Ray Tracing Acceleration on FPGA

Varun Saxena[1], Santhana Bharathi Narasimmachari[1], and Sandilya Balemarthy[1]

[1]Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA
[1]{*vsaxena36, sbn6, vbalemarthy3*}*@gatech.edu*

I. Introduction

Kajiya, an influential researcher in computer graphics history once said:

"Ray tracing is not slow–computers are".

Through this project, we aim to incorporate hardware optimization techniques and parallelize components of the ray tracing algorithm. In order to appreciate the performance gains of optimizing ray tracing one must first understand it's critical position in the graphics pipeline. All graphics pipeline implementations loosely consist of the below five stages:

1) **Vertex Transform**: Transforms input vertex data via vertex and geometry shaders.
2) **Primitive Assembly**: Breaks down the object geometries to points, lines, and triangles.
3) **Rendering routine**: Determines whether a pixel of the viewport needs to be illuminated and the corresponding extent of illumination.
4) **Shader operations**: Determines the colour for a given fragment (out of fragments generated by the rendering routine).
5) **Frame buffer**: Stores the results of the shader operations and forms the final image seen on screen.

The rendering routine within the graphics pipeline can be further siloed into two stages.

1) **Visibility**: Determines if a point is visible at a particular pixel.
2) **Shading**: Determines the shader activity where a shader function acts on each pixel and applies the illumination corresponding to that pixel.

Ray tracing is an extremely time-consuming process as these two stages involve compute intensive shadow ray projections from primary ray and object intersection points to all light sources within the scene and then determining the illumination for each pixel.

However, it is an elegant technique in the sense that it mimics phenomena like reflection and refraction by relying on the same fundamental concept of light rays propagating and bouncing around before reaching the surface of the eye. Unlike an object centric algorithm such as rasterization that involves projecting primitive assembly triangles onto the screen and determining visibility by looping over all pixels, ray tracing is the closest technique to a perfect nature simulator as it simulates propagation of light rays and their collision with objects to perceive a 2D image from a 3D scene.

This uncanny similarity with nature is the reason it is paramount that today's machines are optimized to handle ray-tracing workloads in order to eliminate the currently existing trade off between rendering time and accuracy.

II. Problem Description

In this project, we break down the graphics pipelines into multiple steps and focus on accelerating the compute intensive parts of: primitive assembly, ray generation, ray intersection (sphere and triangle).

1) **Primitive assembly**: This step takes the object vertices as an input and generates triangle (primitive) edges. This calculation is done by forming matrices using the x, y, z and alpha components of the combinations of different vertices. The co-factors of each component form the edge equation coefficient. This process will be run in parallel to compute the 3 edges of the primitive and to generate multiple primitives in parallel.
2) **Ray generation**: This involves generating a ray equation from the eye view (or light source) and casting it on individual pixels. The framebuffer can be broken in multiple tiles and the rays for each of the fragment (pixel) can be generated in cast in parallel. Experimentation will be performed to compute the best tiling approach and tile size.
3) **Ray Intersection**: Multiple primitives will be read and stored in the primitive buffer. The rays which are cast will be evaluated against each primitive in the buffer to solve the visibility problem related to ordering the primitives in terms of closeness to the eye view. The intersection calculation can be pipelined to accelerate the computation per fragment.

Existing approaches use ASICs as an add-on block to GPUs for performing ray-tracing. With the recent advancements in graphics rendering many approaches use rasterization as the initial step to perform scene generation from the eye view and then ray tracing is used to render features from the light sources. This is done to accelerate the graphics rendering process by leveraging the benefits of both approaches. This also hides the compute intensive nature of ray-tracing by using it sparingly instead of using it for every ray source. An accelerated approach can be developed on FPGAs to leverage its highly parallelizable and configurable nature to

load the required algorithm i.e either rasterization or ray-tracing, based on the application as compared to consuming more area by having both even when they might not be needed together.

