ACCELERATING RAY TRACING ON FPGA

By

AJAY SIMHA MODUGALA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2017

To my Parents and Sister

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Greg Stitt for giving me this opportunity to work on this project. I would also like to thank Rasmus Tamstorf from Walt Disney Animation Studios for making this project possible. I thank my sister and my parents for their support and encouragement. I thank my friend and partner in this project Abhay Raj Kumar Gupta for his contributions and continued support.

TABLE OF CONTENTS

page

LIST OF FIGURES

LIST OF ABBREVIATIONS

BVH	Bounding Volume Hierarchy
BVI	Bounding Volume Intersection
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
HPC	High Performance Computing
RTI	Ray Triangle Intersection

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ACCELERATING RAY TRACING ON FPGA

By

Ajay Simha Modugala

December 2017

Chair: Greg Stitt
Major: Electrical and Computer Engineering

Computer graphics are an important component of visual media like architectural models, movies, games and websites. An essential step in presenting computer graphics is rendering 3D models into a 2D image. There are two rendering techniques namely rasterization and ray-tracing.

The rasterization technique is based on projecting the model onto the screen by simple trigonometric transforms. It is fast and can render in real-time. Using tricks, some aspects of images like shadows can be computed in rasterization. But accurate shadows, reflections, refractions cannot be produced by rasterization, which limits the realism of the rendered image.

Ray tracing, on the other hand can perfectly emulate the light transport behavior that is observed in nature. But this process is computationally heavy due to the vast number of calculations involved in finding the ray scene intersections. To make matters worse, the ray traversal algorithms have low memory coherency between rays, making the problem challenging for parallel compute models. This thesis describes the development of architectures for an FPGA based accelerator to speed up the ray-triangle intersection and the BVH traversal steps in ray tracing.

CHAPTER 1
INTRODUCTION

FPGAs as an Accelerator for HPC

High performance computing (HPC) deals with computing and processing large amounts of data as fast as possible. Scientific computations, simulations and ray tracing are some examples of high performance computing. A HPC problem can be approached by breaking down the problem into smaller bits and distributing the work load across many computers. While this approach gives good results and is quick, it is not necessarily the most power efficient or the fastest.

An alternative is to use Hardware accelerators for the same task, like GPUs or Compute accelerators like the Xeon PHI. These accelerators add to the compute power of the PC while keeping the power utilization low. But they are usually limited in the compute models, often working as SIMD processors or vector processors. This expects that the HPC problem is made up of similar tasks operating on different data, like vector addition. But their benefit is lost in situations where the subtasks have differing behaviors.

Another solution is to augment the features of the processor by using specialized hardware. One way of doing this is using FPGAs. The hardware architecture on the FPGA can be based on SIMD architecture or a deep pipeline or many asynchronous processors, respecting the FPGA size constraints. This means that the subtasks don’t need to have the same behavior. Further, FPGAs can be reconfigured to add or modify functionality, keeping the HPC solution flexible. Bonus benefits of using FPGAs are that they have significantly lower power draw in comparison to GPUs and Processors, without sacrificing on performance.

Ray Tracing

When working on 3D graphics, artists begin with designing the 3D scene in a modelling software by defining the shape of the object, its position and the position of lights and the viewing angle. The objects in the scene can have arbitrary complexity but are made of many primitive objects and shapes like triangles and spheres. Once the scene is modelled, the next step is to convert the 3D model into a 2D image, which is better suited for viewing on a screen. This process is termed rendering.

Rendering is done either by rasterization or by ray-tracing. Rasterization is based on projecting the primitives onto the screen. This technique requires a stream of triangles and is suited for Single Instruction Multiple Data architectures like GPUs. Rasterization is capable of real-time frame rates but the trade-off is that light transport phenomena is not accounted for. This implies that optical effects like soft shadows, reflections and caustics are difficult to produce using rasterization. Ray-tracing, on the other hand, models the light transport phenomena to simulate these optical effects to obtain photorealistic images.

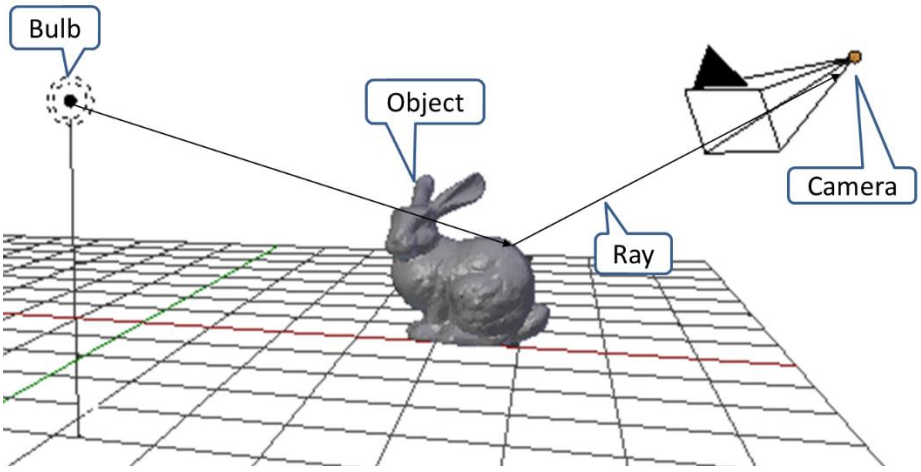

Figure 1-1. Ray Casting

Early ray tracing algorithms were developed in 1968 by Arthur Appel. In his algorithm, Arthur traced the ray paths by shooting rays from a view point through every pixel of the screen into the scene. The closest object intersected by the ray and the corresponding intersection point is recorded. The pixel is then shaded using the material properties of the object that the ray intersects. This technique is known as ray casting.

Ray casting only deals with rays from the view point, known as primary rays. Ray tracing improves on this technique by accounting for the reflection, refraction and absorption properties of surfaces by recursively bouncing the rays off objects, shooting secondary rays from the intersection points of the primary rays. The secondary rays can also be reflected and refracted in a similar manner. The pixel value is then computed as the sum of all the intersections seen by the primary ray and its secondary rays. This allows for the computation of accurate shadows, reflective and refractive effects and illumination from multiple sources. This recursive process is known as Whitted ray tracing.

Whitted Ray Tracing

The minimum requirements to make a rendering are the 3D scene, the camera, the screen and light sources. These components are fed into the raytracing algorithm to make the 2D rendering. The key concept implemented in ray tracing is the light transport phenomena. This includes the emission of light (photons) from sources, light's interaction with the objects in the scene and finally its capture by the camera. The naïve way of modelling this is to cast many rays from the light sources and recursively trace the rays until they meet the screen. But this is wasteful as most of the rays would never hit the screen. The alternative, is to trace the rays from the camera through the screen, also known as back tracing. This is the governing idea of Whitted ray tracing.

