A Shared Memory Architecture for a Hybrid Real-Time Ray Tracing System

by

Ciaran Tuohy, B.A., B.A.I.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2014

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Ciaran Tuohy

September 2, 2014

Permission to Lend and/or Copy

Acknowledgments

I would like to first thank my supervisor Michael Manzke. He has provided me with valuable guidance through the course of this project and could always put me in touch with the right people when I was in need of assistance

I must also thank Michael Doyle and Colin Fowler who helped greatly in bringing me up to speed on current ray tracing research and also offered advice and guidance throughout. Their help has been invaluable.

Cathal McCabe from Xilinx went above and beyond to help me with the many technical issues I had working with the Xilinx hardware and software. Without him I don't know how I would have got this finished

I'd also like to thank my girlfriend and family for all their support this year and in all those that came before. They've helped me in every way I could have hoped for and I will always appreciate what they've done for me

Lastly, I need to thank my fellow classmates in IET, for helping me and for pushing me. Their company may have been all that kept me sane these last 12 months

Ciaran Tuohy

University of Dublin, Trinity College
September 2014

A Shared Memory Architecture for a Hybrid Real-Time Ray Tracing System

Ciaran Tuohy
University of Dublin, Trinity College, 2014

Supervisor: Michael Manzke

The ability to create interactive applications with high fidelity ray traced visuals, has long been a goal towards which the computer graphics community has strove. However, despite decades of research and advancements in computing technology it remains, for any non-trivial application, beyond the capabilities of modern commercial hardware. Furthermore, recent developments in computer architecture research suggest that performance gains will cease to continue in line with Moore's Law as result of the failure of power density scaling.

Consequently, contemporary thought in computer architecture has turned towards more novel approaches to tackle this problem. Fixed function hardware, which had long ago fallen out of favour with chip designers in favour of general purpose computing cores, has recently seen a return of significant interest. Motivated also in part by the need for power efficiency in mobile processors, this development has led to a rich vein of research into custom ray tracing microarchitectures.

In the near future similar fixed function microarchitectures for ray tracing may find themselves on commercial chips accompanying traditional CPUs and GPUs in a heterogeneous architecture. If this is to become a reality an efficient means of transferring large quantities of data into the custom device will be essential. This project presents an architecture wherein memory is shared between a CPU and custom peripheral. An evaluation of the system shows that this architecture is capable of efficiently delivering data to fixed function hardware without placing undue demand on processor time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Of the numerous rendering techniques that have been developed in computer graphics, ray tracing holds the greatest promise in terms of realistic visual fidelity. However the expensive process required to produce ray traced images has prevented its use in most interactive applications. Instead it has been restricted to producing high quality rendered stills or animated films. A recent trend in graphics and hardware research towards hardware accelerated ray tracing, which seeks to make ray tracing viable in an interactive capacity, provides the main motivation for this project. There has been a great variety of these systems designed utilising fixed function, programmable and hybrid approaches [8, 9, 10]. Many of the stages of the ray tracing pipeline have been implemented in these systems from ray generation and ray-scene intersection test, to shading. This trend towards application-specific hardware has been driven by a number of factors.

1.1.1 Performance

Ray tracing is a notoriously expensive process. It has been the focus of research since the 1960s [11] and yet today it still remains an extremely expensive process relative to the capabilities of modern consumer hardware. Indeed even in offline applications with enormous compute resources, only very recent advances in ray traced global illumination [12] have made the process feasible for application in 3D animated film production. Monsters University released in 2013 was the first Pixar film to feature true ray traced global illumination [13]. Much of this computational cost arises from a small few algorithms. In many cases, such as rasterisation [14] and ray-scene intersection testing [15] dedicated hardware has been demonstrated to provide superior performance to software implementations on general purpose hardware.

1.1.2 Dark Silicon

In 2011 Esmaeilzadah et al. presented their paper entitled "Dark Silicon and the End of Multicore Scaling" [16] a thorough analysis of the projected failure of multicore scaling. The Dark Silicon concept refers to the paper's conclusion that even at current lithographic scales (22nm) approximately 21% of the logic on a chip must remain powered off. Their projections suggest that this will increase over time to the point at which only 50% of on-chip logic can be powered simultaneously. The reason that this is occurring is the failure of so called Dennard Scaling which states that power density of chip logic will remain constant as manufacturing scales decrease, i.e power per area will remain constant [17]. This failure in part led to the push toward multicore chip architectures and is now beginning to lead such systems into the realm of diminishing returns.