III. Existing Work

Various researchers have proposed architectures and design to perform ray tracing on the FPGAs. Jorg Schmittler et. al. build on the SaarCOR Hardware Architecture [6] to build a dynamic SaarCOR architecture and tested the functionality on FPGAs [7]. This architecture consisted of multiple dyanmic Ray Tracing Pipelines. The central ray-generation controller generates run configurations for each pipeline and supplies to the available ray tracer in order to allow load-balancing. The ray tracing pipeline is made up of multiple stages covering a re-usable transformation unit, traversal unit, intersecition unit, etc. They support their architecture using 6 memory bank, 2 as a double buffered frame buffer, 1 for shading data and 3 to store the KD-tree (for optimized traversal), object and triangle transformation matrices.

Joshua Fender and Jonathan Rose built a high-speed ray tracing engine [8] which they say is capable of outperforming a 2.4 GHz Pentium 4 running an optimized software ray tracing algorithm. For achieve this level of performance they use bounding volume hierarchy to reduce the number of intersections and primitive to check each cast ray against.

More recently, Yulin Chen et. al. developed an architecture in which they distribute the computation load between the FPGA with Hard Processor System (HPS) [9]. They used various HLS techniques to extract parallelism in the numerical differentiation methods and intersection algorithms. The HPS in their experimentation is a dual-core ARM Cortex-A9 CPU for the CycloneV FPGA. The HPS is used to read the model file, build the polgygon mesh, order the triangle to extract maximum FPGA parallelism and supplies the texture data. The processed data is then send to the FPGA which acts as a device for the host to perform the rendering using the ray tracing algorithm.

IV. Midterm Progress

We have implemented a Proof of Concept of Ray-Tracer in C++ and verified its functionality by rendering a classical teapot object from a .geo format input file that contains faces and vertices information.

First, we explored the .geo file format and built a parser that provided us the number of faces, vertices per face, normals, their respective co-ordinates and texture co-ordinates. Then, we converted the information to triangle primitives and defined a TriangleMesh class that encapsulates all attributes, transformation and intersection algorithms to form the primitive buffer. The ray tracer iterates over each pixel of the image and cast a ray through the center of each pixel forming the primary/cast ray from the camera to the object. The algorithm look for intersections between the cast rays and the fetched primitives. After identifying the intersection point, it calculates the surface texture properties using texture information processed from the .geo file. Finally, the rendered image is stored in a .ppm file format.

Fig. 1. Midterm C run output for ray tracer

Once the Proof of Concept was developed, the code was migrated to an Vitis HLS environment and necessary changes were made to make it HLS compatible as mentioned below:

- HLS allows only pointers to atomic data types. All pointers not adhering to this rule were removed.
- The code was split into Design IPs and testbench components.
- Compute intensive blocks like transformation, ray intersection are chosen as Design IPs.
- C++ Class member functions were migrated to a Vitis HLS friendly hierarchy.
- Multiple C++ porting issues were solved.
- Scene rendering was successful and matches with the CPU tested Proof of Concept results.

Currently, we are working on optimizing the runtime/latency of the complete ray-tracer by applying HLS optimization techniques related to buffer management through through either dataflow or ping-pong buffers, optimizing loop behaviour, etc.

TABLE I
Midterm Test Results for C code

Metric	**Result**
Window Height	128
Window Width	96
Field of View FOV	90
# Primitives	6320
C code execution time	75m 14.3s

The results of current test runs have been shown in table I. This shows that the unoptimized C code has a large run-time in the tune of hours. The resulting image is also shown in Figure 1.

```
6
4 4 4 4 4 4
0 1 2 3 0 4 5 1 1 5 6 2 0 3 7 4 5 4 7 6 2 6 7 3
-1 1 1 1 1 1 1 1 -1 -1 1 -1 -1 -1 -1 1 1 -1 1 1 1 -1 -1 -1 -1 -1
0 0 1 0 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 0 -1 0 0 -1 0 0 -1 0 0 -1 0 -1 0 0 -1 0 0
0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.
```

Fig. 2. Sample geo file for a cubical polygon mesh

V. Data Format

The primary input to our system is a geo formatted text file. This is a common file format in Computer Graphics that is used to represent a polygon mesh object. Objects are represented in terms of faces and faces are in turn represented by vertices. We can either represent the face by listing out each of its vertices or represent multiple faces by listing out the unique vertex points and the specific *unique* vertices to construct each face. The geo format represents faces in the latter fashion. As an example, consider the geo file representing a cube shown in Fig. 2.

