.051587	5.235929
407	0.000022	0.009459	1.023459	0.000126	0.051539	5.230937
408	0.000022	0.009487	1.026761	0.000133	0.054306	5.508327
409	0.000022	0.009472	1.025097	0.000127	0.052141	5.291518
410	0.000022	0.009470	1.025303	0.000127	0.052255	5.303045
411	0.000022	0.009465	1.025336	0.000127	0.052215	5.299264
412	0.000022	0.009486	1.027532	0.000132	0.054678	5.545888
413	0.000022	0.009465	1.025178	0.000126	0.052323	5.310208
414	0.000022	0.009511	1.029677	0.000133	0.055065	5.583797
415	0.000022	0.009463	1.025213	0.000126	0.052596	5.338153
416	0.000022	0.009514	1.030416	0.000127	0.052910	5.369181
417	0.000022	0.009496	1.028815	0.000127	0.053118	5.390659
418	0.000022	0.009510	1.030466	0.000127	0.053298	5.408299
419	0.000022	0.009511	1.029950	0.000127	0.053245	5.403457
420	0.000022	0.009513	1.031067	0.000126	0.053196	5.398116
421	0.000022	0.009490	1.028730	0.000132	0.055641	5.643341
422	0.000022	0.009525	1.032602	0.000127	0.053634	5.442513
423	0.000022	0.009488	1.029418	0.000127	0.054013	5.480496
424	0.000022	0.009587	1.039038	0.000132	0.056130	5.692270
425	0.000022	0.009525	1.034643	0.000127	0.053982	5.477612
426	0.000021	0.009489	1.031551	0.000127	0.054089	5.488415
427	0.000021	0.009483	1.028975	0.000132	0.056438	5.722876
428	0.000022	0.009875	1.070137	0.000126	0.054235	5.503083
429	0.000021	0.009493	1.030420	0.000127	0.054428	5.522557
430	0.000021	0.009499	1.031029	0.000127	0.054656	5.546327
431	0.000021	0.009505	1.031558	0.000127	0.054775	5.557727
432	0.000021	0.009477	1.028705	0.000126	0.054698	5.550062
433	0.000021	0.009495	1.031222	0.000132	0.057144	5.795210
434	0.000021	0.009504	1.032668	0.000126	0.054981	5.578759
435	0.000021	0.009495	1.030961	0.000127	0.055306	5.611402
436	0.000021	0.009480	1.030243	0.000127	0.055664	5.647507
437	0.000021	0.009503	1.032628	0.000131	0.057577	5.838852
438	0.000021	0.009487	1.030868	0.000127	0.055711	5.652427
439	0.000021	0.009540	1.036710	0.000127	0.055696	5.651045
440	0.000021	0.009485	1.031705	0.000127	0.055970	5.676725
441	0.000021	0.009500	1.032386	0.000132	0.058427	5.924487
442	0.000021	0.009498	1.032195	0.000127	0.056107	5.692397
443	0.000021	0.009497	1.032523	0.000127	0.056342	5.715160
444	0.000021	0.009506	1.033107	0.000132	0.058673	5.949483
445	0.000021	0.009492	1.032772	0.000126	0.056277	5.709898
446	0.000020	0.009487	1.031754	0.000127	0.056792	5.761619
447	0.000020	0.009487	1.031974	0.000127	0.056841	5.766308
448	0.000020	0.009501	1.033572	0.000132	0.059491	6.030774
449	0.000020	0.009549	1.038368	0.000127	0.057094	5.792515
450	0.000020	0.009494	1.033470	0.000127	0.057087	5.791635
451	0.000023	0.010524	1.136936	0.000132	0.059746	6.058030
452	0.000022	0.010503	1.135108	0.000127	0.057450	5.828591
453	0.000023	0.010569	1.141965	0.000127	0.057548	5.838158
454	0.000022	0.010521	1.136763	0.000127	0.057717	5.856009

