Accelerating Ray-Tracing on FPGA
Semester project report

Andrew Dobis

Supervisor - Mikhail Asiatici
Professor - Paolo Ienne

January 2020

Contents

1 Introduction

Real-time Ray-Tracing has been the holy grail of computer graphics for many years, up until Nvidia released their *RTX* line of GPUs, using their new *Turing* Architecture, containing what they call *RT-Cores.* This new addition to the Nvidia GPUs enabled them to do what is know as *Hybrid Rendering*, which is a mixture of two rendering techniques *Rasterization* (direct conversion of mesh primitives into a bitmap that can then be displayed onto a screen) and *Ray-Tracing* (more about that later). This method allowed them to render a couple of rays per pixel per frame, when needed, in a Real-Time application, but nothing more. This is due to the high irregularity in the memory access patterns of a standard Ray-Tracer, which causes a very large memory bottleneck due to the way that GPUs utilize their memory[1].

This brings us to our main problem, is it possible to design an accelerator for Ray-Tracing on an FPGA and still achieve any significant performance enhancement, knowing that the application in question has an irregular memory access pattern?

1.1 Basic Computer Graphics

Before we try to tackle our problem, we first need to have a few basic notions about Computer Graphics. A *Scene* is comprised of a *Camera*, one or more *Light-sources* and *Geometry.* The Camera will considered as the source of all of our *primary rays* and the different Light-sources will be the rays' sinks. Geometry is made up meshes which represent a single continuous object (e.g. a sphere can be comprised of a single mesh). A mesh itself is comprised of multiple primitives (usually triangles), which can be considered as the basis of all of the geometry in a scene. These primitives have to organized in someway in memory and there are multiple ways to do so depending on the requirements. The structure we will be using is called a *Bounding Volumes Hierarchy* (BVH), which is a tree-like structure where the leaves contain one or more primitives (grouped by their spatial proximity) and the other nodes contain spatial *Bounding Boxes* in a hierarchical fashion such that the root of the tree contains the entire scene.

Now that we know a little bit more about Computer Graphics, let's take a look at how the Ray-Tracing algorithm works.

[1]For a more in depth look at why Hybrid rendering rather than true Ray-Tracing is used, I highly suggest reading this article by Nate Oh [1].

1.2 Ray-tracing Algorithm

The idea behind Ray-Tracing is to, rather than projecting mesh primitives onto the screen orthogonally, like with rasterization, simulate the light rays in the scene individually in the following manner, first project a ray (aka. a *primary* ray) from the Camera of a given scene. The ray then continues across (i.e. *traverses*) the scene until either it intersects with Geometry, in which case it shoots a *secondary* ray and repeats the same steps as previously stated, or it continues until it fades out (i.e. until the scene has been fully traversed or, in the presence of atmospheric scattering[2] the ray has travelled its maximum distance), in which case we return to the Camera and shoot a new primary ray and repeat. It is also possible that in the case of an intersection, we shoot a special type of ray that is called a *shadow ray* which doesn't generate any new secondary ray in case of a future intersection.

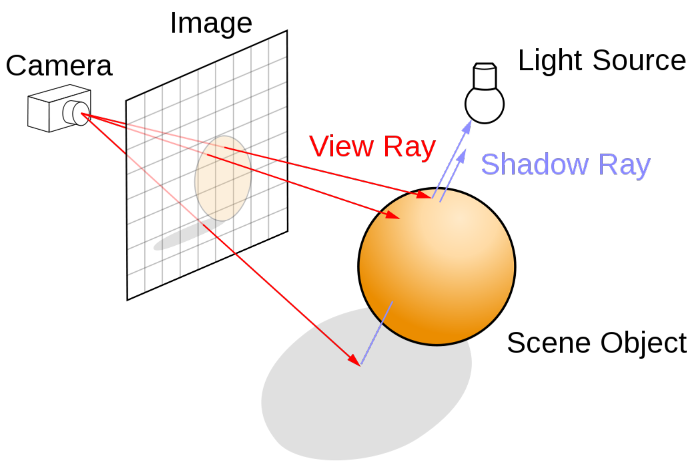


Figure 1: Diagram representation of simple Ray-Tracing in a basic scene containing a light source and a sphere.

For our solution, we will be using an open-source educational Ray-Tracer named *Nori*[3]. The idea behind using Nori to build our accelerator is that it is a rather bare-bones implementation of a Ray-Tracer and thus an accelerator for Nori should, in theory, work for any other Ray-Tracer. The part we will be accelerating will be the BVH traversal by a given ray, and detecting the intersection between a ray and a primitive, since, after a study done

[2]More about atmospheric scattering and anything relating to Ray-Tracing can be read in the amazing book on the subject by Wenzel Jakob et al. [2, §1.2.7].

[3]More details about Nori (the Ray-Tracer, not seaweed) can be found on Wenzel Jakob's Github page [3].

prior to this project, those two pieces comprise 80% of the Ray-Tracer's total computation time.