1) The first line represents the number of faces of the cube.
2) The second line in the file represents the number of vertices for each face (face index array).
3) The third line is the vertex index array that represents the vertex indices used to construct each face (vertex index array).
4) The fourth line records all the unique vertices (vertex co-ordinates).
5) The fifth line contains the normal direction for each face (3 entries per face).
6) The last line contains the mapped texture coordinates for each of the above unique vertex.

VI. Ray Tracer Modelling

Our stated goal was to develop a ray tracing algorithm and accelerate it by implementing specific blocks on the Ultra96 FPGA and while doing so we developed a C++ model, converted into a HLS compliant C model and optimized it to the optimal resource usage and latency.

A. C++ modelling

We started of with creating a C++ model to replicate the tenets of ray tracing. The system starts with parsing the geo input file that contains the vertices of each face of a polygon mesh. The render module creates and initializes a frame buffer corresponding to each pixel in the image canvas. It then iterates over these pixels and creates a primary ray from the camera through the pixel and casts it onto the scene. The next step is to check for intersections of the primary ray with the triangles/primitive shapes present in the polygon mesh.

In case of a hit, we choose the primitive that is closest to the camera's line of sight as this would be the closest primitive visible to the user. Otherwise, we assume that the ray has hit the background of the scene. Finally, we use the hit color decided on the previously stated decision tree and fill the frame buffer corresponding to the primary ray with this hit color. We then perform the same with all primary rays sequentially to fill up all the frame buffers, thereby, making it possible to render a valid image.

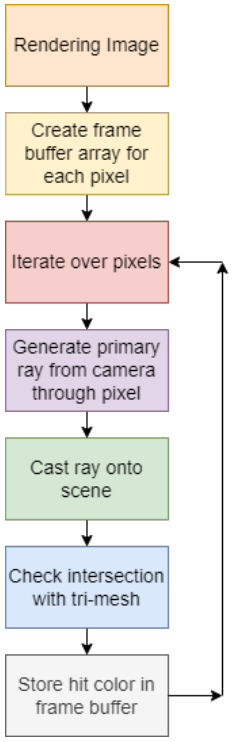

Fig. 3. Brief overview of ray tracing algorithm

B. C modelling

As mentioned in the previous section, we started of with a C++ implementation of ray tracing and tested it with a teapot polygon mesh. While developing the C++ implementation, we extensively used C++ features like classes and pointers to class objects in-order to represent vectors, 4x4 matrices used for transforms and polygon mesh objects. The next step was to make our implementation HLS compliant to proceed to the C-Synthesis stage.

To make the code HLS compliant, we performed the following changes upon our C++ implementation:

1) Partitioned code base into separate functions intended to be used as design IPs.
2) Included a custom *fixed_t* data type that could be toggled for floating point and fixed point.
3) Represent vectors and 4x4 matrices using simpler multi-dimensional floats instead of classes and OOP constructs that posed a hindrance for synthesis.
4) Replaced pointers to non-atomic data types with simpler constructs.
5) Removed complex classes used to store objects.

VII. High Level Synthesis (HLS) Architecture

The C model was modified to make it synthesizable through Xilinx Vitis HLS. We developed an architecture

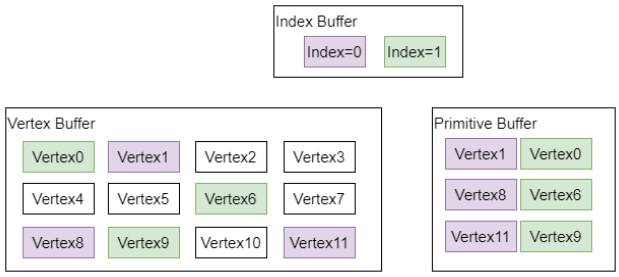

Fig. 4. Buffer Structure for the HLS

over the baseline before applying HLS optimization to ensure that the hardware architecture is able to leverage the available FGPA board resouces to the fullest. Developing the architecture before attempting optimizations gave us a better direction for the optimizations while precisely budgeting the resources to push the speed for lowest latency.