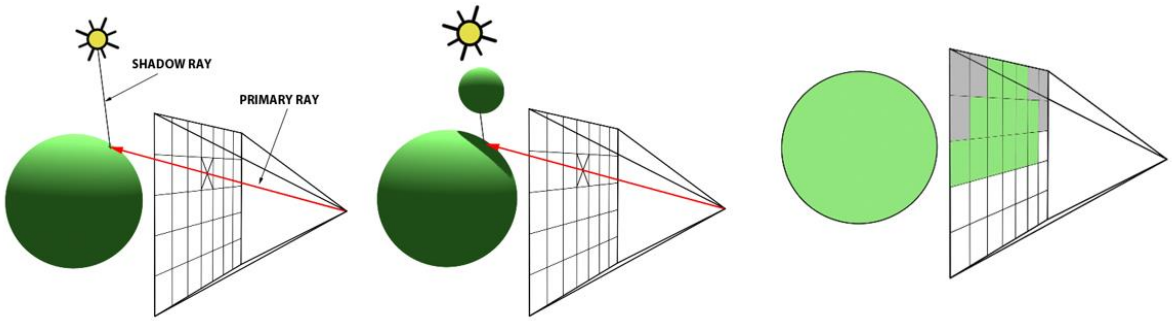

Figure 1-2. Whitted Ray Tracing

The rays originating from the camera are known as primary rays, and they are traced to find the nearest objects in their path. After the intersection point is computed, the secondary rays are computed. They may be reflected rays, refracted rays or shadow rays.

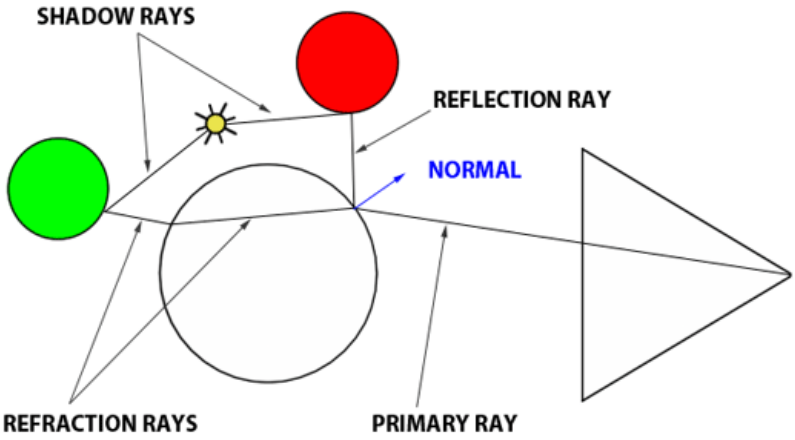

Figure 1-3. Different types of rays

CHAPTER 2
COMPONENTS OF RAY TRACING

While rendering a scene, two things need to be considered. First, the objects visible to the camera must be identified. This translates to finding the nearest unobstructed objects to the camera. This is known as the visibility problem Secondly, the color of the objects must be determined, which is a function of both the material of the object and the viewpoint. This computation is known as shading.

Approaching the Visibility Problem

The visibility problem focuses on determining the nearest intersection between the ray and the 3D scene. To perform this, the scene needs to be understood. An point in the scene is represented using 3 floats. Connecting coplanar points together produces polygons. The polygons can be considered as surfaces, which can be put together to form more complex objects. Surfaces can also be created using functions for spheres, Bezier curves etc. The polygons, spheres and curves mentioned are known as primitives, as they can be used to build more complex objects.

Primitives are chosen to suit the object being modelled. Smooth surfaces call for spheres and Beizer curves, while most objects can be broken down into coplanar polygons. One type of polygon, the triangle, is a commonly used primitive because it is always coplanar and mathematically easy to work with.

The visibility problem, which focuses on determining the nearest ray scene intersections, is addressed by testing each primitive of the scene with the ray and returning the nearest intersection if any. The intersection tests for a ray and a sphere, a ray and a box, a ray and a triangle are well defined and computationally simple. But intersecting a ray with Bezier curves is significantly slower. An optimization to beizer

curves is to break down the surface of the curve into triangles, which are far easier to intersect with.

But determining the point of intersection is still a tough task because the ray triangle intersection consists of may floating point operations, and this routine needs to be executed over many triangles for each ray, which is compute intensive.

Shading the Pixels

Solving the visibility problem allows flattening the 3D scene by picking the visible objects. But this only gives the silhouette of the objects. The next step is to give the right color and texture to these objects. In real life, the appearance of an object is dictated by the way the material of the object interacts with light. A mirror like material reflects all the light that is incident on it, a transparent object allows light to pass through it and refracts it and a colored object selectively reflects some part of the light spectrum and absorbs the rest. In the case of translucent or foggy objects, the light that is incident is diffused in all directions. For shading, it is imperative that all these phenomena are modelled.

Shading is done by computing the total light that falls on the point on the surface intersected by the ray. The light incident on this point can be from light sources and light that is reflected off other surfaces in the scene. To find the contribution of light from the light sources, a ray is cast to the source and the light contribution is computed. But to get a measure of the light reflected off other surfaces, many secondary rays need to be cast from the first point of intersection to these surfaces. Further, to get the amount of light that these surfaces reflect, the total light incident on these surfaces must be known. And the same steps of spawning more secondary rays from the corresponding

intersection points is repeated. This process is recursive. Once a preset level of recursion is attained, the pixel is shaded by adding all the light incident on a point.

This procedure of spawning new rays being recursive, and the calculation of the angle of reflection and refraction following Snell's laws and Fresnel equations makes the ray tracing process slow and compute heavy. This calls for the need to speed up the time needed for calculating the ray scene intersections.

Reducing Render Time Using Acceleration Structures

While using ray tracing to solve the visibility problem, an important step is to find the nearest intersection points of the rays and the surfaces in the scene. Because the surfaces in a scene are primarily made of primitives like triangles, this step reduces to finding the nearest intersection points of the rays and the triangles in the scene. The naïve way to find these intersection points is to test each ray against each triangle in the scene and store the nearest intersection points. But this solution has scalability issues as a scene can have between thousands to millions of triangles which need to be tested against millions of rays, leading to very large number of Ray Triangle Intersection (RTI) tests. With each RTI test being 51 floating point operations, the number of RTI tests done must be reduced.

The solution to this problem is to divide the scene space into subspaces, and test if an incoming ray intersects with the boundaries of the subspace, known as the bounding box. If the ray misses the bounding box, then there is no need to test against the objects in the bounding box, thus culling down number of tests needed to render the image drastically. This idea of quickly rejecting large groups of triangles which definitely won't be touched by the ray is the underlying principle of acceleration structures. Formally, as acceleration structure is a data structure that accelerates ray tracing. Many

types of acceleration structures like Grid based acceleration structure and Bounding volume hierarchy have been in common use.

Organizing the Scene into a Grid

Akira Fujimoto proposed an acceleration structure which was based in dividing the entire scene into a uniform grid, rather than bounding the objects with irregular bounding volumes. The cells in the grid are then marked as containing part of the objects or not. The cells that re not empty are associated with a list of objects that are contained in the cell. This way the scene is uniformly broken down into cell sized chunks. Once the grid is formed, a ray is made to traverse the grid. The grid traversal is extremely efficient as the intersection points of the grid cells and the ray occur at regular intervals.