1.3 BVH Traversal and Primitive Intersection

Let's begin by understanding how a ray traverses the BVH of a given scene. Given a ray, we start by looking at the two bounding boxes that are contained in the root of the tree (i.e. the bounding boxes of the left and right children of the root), if one of the two boxes is on the ray's path, which we can know by computing the intersection between the ray itself (defined by an origin and direction) and the bounding box of the node (defined by a minimum and a maximum), then we move on to said node and compute the intersection between its two children's boxes. We then continue as such until we reach a leaf and, if we intersect with the leaf's bounding box, then we move on to computing the potential intersections between the ray and the primitives stored in the leaf.

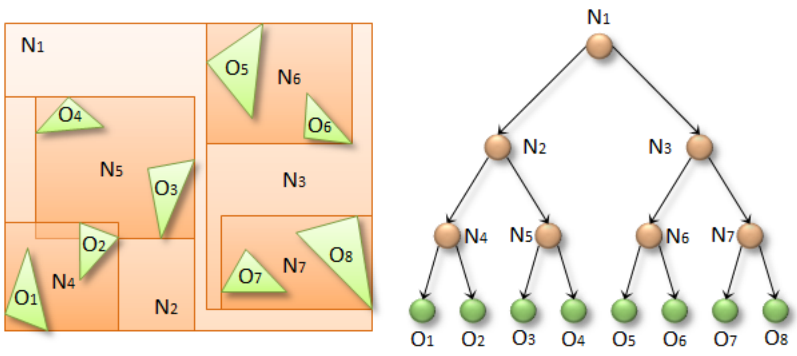


Figure 2: Example of a BVH tree (on the right) and the view from inside the root of the tree (on the left). *Figure found on Nvidia's devblog* [4].

Primitive intersection is a similar process as computing the intersection with the different bounding boxes in the BVH. All we need to do is, once we've reached a leaf that is on our ray's path, to verify if one of the primitive, stored inside the leaf, is on our ray's path. This can be done by computing the intersection between our primitive (defined by three points) and our ray. If an intersection has been found, we return the data about the mesh our ray has intersected with and about the exact location of the intersection on said mesh. If no intersection has been found then we return up the BVH and continue our traversal in a Depth First Search fashion.

2 Hardware implementation

Now that we know how the Ray-Tracing algorithm works, we can go into detail about how our Ray-Tracing hardware solution was implemented. First, we need to have a general view of how the accelerator will be interfacing with the software renderer. In our case we are using Xilinx's SDSoC to run our accelerator along side with the Ray-Tracing software on a Xilinx FPGA. The interface between hardware and software will be as follows: the software renderer will feed our accelerator a ray, the address to the root of the BVH representing the scene we are rendering and the address of the set of primitives used in said scene. Then our accelerator will output whether it has found an intersection for the given ray, and data about said intersection. We will now take a look at how our accelerator was implemented.

2.1 The Environment

Before we dive in to the precise implementation, it is important to understand the environment we will be working it. The goal, for now[4], is to run the accelerator on a Xilinx ZC706 that pairs an FPGA with a dual-core Cortex A9 ARM processor. This allows us to run our accelerated Ray-Tracer entirely on the board, with the accelerator running on the FPGA and the software renderer (minus the accelerated RayIntersect function) to run on the ARM cores.

2.2 The Top-level design

Our accelerator is separated into three distinct modules, namely FetchBVH, Traversal and PrimitiveIntersection, each of which will be explained in detail in the coming subsections. Here we will mostly discuss the Top-Level architecture of our accelerator (i.e. how the three modules are interconnected to form a single accelerator). The three modules are connected in quite a straight-forward fashion. First the accelerator's input ray is fed directly to the Fetch module. Then the node, and Ray-NodeIdx pair, are grouped into a triplet that is fed directly to the traversal whose output is itself given directly to the final module. Finally the final module's either output from the accelerator or sent back to the traversal module to continue the DFS.

The complexity of the Top-Level is in the interface with Mikhail's Memory System, since we have to deal with three different modules each trying to

[4]The idea would be to, later on, adapt the solution to have it run in a much more flexible and realistic environment, such as Amazon F1 instances which use much larger FPGAs paired with Xeon processors.