A. Data Reordering

The ".geo" file provides an index buffer and vertex buffer (shown in Fig. 4). A triangle index is generated between 0, NUM_TRIS-1, where NUM_TRIS is the number of triangles in the polygon mesh. The index buffer then provides the 3 indices to access the vertex buffer in order to get the 3 vertices which make up that primitive. The vertex buffer is designed in such a manner to reduce redundant data when the ray tracer is designed and run in software. But, in our hardware approach, this would lead to unordered access of the BRAM and DRAM which lead to under-utilization of the partition and reduce chances of DRAM burst respectively. So, we developed the primitive buffer, which is a reordered form of the vertex buffer. We perform this reordering on the host CPU. For this, it reads all the vertices which make up the primitive and store it in multi-dimensional arrays. This leads to multiple instances of the same vertex, but all the vertices of a primitive can be fetched directly using the triangle index and read as slices. This has 2 benefits: ensures the access to the buffer is ordered and does not need any access to another buffer like the index buffer. Such a structure, ensures the DRAM can burst effectively when writing the primitive buffer into the BRAM, the BRAM can be tiled effectively to be in-sync with the access pattern and also allows the possibility to effective tiling.

B. Architecture

We designed the architecture to ensure maximum utilization of the FPGA board resources and it helped us build an effective road map for the HLS optimizations. The architecture we develop is shown in Fig. 5. It is developed around the following aspects:

1) **DRAM**: The buffers are stored in the DRAM after the data reordering (mentioned in Section VII-A).
   - The primitive buffers, background color and cam2world matrix are then read into the BRAM. The Ultra96 has enough BRAM resources to store these buffers. Additionally, these buffers are accessed for each primitive check for each ray cast, leading to multiple accesses in each execution cycles, so it is advantageous to store these on board.
   - The texture coordinates buffer and frame buffer are kept in the DRAM as these are accessed once for each ray cast. So, the access cost is not very high in terms of the overall execution cycle.
2) **BRAM**: The BRAM stores the working set of all the data accessed multiple times in the execution cycles, namely, primitive buffer, background color and cam2world matrix.
   - The working set is copied into 2 more sets of arrays to create 3 copies overall in the BRAM. The 3 number is chosen due to resource constraints on the Ultra96 board. This is very important to ensure that the 3 planned cast ray pipelines can run completely independent of each other avoid conflicts which slow down the operation. Even though this would increase the BRAM utilization, but this is important to ensure completely independent execution of the computation lanes. These copies coupled with other optimization techniques listed below, result in a significant speedup of more than 20 times compared to the baseline.
   - The BRAM is partitioned to optimize the data fetch from each. The primitive buffer is partitioned to allow fetching all 3 vertices information in parallel. The other buffers are partitioned based on the loop ordering of where they are accessed. The same partitioning logic is applied to all copies of the buffers.
3) **Frame Coordinates Loop**: All the buffer data is passed onto the frame coordinate loops which loops over the X and Y coordinates of the frame buffer. It then calls a dataflow optimized function to cast 3 rays across the 3 coordinates selected along the X axis in parallel.
4) **Parallel Cast Rays**: The parallel rays cast are fed into independent castRay pipelines.
   - It generates the cast ray vector from the camera view source into the pixel coordinates (X, Y) supplied by the frame coordinates loop.
   - **Primitive Loop**: The cast ray is then checked for intersection against each primitive in the primitive buffer. For each intersection it tracks the intersection distance from the camera to find the nearest intersection as it would form the top visible point. It then passes the hit point back to the frame coordinates lop.
5) **Texture**: For the nearest intersection, it accesses the texture coordinates buffer using the intersection point and finds the texture to be applied to that point based on the primitive it intersects. The loop then passes the final pixel value vectors to the frame buffer in the