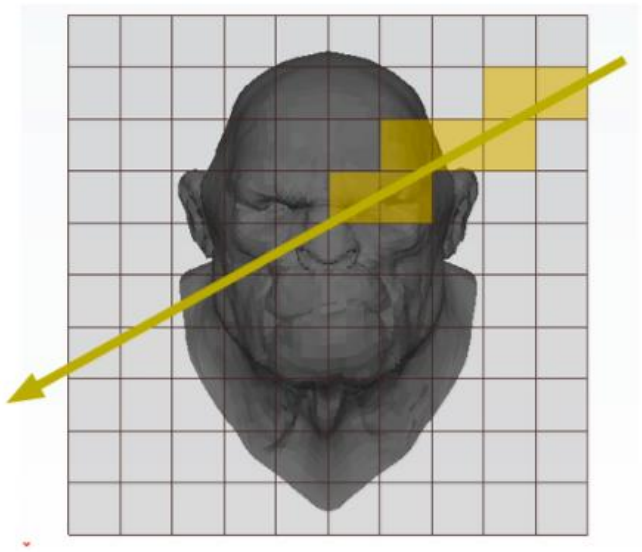

Figure 2-1. Grid acceleration structure

While construction the grid, the grid resolution is an important parameter that needs to be considered. Once the size is chosen, the next task is to assign the triangles to the cells. This is done by comparing the bounding volumes of the triangles and the grid. The grid cells that overlap with the triangle are marked to possibly contain the

triangle. To avoid making multiple copies of the triangles, only a pointer to the triangle mesh is associated with the grid cell. It is possible for a triangle to be referenced by multiple grid cells, leading to repeated tests of a ray with the triangle. This can be avoided by using mailboxes for each triangle which track the rays that intersect with thw triangle and cache the intersection data for fast retrieval when the ray is tested again with the triangle.

Organizing the Scene into Bounding Volume Hierarchy

The Bounding Volume Hierarchy (BVH) acceleration structure, introduced by Kajia and Kay, is based on tightly bounding objects in a volume so that the number of wasted RTIs will be reduced to a minimum. To keep the tests between the rays and the bounding volumes simple and fast, the bounding volumes are made by 6 axis aligned planes which contain the surfaces of the bounding box. So the number of ray triangle intersections are reduced and replaced by cheaper ray bounding box intersection tests. While this idea speeds up the RTI tests over all, computation time is still linearly dependent on the object count.

An improvement to this approach is to bound smaller bounding volumes in lager bounding volumes, which could be first tested for intersections before the lower level bounding volumes are tested. By organizing smaller bounding volumes into lager bounding volumes in a tree like structure, the Bounding Volume Hierarchy(BVH) is created. Practically, the BVH is built in a top down fashion. The scene specially divided using heuristics that ensure that each part has a similar object count or area. Then these parts are recursively divided in the same manner until the number of objects in a bounding volume is less than a threshold.

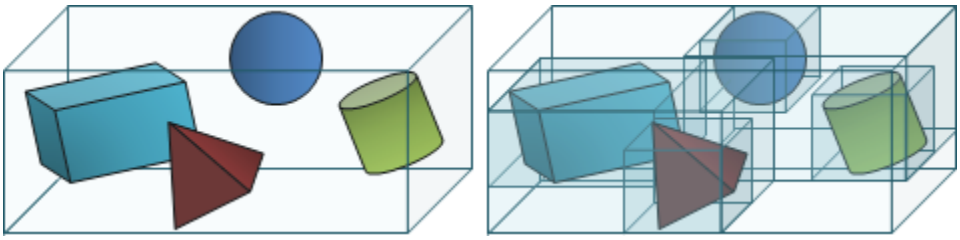

Figure 2-2. Bounding Volume Hierarchy

While grid acceleration structures are faster than BVH acceleration structures, they assume uniform object distribution which is not always the case. Additionally, the grid acceleration structure tends to be memory intensive as well. This makes BVH a more suitable choice for acceleration.

CHAPTER 3
ACCELERATING RAY TRACING

What Makes Ray Tracing Slow?

The performance of software is dependent on the performance of its individual parts, and Ray Tracing is no exception. Ray tracing consists of two major parts, one being the ray scene intersection and the other being the shading. The ray scene intersection is slow because it involved computing the ray triangle or bounding volume intersections billions of times to render a scene. This is further acerbated by the fact that most of the ray object intersections result in poor memory coherence, which further reduces the performance of ray tracing. The second part of ray tracing involves applying Fresnel equations and Snell's law at the intersection points, spawning shadow rays and in the final step combining the light contributions from various sources to the pixel to compute the pixel intensity.

Existing research notes that the ray scene intersection step is the slowest step in ray tracing, followed by shading. Therefore, accelerating is the ray scene intersection step has a greater contribution towards reducing the render time, and is the first logical step towards accelerating ray tracing.

Preliminary Studies on Ray Scene Intersection

The ray scene intersections consist of two parts which are the ray triangle intersection tests and traversing the bounding volumes by the rays. Because these two parts have highly differing functions, two unique hardware architectures were created for each function. In order to make good design choices for these architectures, experiments modelling different architectures were conducted.

Experiments for the Ray Triangle Intersection Architecture

The RTI tests performed in a scene have coherent data flow, i.e. undergo the same number of computations for all the rays and triangles. While this makes the problem embarrassingly parallel and easy to accelerate, the data access for each ray was not coherent. This made memory accesses a major bottle neck in our architecture. The solution to this problem was adapt reusing the data that had been drawn into the FPGA. This was done by choosing an architecture with many parallel RTI pipelines, each of which were fed by a unique ray, and all of which received the same triangle. Thus, instead of accessing a separate triangle for each pipeline, the same triangle was broadcast to all pipelines. But this is only possible if all the unique rays feeding into the pipelines needed to be tested against the same triangles, which happens if all the rays intersect the same child node. Further, to keep all the parallel pipelines busy, a minimum number of parallel rays are required to intersect a bounding volume. If many rays were to intersect a bounding volume, then many rays could be scheduled in parallel, leading to higher throughput.

To increase the number of rays intersecting a bounding box, then number of triangles in the leaf nodes was increased. This led to two lessons, firstly, leaf nodes with higher object count incurred increased RTI tests, also known as wasted RTI tests. Secondly, leaf nodes with higher object count led to shallower BVHs, which were faster to traverse. So higher parallelism is obtained by doing extra RTI tests, but reducing the burden on BVH traversal.

The first experiment was to study the average pipeline utilization assuming infinite RTI pipelines and varied leaf node size. For this experiment, A typical scene was taken and its BVH was constructed with varying leaf node size. Then the sece was

rendered and the number of rays intersecting a leaf node was recoded. The average number of rays intersecting a leaf node was taken to be an indicator of pipeline unitization. Plotting this data revealed a linear relationship between the leaf node size and pipeline utilization.