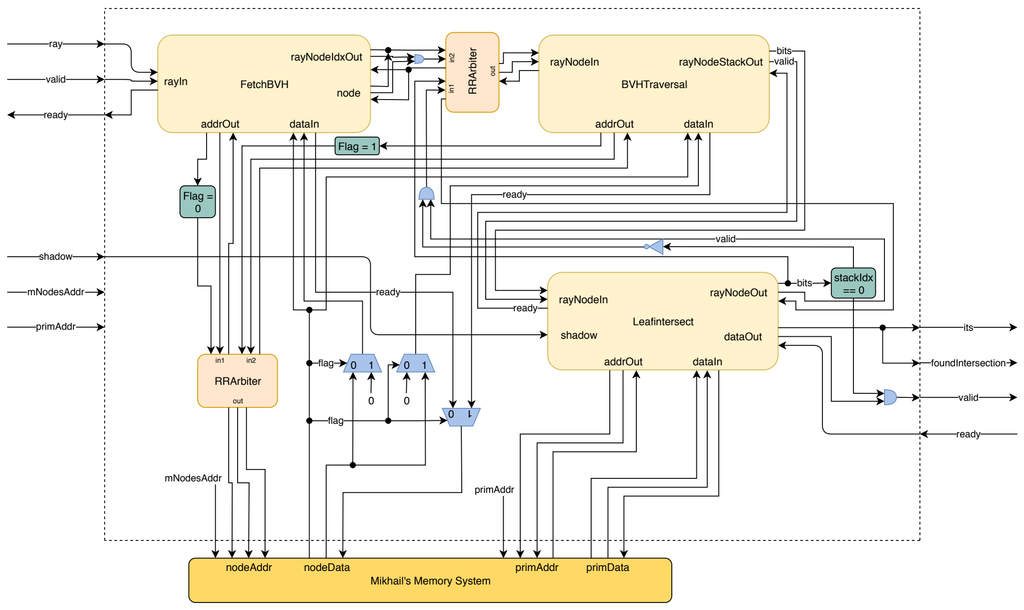


Figure 3: A detailed view of the Top-level implementation of our accelerator, containing 3 modules: FetchBVH, TraverseBVH and LeafIntersection.

access it. A Memory System interface has two decoupled I/Os (i.e. ready-valid interfaces that require a handshake to function), one for the address of the requested data and an other for receiving said data. In our case we have two different Memory System interfaces, one that handles node requests and an other for primitive (triangle) requests. The node interface is shared by two modules: FetchBVH and Traverse. For deciding which address to output, we associate a flag to each request representing which module the request came from. Then the two address requests are connected to a round-robin Arbiter, that will decide on which one to send to the memory system. Finally the data output by the memory system will be sent off to the corresponding module depending on the flag.

Now let's take an in-depth look at how the FetchBVH module was implemented.

2.3 Fetch-BVH

First let's start by understanding what the Fetch module is for. The goal of this module is to initialize the traversal of a BVH by fetching the root of the scene that is being traversed by a given ray. Figure 4 shows a detailed representation of the hardware implementation of the Fetch module. Let's go through the functioning of the implementation.

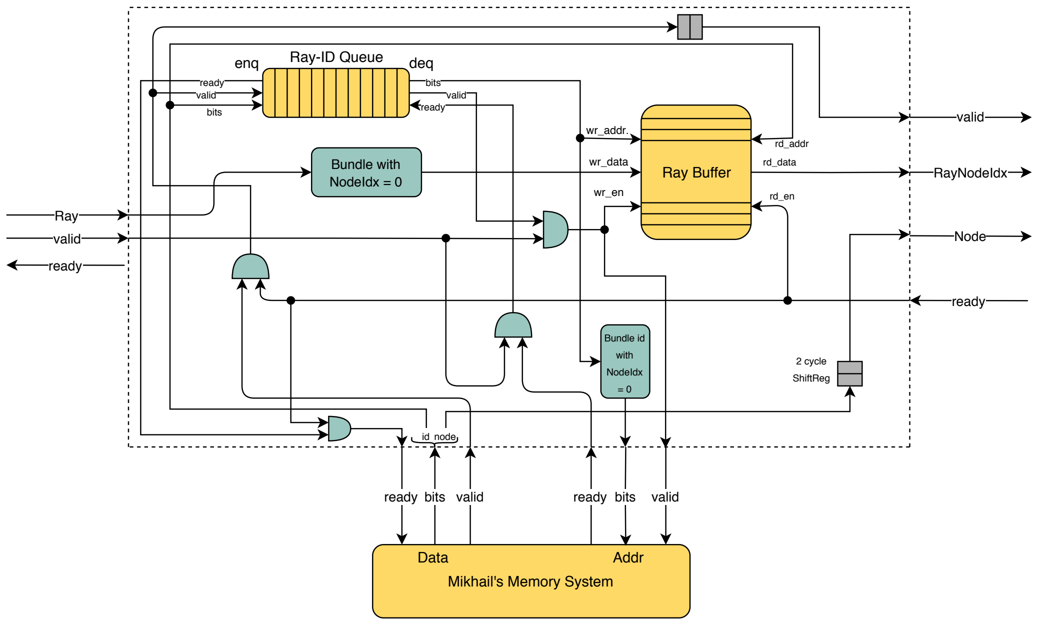


Figure 4: A detailed diagram representing the hardware implementation of the Fetch module.