Continued on next page

Table 1 – continued from previous page

	FPGA Hardware (s)			Python (s)		
Num Quads	**Per-Quad**	**Per-Scene**	**Per-Image**	**Per-Quad**	**Per-Scene**	**Per-Image**
455	0.000022	0.010490	1.135082	0.000132	0.060050	6.089244
456	0.000022	0.010542	1.140038	0.000126	0.057796	5.863415
457	0.000022	0.010505	1.136136	0.000127	0.058094	5.893896
458	0.000022	0.010513	1.138922	0.000132	0.060384	6.122124
459	0.000022	0.010504	1.135650	0.000126	0.058090	5.893018
460	0.000022	0.010493	1.135014	0.000127	0.058358	5.920591
461	0.000022	0.010482	1.134541	0.000126	0.058229	5.908249
462	0.000022	0.010531	1.139288	0.000131	0.060784	6.163768
463	0.000022	0.010539	1.140216	0.000126	0.058692	5.954690
464	0.000022	0.010513	1.135003	0.000126	0.058733	5.958800
465	0.000022	0.010453	1.131865	0.000127	0.059085	5.994174
466	0.000022	0.010482	1.135000	0.000131	0.061284	6.214365
467	0.000022	0.010522	1.139319	0.000126	0.058880	5.974315
468	0.000022	0.010497	1.137148	0.000126	0.059220	6.008362
469	0.000022	0.010510	1.135574	0.000132	0.061901	6.276656
470	0.000022	0.010529	1.140580	0.000127	0.059735	6.060192
471	0.000022	0.010526	1.139561	0.000127	0.059794	6.066250
472	0.000021	0.010491	1.137571	0.000127	0.060214	6.107812
473	0.000021	0.010508	1.138238	0.000131	0.062097	6.296454
474	0.000021	0.010507	1.138268	0.000127	0.060246	6.111291
475	0.000021	0.010528	1.141946	0.000127	0.060470	6.133311
476	0.000021	0.010480	1.136009	0.000131	0.062721	6.359868
477	0.000021	0.010504	1.138628	0.000127	0.060731	6.160471
478	0.000021	0.010487	1.136593	0.000126	0.060455	6.133191
479	0.000021	0.010591	1.148047	0.000127	0.060943	6.181852
480	0.000021	0.010538	1.143272	0.000132	0.063500	6.438611
481	0.000021	0.010484	1.137744	0.000127	0.061041	6.192112
482	0.000021	0.010523	1.140816	0.000131	0.063476	6.435016
483	0.000021	0.010504	1.139470	0.000127	0.061479	6.235876
484	0.000021	0.010497	1.138669	0.000132	0.063983	6.486115
485	0.000021	0.010506	1.141296	0.000127	0.061625	6.250923
486	0.000021	0.010499	1.140121	0.000132	0.064409	6.530193
487	0.000021	0.010508	1.141620	0.000126	0.061555	6.244279
488	0.000021	0.010527	1.143281	0.000126	0.061695	6.258690
489	0.000021	0.010506	1.140101	0.000126	0.061920	6.281150
490	0.000021	0.010530	1.144476	0.000126	0.061917	6.281550
491	0.000021	0.010539	1.144105	0.000131	0.064475	6.536079
492	0.000021	0.010543	1.144665	0.000127	0.062462	6.334733
493	0.000021	0.010554	1.147004	0.000131	0.064897	6.579078
494	0.000021	0.010497	1.140531	0.000127	0.062705	6.360735
495	0.000021	0.010525	1.143281	0.000131	0.065084	6.598429
496	0.000020	0.010497	1.140838	0.000126	0.062885	6.378224
497	0.000020	0.010507	1.142195	0.000131	0.065288	6.619090
498	0.000020	0.010548	1.145920	0.000127	0.063309	6.421553
499	0.000020	0.010522	1.143660	0.000127	0.063667	6.457360
500	0.000020	0.010548	1.146608	0.000127	0.063397	6.429039