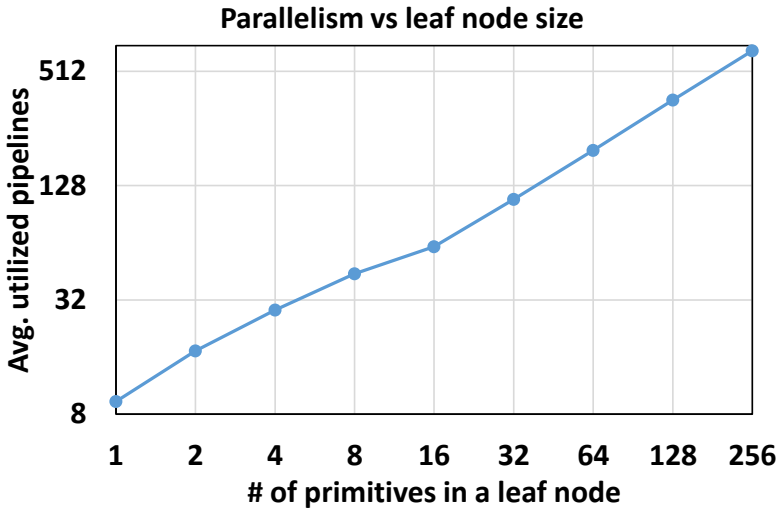

Figure 3-1. Available parallelism vs the leaf node size for a hypothetical architecture

The second experiment was to study the number of wasted RTI tests performed while increasing the leaf node size. This was done by utilizing counters in the rendering code. This resulted in a linear increase in the number of wasted RTIs performed.

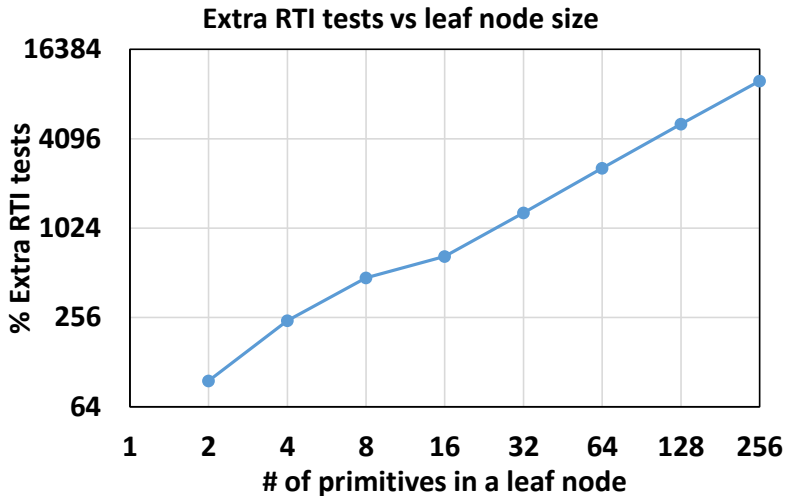

Figure 3-2. Increased wasted RTI tests with leaf node

As seen, with increase in the leaf node size, we see an increase in the wasted RTIs but also see an increase in parallelism.

Finally, a projection of the render time vs the number of pipelines was plotted to study the benefits of having more pipelines.

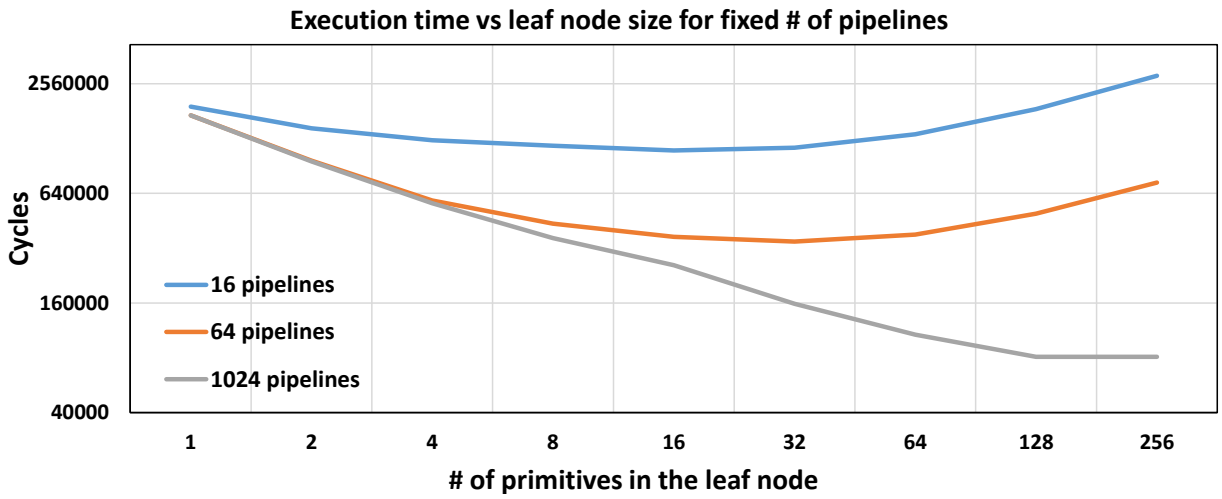

Figure 3-3. Execution time vs leaf node size for the hypothetical architecture

This led to the idea that for a given number of pipelines, leaf node objective count ranging from half to the same number of pipelines led to the fastest render time. Beyond this point wasted RTIs would slow down the system.

Experiments for the Bounding Volume Hierarchy Traversal Architecture

The second part of the ray scene intersection is the BVH Traversal architecture. When a ray needs to be intersected with a scene, it is first intersected with the acceleration structure of the scene. This is done by checking if the ray intersects the parent node of the BVH. If it does intersect, then the children of the parent node are checked for intersection with ray. This happens recursively until the leaf nodes are reached. This is called BVH traversal. If the ray misses the leaf nodes, or if the nearest n leaf nodes are needed, then ray back traverses the tree up the tree up to a point

where the ray intersects multiple nodes with the same immediate parent. At that point, the unexplored node is traversed. The sequence of nodes traversed by the ray is called the traversal path.

The issue with accelerating BVH traversal lies in the fact that the length of the traversal path is variable, and so are the nodes explored. This makes traversing a batch of rays in parallel challenging unless the rays are coherent. But ray coherency can be assumed only in primary rays. So, the architecture must be designed to support incoherent rays, which also implies that the problem is not embarrassingly parallel. And like the RTI tests, BVI tests are also memory intensive and incoherent. Some architectures propose the use of several independent cores for traversing rays incoherently, while other methods try to sort rays for improved coherency and then traverse a packet of rays. But these methods have memory bandwidth issues and poor performance for secondary rays respectively.