First off, the given ray is associated to an ID obtained from the Ray-ID Queue and a node index (NodeIdx) which is initialized at 0 (i.e. at the root of the BVH). The ray and the node index are then stored inside the Ray-Buffer at the address defined by the ray's given ID, note that the Ray-ID Queue contains as many IDs is free space in the Ray-Buffer. Whence the ray is stored inside the buffer, the id and node index are sent off, via an interface with the top-level, to Mikhail's Memory System so that it can fetch the node from memory. At this point, while we are waiting for the result of the memory read, we can start working on a new ray. We can thus have as many in-flight rays as there are IDs in the Ray-ID which is how we enabled high pipeline-level parallelism. Whence we have received a result, in the form of a node associated to an ID, from the Memory System, we use the ID as a read address to retrieve the Ray-NodeIdx pair from the Ray Buffer. Then we en-queue the ID back into the Ray-ID Queue (since we're done using it). Finally we output the retrieved Ray-NodeIdx pair along with the fetched root of the BVH delayed by two cycles. The delay must in order to synchronize the node with the retrieved Ray-NodeIdx pair, since the buffer we are using has a two cycle read latency. Our final output is considered valid two cycles after the Memory system has output valid data.

A comment should added about the above design, since it is, at its core, quite flawed. The module will always fetch the root of the same scene and cause contention with the traversal module in the top level node fetching interface. A better way to do this would have been to fetch the root of the scene at initialization (before a single output is considered ready) and then storing it in a single line cache that is used for subsequent root fetching. This solution would avoid any contention with the traversal module and would probably lead to a better performing module. This wasn't implemented in the current solution due to time constraints and should be done in future work on the project.

2.4 BVH Traversal

Now let's move on to our second module which takes care of traversing the BVH using a given ray and the root of the scene. This module is far more complex than the previous one and must thus be separated into multiple sub-modules, which we will describe in detail in the following subsections.

2.4.1 Top-level Traversal

Our module is separated into five distinct sub-modules, namely InitTraversal, BboxIntersect, RayBuffer, TraversalControl and PrepFetch. The top-level of the module isn't as straight-forward as the top-level of the accelerator, since here the same ray can loop multiple times through the module (until it reaches a leaf, at which point it is output to the next module). Figure 5 shows a detailed view of the top-level implementation of our module.

Let's first look at the outputs of the InitTraversal sub-module, namely a Ray-RayID-Node-NodeIdx-StackIdx group and a valid signal. Said group is sent off to the RayBuffer sub-module where it will be sharing the writing interface to the buffer with the in-flight rays wanting to continue their traversal. The RayBuffer sub-module will then have an interface that will allow us to connect the selected Ray-RayID-Node-NodeIdx-StackIdx group to the BboxIntersect sub-module, so that said ray can proceed with the traversal. Since said sub-module has a latency of 14 cycles, we have to delay the rayId and it's corresponding valid signal by 12 cycles using a shift Register. When the rayId and valid signal leave the shift Register, we will have 2 cycles left before their corresponding result is output by the BboxIntersect sub-module during which we will perform a preemptive read of the associated data, that is stored inside the RayBuffer sub-module, so that the result of the BboxIntersect sub-module will be synchronized with the data read from the Ray-Buffer sub-module.

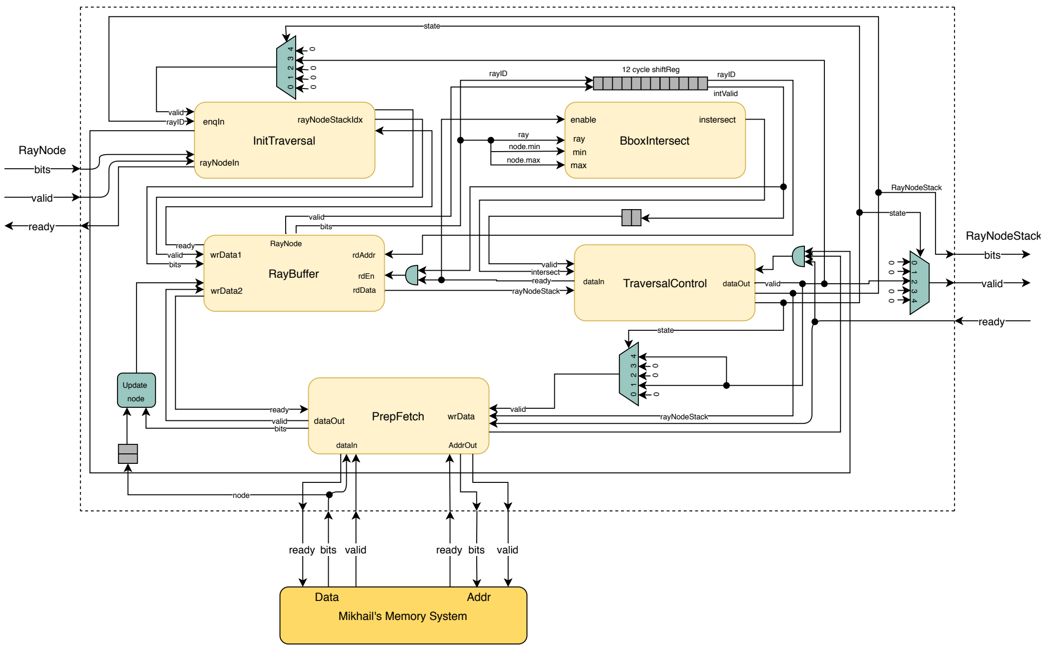


Figure 5: A detailed diagram representing the hardware implementation of the entire Traversal module, which has been separated into multiple sub-modules.