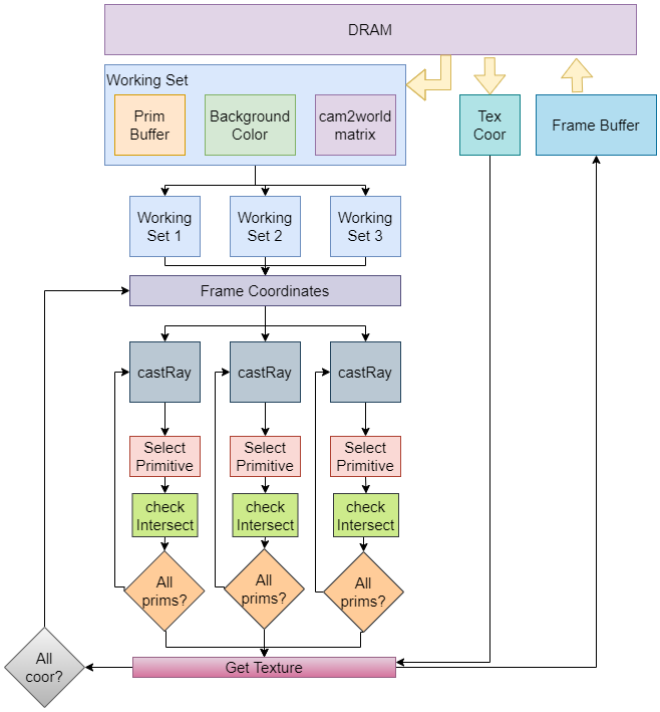

Fig. 5. HLS Architecture

DRAM.

Other micro-architectural optimizations are:

1) **Optimized Square root**: The default square root function included in the HLS math library did not support fixed point and was not optimized for uniform latency and performance. Since computing square root is an important part of the ray triangle intersection calculations, optimizing the square root should result in a reduction of the latency.
2) **Loop unroll**: To improve the performance of fundamental math operations such as matrix multiplication, cross product and dot product used often in the ray tracing, we have unrolled the loops in them. But unrolling loops containing lot of mathematical expressions such as division or modulo operations increased the DSP utilization, so we restricted the unrolling to functions that can keep the resource utilization under control.
3) **Pipelining**: Various loop were automatically pipelined by Vitis HLS. This initially led to a blow in resource utilization. Then, we performed a study of each loop being pipelined and made changes to allow the auto-pipelined loops to run without conflicts or II violations. One of main loops being pipelined is the loop which checks for intersections across the different primitives. This allows the primitive intersection checks to have a higher throughput.
4) **DSP usage and Data width reduction**: When the HLS synthesizable ray tracing was implemented with variables using *float* data type, the DSP utilization was very high even without any additional unrolling or optimizations. In order to reduce this high DSP utilization, we changed *float* data type to *fixed point* data type of width (48, 16). But even with *fixed point* data type, once other optimization techniques such as pipe-lining, unrolling and tiling were added, DSP utilization went beyond limits once again. Then we experimented with different *fixed point* data types and finally settled with (32, 16) bits configuration where the accuracy of the image data is preserved as well the resource utilization is under the limits.
5) **Dataflow** After changing the data type of variables used to *fixed point* of (32, 16) configuration, we had enough room for further optimizations. So we explored the task level parallelism (dataflow) for casting rays. With the available resources, we calculated that three rays can be casted simultaneously and can be pipelined using HLS dataflow pragma. To achieve this, we have made duplicate copies of arguments required to find intersections such as primitive buffers and made every cast ray independent of each other. This resulted in a huge boost for reducing the rendering latency.

VIII. Ray Tracer Transform

The ray tracer also supports matrix transforms which allow changing the camera to world and object to world matrices. The object to world matrices changes the orientation of the object upon read operation, whereas the camera to world matrix is sent via the DRAM to the FPGA which applies it which is used to transform the cast ray to change the view point. This allows modifying the object by rotating, flipping, etc. These changes are automatically handled by our architecture and the user would only need to change these above mentioned matrices in the host CPU testbench. The results are shown in Fig. 6. This makes the code more generic and also maintains the same latency and resource usage for any type of transform.