Because there is no guarantee that the nearest intersecting object will lie in the nearest bounding volume, a ray may need to back traverse and explore alternate paths in the BVH. But if back traversal happened infrequently enough, then the architecture could be designed to accelerate traversal paths that have no back traversals. In order to answer the question of how frequently back traversals occurred, an experiment to obtain the back-traversal statistics was introduced. A complex scene was rendered and counters for back traversal were added to each ray.

The statistics resulted in 40% of the rays not needing any back traversal, which was significant enough to design an architecture that sped up back traversal free traversal paths.

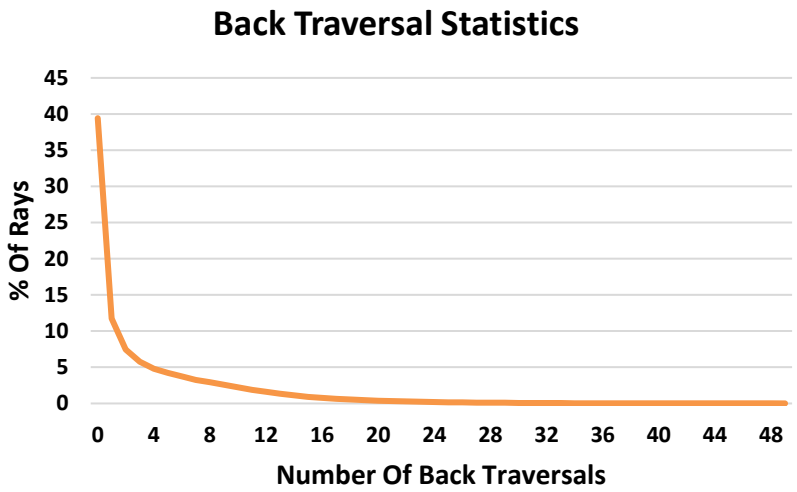

Figure 3-4. Back traversal statistics

Further, the distance in BVH levels between the back traversal was also studied. This helps determining the locality of the traversal paths. The statistics show that about 90% of the rays back traverse within 8 levels of the tree. This information helps in determining the breakdown of the tree into subtrees.

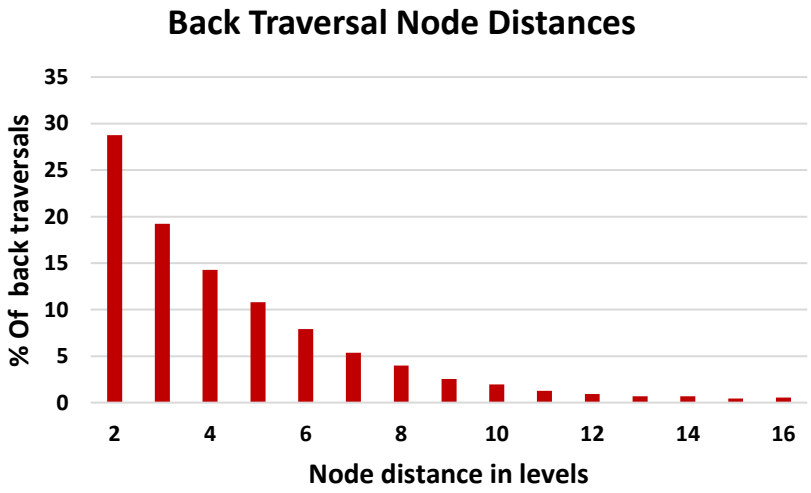

Figure 3-5. Distance between nodes in the back traversals

.

CHAPTER 4
ARCHITECTURE FOR RAY TRIANGLE INTERSECTION

RTI Pipeline

A ray is against a triangle for intersection by means of the Moller Trumbore algorithm. The operations in the Moller Trumbore algorithm are reorganized, as in the figure, to minimize latency. The reorganized datapath is encoded into a pipeline in and a pipeline containing the valid bit is also provided. When a ray and a triangle are passed into the datapath, a '1' is sent into the start of the valid pipeline. This is propagated through the pipeline and can be invalidated in stage 4 of the pipeline if the ray does not intersect the triangle.

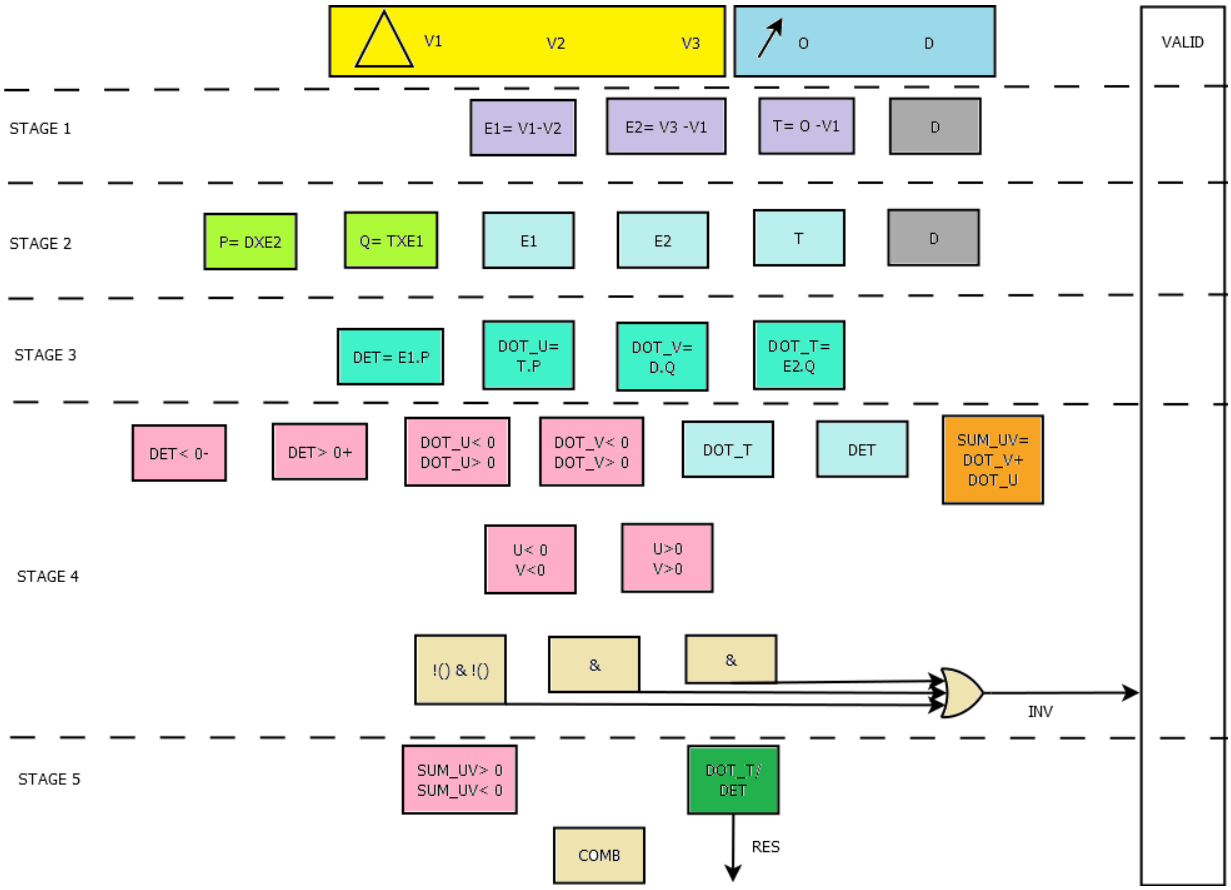

Figure 4-1. The RTI pipeline

This pipeline can perform one RTI test per cycle, and needs to be fed a new ray and triangle for each cycle. But to obtain a higher rate of RTI tests per second, some form of parallelism must be implemented.