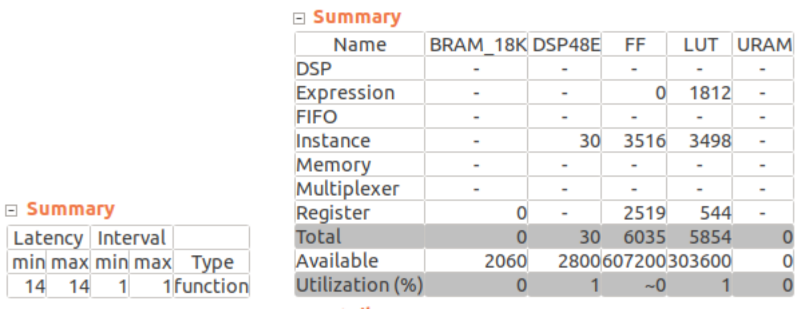
Let's now take a look at a second challenging element of our design, i.e. the interpretation of the output state of the traversal control module. Outputs from the control module are associated with a state, which represents the state at which the ray is in its traversal of the BVH. Then, depending on the state of the output ray, it will be sent to different sub-modules or even output from the traversal module. That is the cause of all of the valid signal multiplexing that can be seen in the above diagram. Let's move on and take a detailed look at each individual sub-module.

2.4.2 Ray-id initialization

The goal of the InitTraversal sub-module is to associate a Ray-Id to each passing ray. This sub-module thus contains a Ray-ID Queue and functions similarly to the Ray-Id Queue found in the FetchBVH module. ray-Ids can be en-queued back into the Ray-ID Queue via an en-queuing interface found on the sub-module. Ray-Ids are sent back to the Queue when the ray output by the control sub-module is in the DONE state.

2.4.3 Bounding box intersection detection

The goal of the BboxIntersect sub-module is, given a ray and a node's bounding box, to detect whether or not the ray has intersected with the bounding box. Due to the high number of floating point operations found within the module, I thought that it would be best implemented using High Level Synthesis (HLS). The software function for bounding box intersections had to be adapted in such a way that it was entirely independent from any other piece of code, I thus had to redefine certain types from the Ray-Tracer and needed to add pragmas enabling loop unrolling, array flattening and pipe-lining in order to achieve an optimal result[5] (i.e. a result with the lowest latency and an initiation interval of 1). Said results can be found in figure 6.



Summary

Latency		Interval		
min	max	min	max	Type
14	14	1	1	function

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1812	-
FIFO	-	-	-	-	-
Instance	-	30	3516	3498	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	0	-	2519	544	-
Total	0	30	6035	5854	0
Available	2060	2800	607200	303600	0
Utilization (%)	0	1	~0	1	0

Figure 6: Summary of the BboxIntersect sub-module's latency (amount of cycles it takes to go from the input to the output) and initiation interval (on the left), and its resource usage (on the right).

2.4.4 Ray-Node Buffering

The goal of the RayBuffer sub-module is to handle the reading and writing into the buffer which it contains. The reading part functions the same way as in the FetchBVH module, with the read-address and read-enable coming from outside of the sub-module. The writing however differs because we now need to handle two potential writers, rather than simply just one. This is done by using a Round-Robin Arbiter which, as the name entails, schedules the different writers in a Round-Robin manner.

2.4.5 Traversal Control

The goal of the TraversalControl sub-module is to use the result, from the BboxIntersect sub-module, to decide where to send the ray next. This idea is

[5]The entire code used for the HLS can be found in appendix 1 (§5.1).

implemented using an implicite 4 state FSM for each ray, where the possible states are TRAVERSE, LEAF, DONE, and GOBACK. One of the four states is selected depending on the output of the BboxIntersect sub-module and the node currently associated to our ray. A diagram of our state machine, with an added Initial state in the center (for clarity), is shown in figure 7. The output of the control sub-module will be the input group, with potentially an updated nodeIdx and StackIdx, and the output state of our ray. The implicit FSM is implemented using multiple state transition signals (similar to the ones in figure 7). Since one of the states implies a read from the stack memory, all of the results must be delayed by 2 cycles in order to mainain coherence.

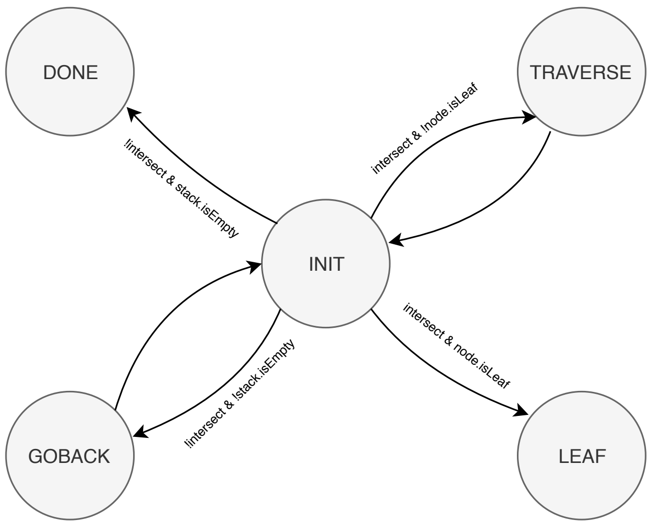


Figure 7: State diagram for the ray in the control sub-module. Here DONE and LEAF are both terminal states, i.e. a ray that ends up in one of those two states will not reenter the control sub-module.