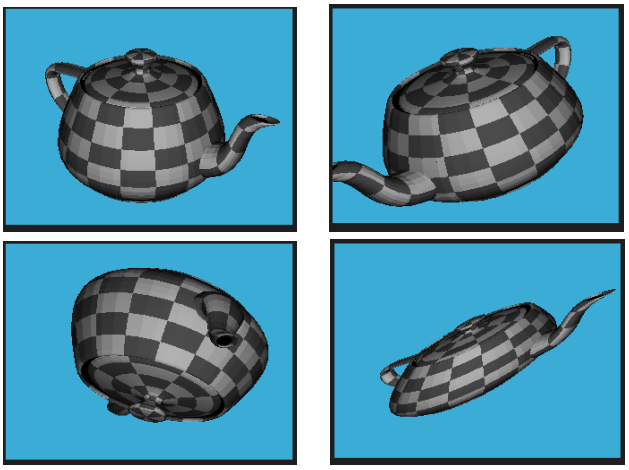

Fig. 6. Different transformations of the object generated from the ray tracer

IX. Design Choices / Challenges

1) **Square root math function**: Ray tracing and intersection calculations involve vector and math operations

TABLE II

OPTIMIZATION TECHNIQUES

Optimization	Latency	Speedup	BRAM_18K	DSP	FF	LUT
Floating point	2400s	-	0%	52%	25%	33%
Base: Fixed point	97.33s	-	0%	8%	9%	26%
Store in BRAM Array Partition #1	39.37s	2.47x	15%	25%	29%	61%
Optimized sqrt Targeted loop unroll Increased precision	6.651s	14.6x	23%	136%	22%	42%
Targeted loop unroll Array partitioning #2	4.993s	19.5x	31%	142%	19%	37%
Inlining lib functions Reordering BRAM structure DSP usage: Removed modulo DSP usage: Removed unrolling Array partitioning #3	5.012s	19.4x	46%	93%	23%	42%
Reduced data width Added techniques for sqrt and div for handling for precision loss	4.964s	19.6x	32%	34%	15%	26%
Dataflow optimization Parallel casting of 3 rays	1.665s	58.46x	91%	92%	31%	52%

```
import numpy as np
from pynq import MMIO
import pynq
import data
from datetime import datetime

resultInteresectionIndex = 5180
overlay = pynq.Overlay('intersect.bit')
top_ip = overlay.rayTriangleIntersect_0
top_ip.signature

dirptr = data.dir_Buffer.physical_address
p1ptr = data.P1_Buffer.physical_address
p2ptr = data.P2_Buffer.physical_address
p3ptr = data.P3_Buffer.physical_address
resultptr = data.result.physical_address

top_ip.write(0x10, dirptr)
top_ip.write(0x1c, p1ptr)
top_ip.write(0x28, p2ptr)
top_ip.write(0x34, p3ptr)
top_ip.write(0x40, resultptr)

top_ip.write(0x00, 1)
isready = top_ip.read(0x00)

# Time the kernel execution
exe_start = datetime.now()

while( isready == 1):
    isready = top_ip.read(0x00)

exe_end = datetime.now()
time_diff = exe_end - exe_start
print("Execution time", time_diff.microseconds, "us")

if data.result[0] != -1:
    print("[SUCCESS] Ray Intersection Found!!!!!")
    calculatedIntersectionIndex = data._to_int (data.result[0])
    print("Calculated Intersection Index: ", calculatedIntersectionIndex)

    if calculatedIntersectionIndex == resultInteresectionIndex:
        print("IP Verification: PASS")
    else:
        print("IP Verification: FAIL")
else:
    print("[FAIL]: Intersection not Found :(")
```

```
Execution time 843 us
[SUCCESS] Ray Intersection Found!!!!!
Calculated Intersection Index:  5180
IP Verification: PASS
```

Fig. 7. IP Functional verification on Pynq FPGA

TABLE III

TEST METRICS

Metric	Result
Window Height	320
Window Width	240
Field of View FOV	20
# Primitives	6320
Floating point C sims runtime	21.43 sec
Fixed point C sims runtime	12m 17 sec
Fixed point synthesized latency	1.665 sec

where normalizing the vectors is a crucial step in the process. In order to normalize a ray vector, we used the default fixed point square root function provided by the hls math library. But because of the way square root calculation was implemented, the C synthesis report had a variable latency with a significant difference between the minimum and maximum latency.
In order to resolve this issue, we explored alternate methods and algorithms to implement square root function and finally settled with the CORDIC based fixed point square root implementation provided by Xilinx in their example code. Upon using the CORDIC algorithm the gap between minimum and maximum latency reduced significantly. The CORDIC algorithm uses a digit by digit approach to achieve the precision and it runs for a fixed number of iterations ensuring similar execution time for computing square root of any number.