Increasing Throughput: Multiple Pipelines

To get higher throughput many pipelines can instantiated and run in parallel. But this requires two challenges to be solved. Firstly, the pipelines must be fed with ray and triangle data every cycle despite memory bandwidth limits, and secondly, only the nearest ray triangle intersection must be returned.

Memory Bandwidth Limit

The first challenge is to minimize the memory bandwidth requirements of many pipelines in parallel. Each pipeline needs a new ray and a new triangle every cycle, and that is about 72 bytes of data per pipeline per cycle. This equates to 7.5 GBytes of data per second per pipeline (running at 100 Mhz). Most memory systems will be by just a few pipelines if the memory bandwidth is not utilized well. But the silver lining is that primitives are tested against many rays, and likewise, a ray is intersected by many rays. This idea is leveraged by sharing or broadcasting data to multiple pipelines and caching commonly accessed data.

This is done by collecting the rays that need to be tested for intersection against a leaf node, and caching them on the chip. The pipelines receive their rays form the ray buffers. Separately, the triangles in the leaf node are sequentially streamed and broadcasted to all the pipelines. This way multiple pipelines can run in parallel with a small memory bandwidth footprint.

Determining the Nearest Intersection

When a ray is tested for intersections against the primitives in a leaf node, it is quite possible that the ray might intersect with multiple primitives. In such cases, the nearest intersection is desired as it is the point in the scene that is not occluded by anything else. This is done by using a minimum capture circuit which receives a stream of intersections associated with a ray and stores the nearest intersection.

Hardware Architecture

The hardware architecture consists of the ray packet buffer, which caches a packet of rays that shall be tested against a stream of triangles, a triangle buffer, which holds the triangle being tested and broadcasted, the RTI pipelines, minimum capture units at the end of each pipeline and the result capture buffer, that captures and reads out a packet of nearest intersections corresponding to the ray packet.

The rays are stored in the DRAM, and streamed into the RTI accelerator by a DMA. The DMA is connected to the ray packet buffer, which is a wide shift register. The DMA shifts up to n rays at a time, where n is the number of pipelines, into the ray packet buffer. The ray packet buffer is segmented, with each segment containing a ray. Each segment feeds into a unique RTI pipeline.

The triangles are stored in the DRAM of the FPGA board and are streamed in to be broadcasted to all the pipelines. The streaming is managed by a DMA controller. On the NOVO G, the DMA controller takes 3 cycles to load each triangle from memory. A triangle buffer is used to assemble the chunks of the triangle before it can be broadcast. The triangle buffer is designed to be a double buffer with one buffer assembling the upcoming triangle and the other buffer broadcasting a complete triangle to the pipelines.

The minimum capture unit is a simple compare capture unit that compares the nearest intersection obtained and updates the minimum if a nearer intersection is found. One minimum capture unit is provided for each RTI pipeline,

Finally, the result capture buffer is designed to be the exact opposite of the ray packet register. It is also a shift buffer, with each segment capturing the output from a single min capture unit. Once the minimums are captured, this buffer feeds into a DMA which writes into the result DRAM.

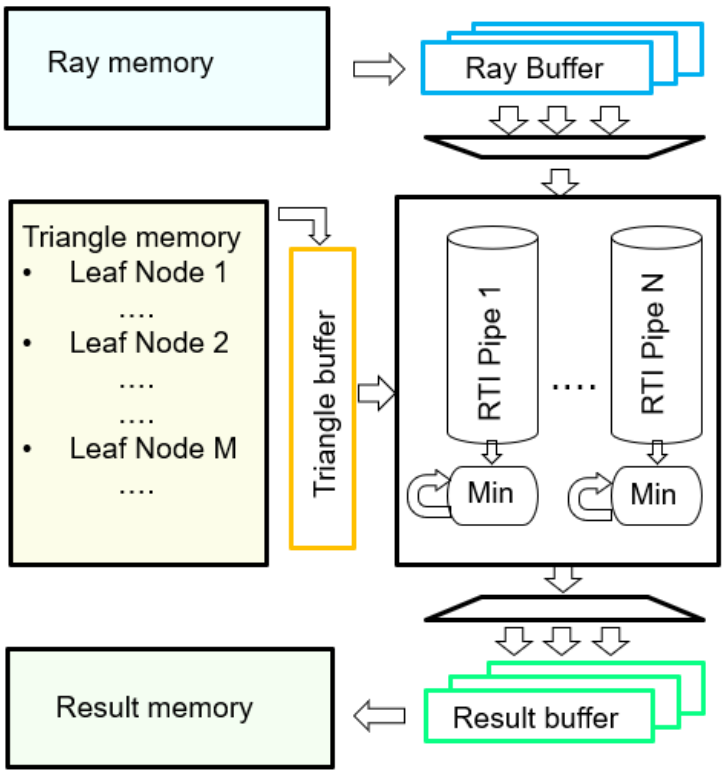

Figure 4-2. RTI acceleration architecture

Workings of the Hardware

The hardware is used by loading the list rays intersecting each leaf node (LNIS) into the ray DRAM. The triangles are stored in to the DRAM in a flat array such that all the triangles contained in a leaf node appear in contiguous memory locations in the

DRAM. This simplifies streaming the triangles into the hardware. Then the hardware is given the start address of the list of triangles contained in a leaf node along with the triangle count. Similarly, the start address of the list of rays that intersect this leaf node and the ray count are provided to the hardware. Next, the go signal is asserted to enable the hardware. The hardware streams in up to n rays to fill in the ray buffer, where n is the number of RTI pipelines. Then the ray buffer is held constant while all the triangles that belong to the leaf node are sequentially streamed into the triangle buffer. The triangle buffer takes 3 cycles to assemble a triangle and one cycle to broadcast it. Because of this, the RTI pipelines get a new triangle once every 4 cycles. A data valid bit is set every time the pipelines receive new data. The RTI pipeline determines if an intersection exists and passes this intersection to the min capture unit which selectively stores the nearest intersection. Finally, the processing of n rays is completed when the result capture buffer captures the intersections and stores it into the DRAM. If the number of rays intersecting the leaf node exceed the number of pipelines, then the above steps are repeated until all the rays are processed, n rays at a time.