2.4.6 Memory System Interfacing

The goal of the final sub-module, i.e. the PrepFetch sub-module, is to interface with Mikhail's Memory System. To do that, the sub-module requires to have a buffer so that it can store rays while they await the requested nodes. The sub-module also takes care of synchronizing the data so that the output from the module is coherent.

We will now move on to the final part of our accelerator, the Leaf intersection module.

2.5 Leaf Intersection

This final module was slightly less challenging to design than the previous one. The goal of the leafIntersect module is to, given the output of the traversal module (i.e. a Ray-Node-NodeIdx-StackIdx group), check for a potential intersection between the given ray and one of the primitives found in its associated node.

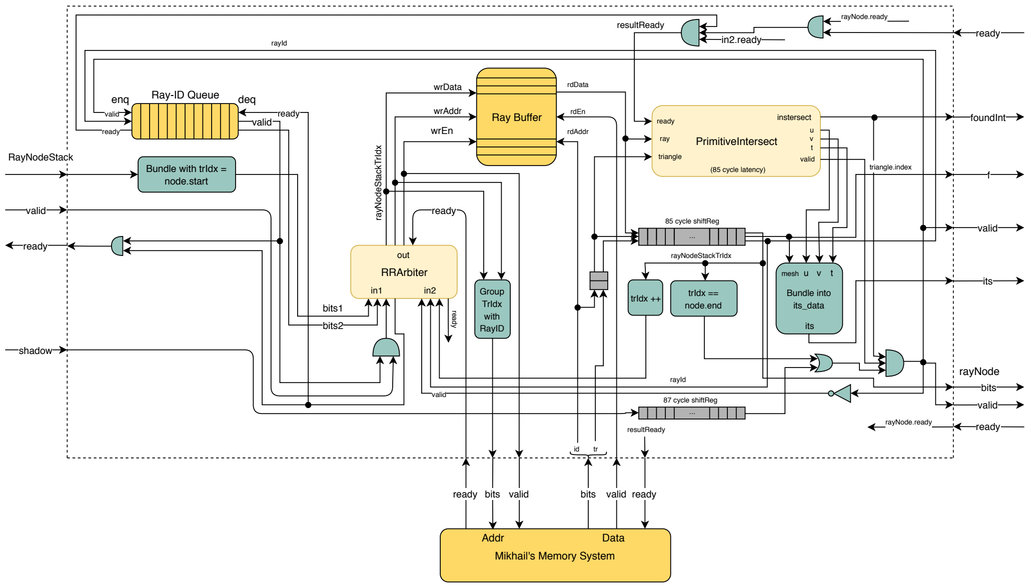
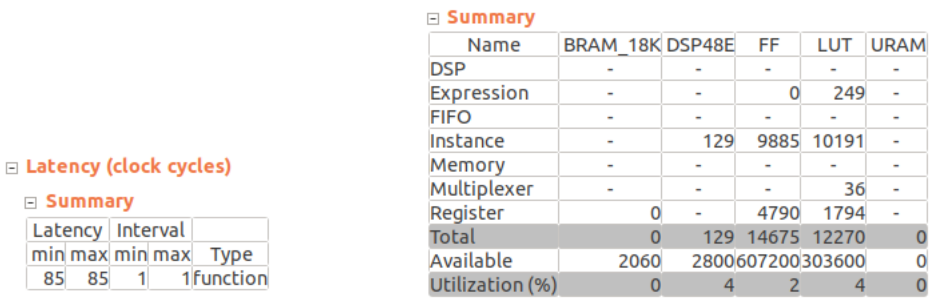


Figure 8: Detailed diagram representing the hardware implementation of the entire LeafIntersect module.

The implementation of this module was done in a similar way as the traversal module, but with less sub-modules. We start by associating a given Ray-Node-NodeIdx-StackIdx group to a new rayId (which is done the same way as in the two previous module using a RayId Queue) and a loop index (named trIdx meaning triangle index) with an initial value of node.start (the index within a global triangle memory at which the first primitive in the node is stored). Whence the new in-flight bundle is created, we want to fetch the requested primitive (located in memory at `PRIM_MEM_ADDR + trIdx`), and to do that we need to store the in-flight bundle in a Ray-Buffer (as was done in the two previous modules). The writing interface for the buffer is shared by both the newly incoming ray bundles and the ones that are still inside the intersection loop (trIdx < node.end) using a Round-Robin Arbiter.