2) **Data Precision**: To reduce the BRAM and DSP utilization we had to reduce the fixed point data width from (48, 16) to an optimal data width configuration where the image quality is not compromised. As we had to find the right trade off between accuracy vs. resource utilization, we experimented with different fixed point width configuration and finally settled with

(32, 16). With the reduced precision for fractional portion, smaller fractional numbers which were earlier considered non-zero became zero, now creating divide by zero exceptions during calculations. To overcome the issue without increasing the data width, we used a custom wrapper divide function which checks for non-zero divisor before actually performing the division operation.

3) **DSP Usage**: With excessive use of HLS unroll pragma in all custom math operations, the DSP utilization went beyond 100%. So to bring the DSP utilization under control, we selectively removed unroll operations from least used non-intensive compute functions and converted modulo operations into look-up tables. This helped reducing the DSP utilization by nearly 10%.

4) **BRAM Ports**: To cast multiple rays simultaneously and compute intersections with the primitives in the scene using HLS dataflow (task level pipeline), the data used in each task for computation must be independent of each other. This includes the primitive buffer of size 6320 (Number of primitives) * 3 (Number of vertices) * 3 (Co-ordinates) also to be duplicated and provided as an individual input to each cast ray. With increased number of copies of a larger data structure, the partition level could not be kept at complete partition level anymore. So we had reduced the partition to dimension 2 alone for the primitive buffers.

5) **BRAM vs DRAM Usage**: Even though copying the input data from DRAM to BRAM is a common practice in FPGA accelerators, it might not be advantageous in all scenarios and it requires a case by case decision. For the ray tracing accelerators, the inputs are primitive buffers, background color, transformation matrix, and outputs are texture and output frame buffer. Out of these the texture and frame buffer are accessed only to write the final computed value of the pixel and do not involve in frequent access patterns. So copying them from DRAM to BRAM and writing them back to DRAM will take more cycles than keeping them in DRAM itself and writing the final values to them only once. So, we chose to copy primitive buffer, background color and camera to world transformation matrix alone to BRAM and partitioned them for further computations.

X. Evaluations

For testing and synthesis we developed a completely automated command line flow to increase productivity. We created shell scripts to run floating point and fixed point simulations by changing a config header file. The top level Makefile is configured for multiple options and generates the execution time report of the code. Additionally, we created a TCL script for the synthesis run by a shell script which configures the synthesis to run with the fixed point configuration. All of these features made development more streamlined.

For evaluating the ray tracer design for functionality and performance, we used different methods including running a HLS fixed point C simulation, synthesizing the complete design for Ultra96, generating a FPGA bit stream of ray intersection algorithm and running it on Pynq FPGA. For synthesis we used the Xilinx Vitis HLS 2021.1. The design runs on 100MHz clock (based on the FPGA clock). With all optimizations as listed in Table II, we are able to reduce the rendering latency for an image of dimensions 320 x 240 pixels with 6320 primitives from 97.33 seconds to 1.665 seconds at a speedup of 58.46x. Whereas, focusing only on the ray and triangle intersect calculation (most compute intensive portion of the ray tracing algorithm), we are able to identify intersection of any cast ray with the provided triangle vertices of the object to be rendered within 843 microseconds. This value when scaled up for a whole image becomes comparable to that of image rendering using rasterization technique.

The architecture mainly requires the BRAM and DSP in higher numbers. So, the design bottleneck is the number of available BRAM and DSP on board. Their utilization scales linearly with the increase in the number of castRay lanes. Thus, we see 3x increase in BRAM and DSP usage when moving from a single castRay lane to 3 lanes. The FF and LUT also scale by 2x. We noticed that the DSP usage was very high when the precision was kept high and using the existing sqrt and divide functions from the HLS math library. Using the custom division and CORDIC algorithm based sqrt function allowed use to handle lower precision issues and bring down the DSP usage significantly as seen from row 7 in II. The auto-unrolling and auto-pipelining for our custom functions performed by Vitis blows up the DSP usage so we performed targetted study to unroll and pipeline to optimize the resource usage and keep the utilization within limits.