Ray Packet Interleaving to Boost Performance

The triangle buffer takes 3 cycles to assemble a triangle and one cycle to broadcast it. Because of this, the RTI pipelines get a unique ray triangle pair only once every 4 cycles. But the pipelines are capable of processing one ray every cycle, and can be better utilized. This is done by temporally interleaving 4 ray packets stored in 4 ray buffers such that at every n th cycle, buffer n%4 (modulus) feeds the pipelines. Likewise, 4 collection buffers store the results. Thus, 4p rays can be processed parallelly, with p being the pipeline count.

CHAPTER 5
ARCHITECTURE FOR BVH TRAVERSAL

Complex scenes with millions of triangles are organized into BVHs which are usually 25 to 32 levels deep and are in the multi gigabyte sizes. Further, rays can take any path along the BVH to find its nearest intersection point, leading to random memory accesses. To improve the cache locality, the BVH is divided into multiple levels, namely the top level BVH and the bottom level BVH.

The top level BVH is a BVH of the objects in the scene, with each object being a leaf node. For instance, a tree or a bird can be considered an object. The bottom level BVH is the BVH of the triangles in an object. To find the nearest intersection, a ray traverses the top-level tree and then the bottom-level tree. To prevent loading the top level BVH multiple times, the nearest N intersections between the ray and the N nearest objects it sees are stored. The traversal of the bottom level trees can be parallelly done to improve parallelism. The hardware described below manages traversal of the top level BVH traversal and returns the nearest N intersections seen by a ray.

BVI Pipeline

The BVI pipeline consists of checking of the ray intersects with the bounding planes of the bounding volume and then testing if these intersections are on the surface of the bounding volume. The intersection pipeline consists of 6 substracts and 6 multiplies. The pipeline requires a new ray and a bounding volume each cycle and results in a new intersection test every cycle.

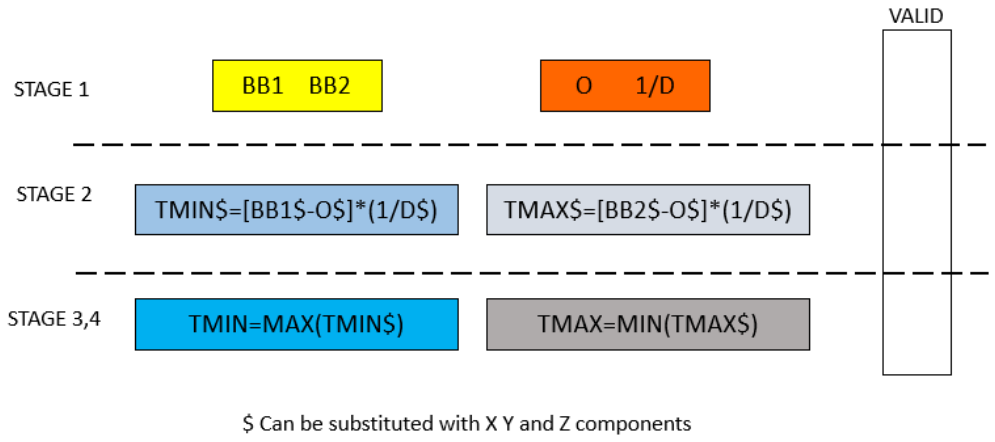

Figure 5-1. BVH pipeline data path

Increasing Throughput: Deep Parallelism

When a ray traverses a tree, it can take any path to reach the nearest leaf node in its path. When a ray intersects a bounding volume, the bounding volumes contained in it, also known as its children, are loaded and tested against the ray. While it is ideal to have the children in a memory location that is contiguous with the parent bounding volume to leverage burst performance of the ram, it is not the case in a BVH. The BVH is stored as a list of nodes, which have pointers to their children and parents. This leads to a lot of random memory accesses while the ray traverses. Further, if the ray does not find an intersection with a triangle in the nearest node then the ray needs to back traverse along the BVH tree to a different leaf node. To back traverse, the ray needs a stack that stores a list of possible alternate routes the ray can take while it traverses the BVH. Compounding this, a ray stack is to be maintained for each ray being processed. This necessitates a stack management unit which would need additional memory bandwidth and could have random accesses.

The next set of issues arise with traversing multiple rays in parallel in a SIMD style, also known as wide parallelism. But each ray can traverse a different path from

another, and can even have different number of back traversals from a ray travelling in the same direction. Therefore, SIMD approaches fail or have low pipeline utilization because the rays don't have the same execution flow. Further naïve caching methods used by the L3 or L2 cache simply cannot detect the memory access patterns of these rays and won't help in speeding up traversals of large BVHs.

The other form parallelism, namely deep parallelism, is a viable option to the case of BVH traversal. The deep parallelism based architecture is based on arranging the traversal pipelines back to back, with each pipeline managing the traversal of a single level. Further, the pipeline is attached to small BRAMS which cache the bounding volumes present at a specific level. This means that the available memory bandwidth to the traversal pipelines is the collective bandwidth of the BRAMS associated with each pipeline, which exceeds DRAM bandwidth many times over. Adding to this, the back to back connections allows a ray that is read in to the first level pipeline to go through all the pipelines.

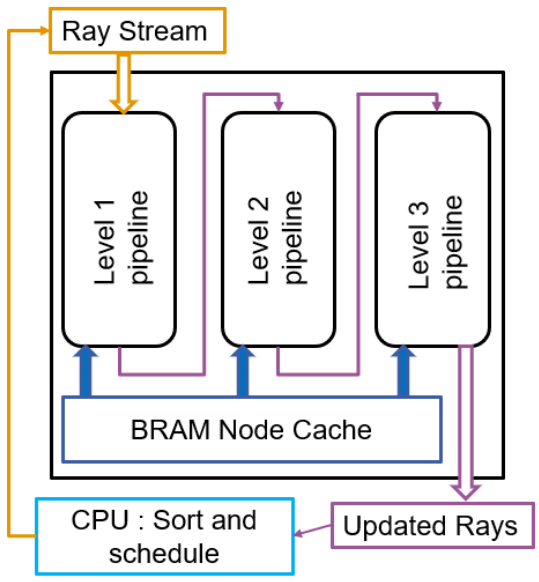

Figure 5-2. BVH traversal architecture

The back traversals are performed by recirculating a ray that exits the last BVI pipeline into the first BVI pipeline. A reduced ray traversal stack maintains a notion of the traversal path taken by the ray and informs the pipelines to chose alternate paths if necessary. This enables a ray to find its next nearest intersections.

Reduced Ray Traversal Stacks

The final issue to be addressed is to manage the ray traversal stack. The ray traversal stack stores pointers to the bounding volumes which are viable alternate traversal paths for the ray. This can mean that a 32-level deep BVH can have up to 31 stack entries, with each entry being 32 bits wide. But a novel scheme to maintain the ray traversal stack is developed that reduces this storage to just 2 bits per level. This reduction in the ray traversal stack size can be leveraged by packing the reduced ray traversal stack along with a ray and passing it through the pipeline. This way the need for a stack management unit is completely removed. The reduced ray traversal stack is named as the ray history.