Whence a response is received by the Memory System (under the form of an ID-Triangle bundle) the ID is sent to the ray-Buffer to retrieve the data associated to the memory read, after that the associated ray is sent off to the



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
85	85	1	1	function

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	249	-
FIFO	-	-	-	-	-
Instance	-	129	9885	10191	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	0	-	4790	1794	-
Total	0	129	14675	12270	0
Available	2060	2800	607200	303600	0
Utilization (%)	0	4	2	4	0

Figure 9: Results from the HLS of the primIntersect sub-module. The module's latency (on the left) and its resource usage (on the right).

Prim-intersect sub-module. Since said sub-module has an 85 cycle latency, we have to delay the data associated to its input, using a shift register, in order to keep it aligned with its result. The delayed bundle is then used to check whether it should output from the module or if it should be sent off to continue the loop. This is done by comparing the value of trIdx with node.end: if they're equal, then the 2^{nd} input of the buffer's writing interface's arbiter is valid, otherwise it isn't. Finally, if an intersection has been found, then the computed intersection data is output from the module along with the in-flight bundle. The shadow input is used to stop the loop in case of a shadow ray, which simplifies the computation and allows us to have a faster result.

One last element should be taken into account, the primIntersect sub-module, which will now be explained.

2.5.1 Primitive Intersection

This sub-module was done using HLS, since it consisted mostly of floating point operations without any control branching. The goal of the sub-module is to, given a ray and a triangle, detect whether or not the two intersect each-other and, if so, compute the location of the intersection in the scene. As for the bounding box intersection sub-module (§2.3.3) the software function had to be isolated from the rest of the renderer, which meant re-writing all of the used types and functions from external libraries (such as the Eigen library in our case) and adding certain pragmas for loop unrolling and array flattening[6]. The results from the HLS can be found in figure 9.

[6]The source code used to synthesise the hardware can be found in appendix 2 (§5.2)

3 Results

It is difficult to give any meaningful results with the current state of the project, since the accelerator has yet to be run on the targeted platform. However we can give an estimation of the resource usage using the implementation and data retrieved from the HLS modules, since those are the only ones doing any sort of intense computation. In our implementation, we explicitly use 5 BRAM memories (3 in traversal, 1 in fetch and another in leafIntersect) and 3 different FreeLabelQueues. The rest of the resources are mostly used up by the HLS modules for which the data can be found in figures 6 and 9. Note that the targeted platform in those figures is different from the one we are targeting. Our board actually has around 1/4 of the resources of the board targeted in the figures.

3.1 Future work

The accelerator in its current state still needs some work to be done on it. First off the FetchBVH module needs to be optimized to avoid useless requests to memory, as was said at the end of paragraph §2.2. Secondly, the accelerator has never been run along side the software renderer on an FPGA board, due to time constraints, and was only ever run in simulation. Doing that would require to merge the adaptions done to the renderer prior to this project with the current accelerator using platforms like SDSoC to then test the accelerator directly on the FPGA. Finally, one would also need to test the accelerator with and without using Mikhail's Memory system to then be able to note the total performance boost, if any is present.

4 Conclusion

In conclusion, building an accelerator for Ray-Tracing proved to be far more challenging than expected, especially in the traversal module, due to the unpredictable behavior of the given rays. Any given ray could go from immediately intersecting with a leaf to looping through the whole accelerator for around a hundred cycles without ever intersecting with anything. The biggest bottleneck however is definitely the random memory accesses caused by this type of unpredictable application and that is where I honestly think that Mikhail's memory system can be of great use, which can now be proven with its use along side the Ray-Tracing accelerator.

5 Appendix

5.1 BboxIntersect HLS C code

```
#include <limits>
#include <inttypes.h>
#include <algorithm>

typedef struct {
 float o[3];
 float d[3];
 float dRcp[3];
 float mint;
 float maxt;
} Ray3f;

/// Check if a ray intersects a bounding box
bool rayIntersect(Ray3f ray, float min[3], float max[3]) {

#pragma HLS ARRAY_PARTITION variable=min complete
#pragma HLS interface ap_none port=min

#pragma HLS ARRAY_PARTITION variable=max complete
#pragma HLS interface ap_none port=max

#pragma HLS ARRAY_PARTITION variable=ray.o complete
#pragma HLS interface ap_none port=ray.o

#pragma HLS ARRAY_PARTITION variable=ray.d complete
#pragma HLS interface ap_none port=ray.d

#pragma HLS ARRAY_PARTITION variable=ray.dRcp complete
#pragma HLS interface ap_none port=ray.dRcp
#pragma HLS pipeline

#pragma HLS interface ap_ctrl_none port=return

    float nearT = -std::numeric_limits<float>::infinity();
    float farT = std::numeric_limits<float>::infinity();

    for (int i=0; i<3; i++) {
```

```
    #pragma HLS unroll
            float origin = ray.o[i];
            float minVal = min[i], maxVal = max[i];

            if (ray.d[i] == 0) {
                if (origin < minVal || origin > maxVal)
                    return false;
            } else {
                float t1 = (minVal - origin) * ray.dRcp[i];
                float t2 = (maxVal - origin) * ray.dRcp[i];

                if (t1 > t2)
                    std::swap(t1, t2);

                nearT = std::max(t1, nearT);
                farT = std::min(t2, farT);

                if (!(nearT <= farT))
                    return false;
            }
        }

        return ray.mint <= farT && nearT <= ray.maxt;
    }
```