XI. Results

Listed above in Table. II are the optimization techniques we tried and their corresponding latency, resource utilization and speedup gained from the baseline. The major boost in speedup came when the unroll was implemented to all fundamental math functions, but the speedup gain came at the cost of resources. As discussed earlier in the Section VII-B, the reduction in data width helped meet the resources in the FPGA of our choice and provided us with enough room for exploring other optimizations. With the reduced fixed point data width, and a reduced precision for intersection detection, the number of cycles required to detect intersection were brought down. In addition to this, the parallelism introduced at the cast ray level helped in linearly scaling down the latency to achieve a speedup of 58.46x.

XII. Conclusion

We have successfully developed a ray tracing core for the project. We have developed a HLS architecture for the

ray tracer core which allowed for successful optimizations at different levels of granularity. We have met all the milestones discussed in the project proposal and met the plan/design goals set forth during the same including "primitive assembly", "ray generation" and "ray intersection". We have achieved a 58.46x speed from the baseline (hardware) synthesized fixed point implementation latency, 1442x speed up compared to the synthesized (hardware) floating point implementation latency, 13.1447x compared to the C run on the host CPU (software) with floating point and 442x compared to the C run on the host CPU (software) with fixed point. We achieved these target by keeping the resource utilization within the constraints of the Ultra96 board and efficiently used the resources in our design. The design supports transformations (rotations, zooming, etc.) within the same latency and allows moving to larger frame sizes with the same resource usage. This makes our design robust and generic. The proposed architecture can be used as cores in larger design for achieve the speedup.

XIII. Future Work

The following updates can be made to advance the design further:

1) Testing on a larger board with more resources (Xilinx U50).
2) Developing an updated architecture to allow sharing copies between different castRay lanes to ensure scalability.
3) Performing design space exploration (DSE) by checking scalability across different levels of granularity, namely, primitive level parallelism, lane level parallelism and core level parallelism.
4) Develop a bounding volume hierarchy wrapper on the intersection unit to reduce the number of intersections to check and reducing the overall latency of generating each frame.

References

[1] Michael F. Worboys (30 October 1995). GIS: A Computer Science Perspective. CRC Press. pp. 232–. ISBN 978-0-7484-0065-2.

[2] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. 2002. Ray tracing on programmable graphics hardware. ACM Trans. Graph. 21, 3 (July 2002), 703–712.

[3] Pitkin, Thomas A.. "GPU ray tracing with CUDA." (2013).

[4] Gunther, J.; Popov, S.; Seidel, H. P.; Slusallek, P., "Realtime Ray Tracing on GPU with BVH-based Packet Traversal", IEEE Symposioum on Interactive Ray Tracing, 2007, pp. 113-8.

[5] Moon, B., Jun, J.Y., Lee, J., Kim, K., Hachisuka, T. and Yoon, S.-E. (2013), Robust Image Denoising Using a Virtual Flash Image for Monte Carlo Ray Tracing. Computer Graphics Forum, 32: 139-151. https://doi.org/10.1111/cgf.12004

[6] Schmittler J., Wald I., Slusallek P., "SaarCOR - A Hardware Architectrue for Ray Tracing", Proceedings of the ACM SIGGRAPH/Eurographics Conferences on Graphics Hardware (2002), pp. 27-36

[7] Jorg Schmittler, Sven Woop Daniel Wagner, Wolfgang J. Paul, Philipp Slusallek, "Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip", Proceeding of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (2004), pp. 95-106

[8] J. Fender and J. Rose, "A high-speed ray tracing engine built on a field-programmable system," Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798), 2003, pp. 188-195, doi: 10.1109/FPT.2003.1275747.

[9] Y. Chen, R. Yin, B. Gao, L. Peng and M. Gong, "Ray Tracing on Single FPGA," 2021 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC), 2021, pp. 1290-1294, doi: 10.1109/IPEC51340.2021.9421209.

[10] https://www.scratchapixel.com/lessons/3d-basic-rendering/transforming-objects-using-matrices

[11] https://github.com/Xilinx/Vitis-HLS-Introductory-Examples/tree/master/Modeling/fixed_point_sqrt