Hardware Architecture

The traversal core utilizes deep parallelism by implementing back to back traversal pipelines, with each traversal pipeline managing a level of the subtree. Because each level of the BVH has 2 children in the case of a BVH 2, 2 parallel BVH pipelines are used to simultaneously test both children at a level in the BVH and the nearest child is identified. This information is used in determining the node that must be traversed next. The traversal pipeline, which manages traversing a single level of the BVH, is known as the subtree level pipeline. The subtree level pipeline consists of 2 BVI pipelines, a comparator that determines the nearest valid intersection, the history update logic, the leaf node latch logic and supporting BRAMs.

Two BVI pipelines

The subtree level pipeline has two BVI pipelines that receive the bounding volume data from two separate BRAMs. The left side children in a level of the BVH are stored into one BRAM and the right-side children are stored into the other BRAM. This way the memory bandwidth is effectively doubled while avoiding data replication. The ray being processed is broadcast to both pipelines along with the address of the bounding volumes that the ray is to be tested with. This address is access the specific bounding volume from the BRAM. The MSB of the address is reserved to indicate if the bounding volume being accessed is a child. This bit triggers the latching logic to store the leaf node if it is the next node to be traversed.

Compare Unit

The compare unit receives the intersection data from each pipeline and determines the number of bounding volumes intersected by the ray, the nearest bounding volume and the furthest bounding volume. This is used by the ray history update unit to determine the next bounding volume to be traversed.

History Update Unit

The ray history consists of two bits to store the state of the ray with respect to the current level it is travelling. This along with information about the number of interactions and the nearest intersection and a check to see if the current level is the lowest explored level with multiple intersections, completely defines the state of the ray and which node it must traverse next.

The states in the history are:

1. 00: State indicating that the ray has not traversed the current level before.
2. 01: State indicating that the ray needs to be ignored / passed through

3. 10: State indicating that the ray has seen multiple intersections at this level for the first time.

4. 11: State indicating that the ray has traversed all possible paths in the current level

The choice if traversing the nearest or next nearest child is made as follows:

- The nearest child is explored if the ray traverses a level for the first time and the ray sees one or more intersections
- The next nearest child is explored if the ray traverses a level for the second time, with the current level being the lowest explored level with multiple intersections.

The history update logic encodes these conditions and updates the history accordingly.

Leaf Node Latch Unit

If the bounding volume that needs to be traversed next is a leaf node, then it can be considered as nearest intersection and is attached to the ray. This unit makes sure that the nearest intersections are latched.

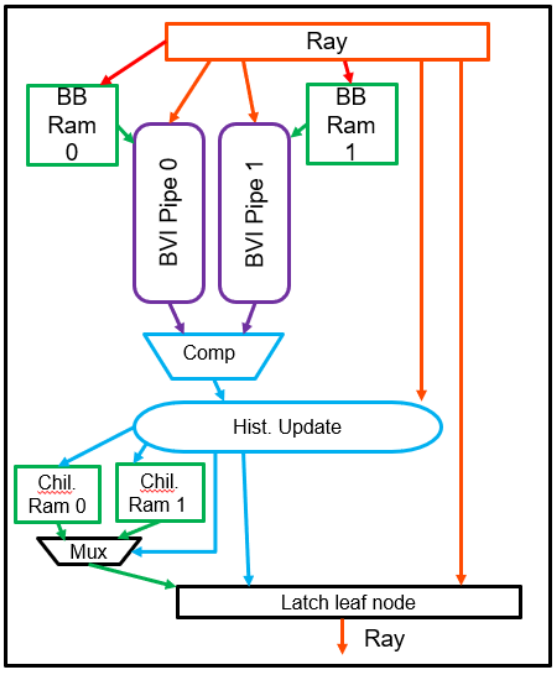

Figure 5-3. Single subtree level pipeline

Workings of the Hardware

The hardware is used by loading the top level BVH into the BRAM by individually enabling the appropriate BRAM controller of a subtree level pipeline and sequentially writing the bounding volume data. Then the rays are formatted to be acceptable by the FPGA and written into a contiguous memory location on the DRAM. Then the FPGA is provided with the addresses of the start location of the rays and a separate location to store the results.

The architecture also has 2 DMA controllers, one to read in ray data from the DRAM and another to write the processed ray into DRAM. The read DMA streams in rays into the traversal core, which in turn, processes the rays and feeds the write DMA, which stores the processed rays onto DRAM. These DMA controllers receive the addresses of the start of the rays and the start of the result storage area.

When the go signal is asserted by the CPU, the DMAs are turned on and the pipeline is enabled. The pipeline stays enabled until all the rays are processed and written into the result storage area. A ray is considered processed when either:

- The ray has fully explored the BVH and has not found N intersections
- The ray has found N intersections.

Performance Predictions

The hardware is designed to process about 50 M rays/ second given that each ray needs to perform about 4 back traversals on average, and the traversal core is running at 200 MHz. This excludes the time taken to load the BVH and the time taken to copy the rays from and into DRAM. This number can be improved by

- Doubling the number of pipelines (wide parallelism) and leveraging the BRAM in dual port mode (2X boost)

- Enforcing balanced trees so that individual node IDs and child node indicators can be avoided, which saves BRAM.
- Using 16 bit floats for the Bounding volume to save on BRAM
- These optimizations can in total lead to 3-4X performance gains.

CHAPTER 6
CONCLUSION

This thesis explores the workings of ray tracing, delving deeply into the methods for finding the ray scene intersection. Core areas for acceleration are identified as the RTI tests and the BVH traversals. The challenges in parallelizing these parts are discussed and can be summarized as memory bandwidth and hardware overhead for RTI tests and incoherent ray traversals for BVH Traversals.

The RTI hardware is SIMD machine that processes a ray packet at a time and the BVH hardware is a superdeep pipeline that processes a stream of rays. The RTI hardware is found to be capable of 1.35 G RTI/s. It was also found that the ideal number of RTI pipelines was proportional to the number of objects in a leaf node, and larger leaf nodes improved available parallelism in the RTI hardware.

LIST OF REFERENCES

Arthur, A. (1968) "Ray-tracing and other Rendering Approaches"(PDF), lecture notes, MSc Computer Animation and Visual Effects, Jon Macey, University of Bournemouth

Fujimoto, A., Tanaka, T., & Iwata, K. (1986). Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, *6*(4), 16-26.

Spjut, J., Kensler, A., Kopta, D., & Brunvand, E. (2009). TRaX: A multicore hardware architecture for real-time ray tracing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28(12), 1802-1815.

Whitted, T. (2005, July). *An improved illumination model for shaded display. In ACM Siggraph 2005 Courses* (p. 4). ACM.

BIOGRAPHICAL SKETCH

Ajay Simha Modugala received his master's degree in electrical and computer engineering from the University of Florida in the fall of 2017. He has completed his undergraduate studies from Indian Institute of Technology Roorkee, India in July 2013. This thesis is a result of research work while working as an assistant for Dr. Greg Stitt at the University of Florida.