5.2 PrimIntersect HLS C code

```
typedef struct {
    float p0[3];
    float p1[3];
    float p2[3];
} myTriangle;

typedef struct {
    float o[3];
    float d[3];
    float dRcp[3];
    float mint;
    float maxt;
} Ray3f;
```

```
bool rayIntersect(myTriangle triangle, Ray3f ray, float *u, float *v, float *t)

#pragma HLS ARRAY_PARTITION variable=ray.o complete
#pragma HLS interface ap_none port=ray.o

#pragma HLS ARRAY_PARTITION variable=ray.d complete
#pragma HLS interface ap_none port=ray.d

#pragma HLS ARRAY_PARTITION variable=ray.dRcp complete
#pragma HLS interface ap_none port=ray.dRcp

#pragma HLS ARRAY_PARTITION variable=trianlge.p0 complete
#pragma HLS interface ap_none port=trianlge.p0

#pragma HLS ARRAY_PARTITION variable=trianlge.p1 complete
#pragma HLS interface ap_none port=trianlge.p1

#pragma HLS ARRAY_PARTITION variable=trianlge.p2 complete
#pragma HLS interface ap_none port=trianlge.p2
#pragma HLS pipeline
#pragma HLS interface ap_ctrl_none port=return

    /* Find vectors for two edges sharing v[0] */
    float edge1[3], edge2[3];

    //Inlining subP3f
    edge1[0] = triangle.p1[0] - triangle.p0[0];
    edge1[1] = triangle.p1[1] - triangle.p0[1];
    edge1[2] = triangle.p1[2] - triangle.p0[2];

    edge2[0] = triangle.p2[0] - triangle.p0[0];
    edge2[1] = triangle.p2[1] - triangle.p0[1];
    edge2[2] = triangle.p2[2] - triangle.p0[2];

    /* Begin calculating determinant - also used to calculate U parameter */
    float pvec[3];

    //Inlining cross product
    pvec[0] = (ray.d[1] * edge2[2]) - (edge2[1] * ray.d[2]);
    pvec[1] = (ray.d[2] * edge2[0]) - (edge2[2] * ray.d[0]);
    pvec[2] = (ray.d[0] * edge2[1]) - (edge2[0] * ray.d[1]);
    /* If determinant is near zero, ray lies in plane of triangle */
    //Inline dot product
    float det = edge1[0] * pvec[0] + edge1[1] * pvec[1] + edge1[2] * pvec[2];

    if (det > -1e-8f && det < 1e-8f)
        return false;
    float inv_det = 1.0f / det;

    /* Calculate distance from v[0] to ray origin */
    float tvec[3];

    //Inlining subP3f
    tvec[0] = ray.o[0] - triangle.p0[0];
    tvec[1] = ray.o[1] - triangle.p0[1];
tvec[2] = ray.o[2] - triangle.p0[2];

    /* Calculate U parameter and test bounds */
//Inline dot product
float u_tmp = (tvec[0] * pvec[0] + tvec[1] * pvec[1] +
    tvec[2] * pvec[2]) * inv_det;
*u = u_tmp;

    if (u_tmp < 0.0 || u_tmp > 1.0)
        return false;

    /* Prepare to test V parameter */
    float qvec[3];
    //Inlining cross product
    qvec[0] = (tvec[1] * edge1[2]) - (edge1[1] * tvec[2]);
    qvec[1] = (tvec[2] * edge1[0]) - (edge1[2] * tvec[0]);
    qvec[2] = (tvec[0] * edge1[1]) - (edge1[0] * tvec[1]);

    /* Calculate V parameter and test bounds */
    float v_tmp = (ray.d[0] * qvec[0] + ray.d[1] * qvec[1] +
        ray.d[2] * qvec[2]) * inv_det;
    *v = v_tmp;

    if (v_tmp < 0.0 || u_tmp + v_tmp > 1.0)
        return false;
    /* Ray intersects triangle -> compute t */
    //Inline dot product
    float t_tmp = (edge2[0] * qvec[0] + edge2[1] * qvec[1] +
        edge2[2] * qvec[2]) * inv_det;
    *t = t_tmp;


    return t_tmp >= ray.mint && t_tmp <= ray.maxt;
}
```

References

[1] Nate Oh, *The NVIDIA Turing GPU Architecture Deep Dive: Prelude to GeForce RTX - Turing RT Cores: Hybrid Rendering and Real Time Raytracing*, www.anandtech.com/show/13282/nvidia-turing-architecture-deep-dive/5, 2018.

[2] Wenzel Jakob, Matt Phar and Greg Humphreys, *Physically Based Rendering: From Theory To Implementation*, pbrt.org, 3rd edition, 2018.

[3] Wenzel Jakob, *Nori: an educational ray tracer*, wjakob.github.io/nori

[4] Figure reference:
https://devblogs.nvidia.com/wp-content/uploads/2012/11/fig03-bvh.png