Ray-Quadratic Intersection HLS Code

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#define MAX_QUAD 50

float calcInvSqRoot( float n ) {

   const float threehalfs = 1.5F;
   float y = n;

   long i = * ( long * ) &y;

   i = 0x5f3759df - ( i >> 1 );
   y = * ( float * ) &i;

   y = y * ( threehalfs - ( (n * 0.5F) * y * y ) );

   return y;
}

void dma_master_test(volatile float *arr, int num_quads){
        #pragma HLS INTERFACE m_axi port=arr depth=514 offset=slave
        #pragma HLS INTERFACE s_axilite port=num_quads
        #pragma HLS INTERFACE s_axilite port=return

        float buff[(MAX_QUAD * 10) + 14];
        float t_finals[MAX_QUAD] = { 0.0 };

        memcpy(buff, (const float*)arr, ((MAX_QUAD * 10) + 14)*sizeof(float));

        float px = buff[8];
        float py = buff[9];
        float pz = buff[10];
        float dx = buff[11];
        float dy = buff[12];
        float dz = buff[13];

        int x;


        for (x = 0; x < num_quads; x++) {
                #pragma HLS PIPELINE II=1
                #pragma HLS LOOP_TRIPCOUNT min=1 max=MAX_QUAD
                int mult = x * 10;
                float a = buff[14 + mult];
                float b = buff[15 + mult];
                float c = buff[16 + mult];
                float d = buff[17 + mult];
                float e = buff[18 + mult];
                float f = buff[19 + mult];
                float g = buff[20 + mult];
                float h = buff[21 + mult];
                float i = buff[22 + mult];
```

```
float j = buff[23 + mult];

float A = (a * dx * dx) +
                        (2 * b * dx * dy) +
                        (2 * c * dx * dz) +
                        (e * dy * dy) +
                        (2 * f * dy * dz) +
                        (h * dz * dz);

float B = 2 * (
                (a * px * dx) +
                (b * (px * dy + dx * py)) +
                (c * (px * dz + dx * pz)) +
                (d * dx) +
                (e * py * dy) +
                (f * (py * dz + dy * pz)) +
                (g * dy) +
                (h * pz * dz) +
                (i * dz)
        );

float C = (a * px * px) +
                (2 * b * px * py) +
                (2 * c * px * pz) +
                (2 * d * px) +
                (e * py * py) +
                (2 * f * py * pz) +
                (2 * g * py) +
                (h * pz * pz) +
                (2 * i * pz) +
                (j);

if (A == 0) {
        continue;
}

float disc = (B * B) - (4 * A * C);

if (disc < 0) {
        continue;
}

float ds = sqrt(disc);

float t0 = (-B - ds) / (2 * A);
float t1 = (-B + ds) / (2 * A);

if (t1 < 0.0008) {
        continue;
}

if (t0 > 0.0008) {
        t_finals[x] = t0;
        continue;
}
```

```
                else {
                        t_finals[x] = t1;
                        continue;
                }
        }

        int success_index = -1;
        float t_final = 1000000.1;

        for (x = 0; x < num_quads; x++) {
                #pragma HLS LOOP_TRIPCOUNT min=0 max=MAX_QUAD
                if (t_finals[x] < t_final && t_finals[x] != 0) {
                        t_final = t_finals[x];
                        success_index = x;
                }
        }

        if (success_index >= 0) {
                int mult = success_index * 10;
                float pox = (px + t_final * dx);
                float poy = (py + t_final * dy);
                float poz = (pz + t_final * dz);

                float n_x = (buff[14 + mult] * pox + buff[15 + mult] * poy + buff[16 + mult]
                * poz + buff[17 + mult]);
                float n_y = (buff[15 + mult] * pox + buff[18 + mult] * poy + buff[19 + mult]
                * poz + buff[20 + mult]);
                float n_z = (buff[16 + mult] * pox + buff[19 + mult] * poy + buff[21 + mult]
                * poz + buff[22 + mult]);

                float mod = calcInvSqRoot((n_x * n_x) + (n_y * n_y) + (n_z * n_z));

            buff[0] = t_final;

                buff[1] = pox;
                buff[2] = poy;
                buff[3] = poz;

                buff[4] = n_x * mod;
                buff[5] = n_y * mod;
                buff[6] = n_z * mod;
                buff[7] = success_index;


        }
        else {
                buff[0] = -1;
        }

        memcpy((float *)arr, buff, 8*sizeof(float)); // copies the answers into "out_buff" *only*
        return;
}
```