A conclusion that may be drawn from this is that alternative architectural approaches are required if we wish to see the performance gains to which we have become accustomed. Since it would make little sense to fill all of our chips with general purpose hardware if we can only power a certain percentage of it simultaneously, then it follows that fixed function or application specific hardware might be implemented along side the traditional general purpose processing cores. In order to maximise the benefit from this additional custom logic that will be available it would be prudent to only implement custom microarchitectures to accelerate frequently used algorithms. Suitable algorithms within the field of ray tracing include ray-scene intersection tests and bounding volume hierarchy construction.

1.1.3 Power Efficiency

The last few years has seen what has been popularly referred to as the"Mobile Revolution", the dramatic increase in popularity of mobile computing devices such as smartphones and tablets. Development of these devices has advanced rapidly and recent mobile processor designs boast quad core processors clocking at over 2GHz [18]. However, advancements in battery technology have lagged behind the pace of advancements on the processing side. The need for portability in these devices restricts the size of battery that can be utilised meaning that low power consumption is heavily prioritised in mobile processor design. Most of these processors are be custom system on chip (SoC) implementations based on low power RISC processors from ARM and featuring dedicated hardware for certain important tasks in mobile communications such as audio decoding and image processing. Chip designers with a history in graphics such as NVIDIA have already begun to implement some graphics related custom microarchitectures such as for rasterisation [19].

1.2 Contribution

Many of the custom ray tracing microarchitectures discussed, such as those by Doyle [4] and Schmittler et al. [8], are designed and evaluated individually. Clock frequencies are assumed or taken for the system independent of all else and resource utilisation figures similarly consider only the microarchitectures in isolation. However, if we are to consider that future processors may feature heterogeneous architectures consisting of multicore, manycore and fixed function components, then it becomes obvious that we must also have an efficient means of communicating between the main processor systems and the custom microarchitectures. As these systems are typically designed with modularity in mind, they can not consider the limitations that may be imposed by such a communication interface in practical terms.

This thesis outlines a shared memory architecture to facilitate efficient communication between a processor and custom hardware using the BVH builder outlined by Doyle et al. [4] as a design target. This unit is suitable as it requires a large quantity of data movement to and from the custom microarchitecture and also some control signalling from the processor to the hardware. The goal is to produce a system that would allow for the development of a hybrid hardware/software ray tracing system.

Chapter 2

Background

This background section will be primarily focused on the three research areas most closely related to the project, namely: Ray Tracing and Global Illumination, Acceleration Data Structures and Ray Tracing Microarchitectures. Each of these fields will be discussed providing context for the investigation and justification for the details of the proposed research.

2.1 Global Illumination

Global illumination is a blanket term used to refer to a set of advanced rendering techniques used for 3D scenes which consider both direct illumination and indirect illumination. Direct illumination is considered to be illumination that results from light directly incident on a surface from a light source, whereas indirect illumination results from light incident on a surface that was itself reflected off other surfaces in the scene. In most cases, reflections, refraction and shadow casting are examples of visual effects that can be achieved through global illumination.

A subset of global illumination techniques use the ray model of light transport in order to compute the lighting value at a particular point in the scene. This subset of techniques is sometimes referred to as ray-based global illumination. Many of these techniques such as Path Tracing [20] are extremely computationally expensive but produce excellent results, as such their use has traditionally confined to offline applications. Other techniques such as ray casting and ray tracing are suitable for real time applications.

2.1.1 Ray Casting

Amongst the simplest of the ray-based global illumination techniques is the method of ray casting. This sometimes ambiguous term is used here to refer to a ray based illumination model that only accounts only for direct illumination. As such it can provide hard shadows but no effects like reflection or refraction. However it is a clear and simple demonstration of the principles on which the more sophisticated techniques build.

Originally proposed by IBM researcher Arthur Appel in 1968, the ray casting technique considers a scene that is projected onto a two dimensional viewing plane by some perspective projection [11]. Consider this plane to be some distance in front of our virtual camera and that the plane is divided into uniform segments each corresponding to a pixel on the display of the corresponding device. A ray is then cast from the virtual camera out through each of the segments of the viewing plane into the scene. Each ray then searches the scene to find its first point of intersection with a scene object i.e. the closest point in the scene to the camera along that line.

We now have the intersection point but because we have gone backwards from the camera rather than forwards from the light we do not know what colour this point is meant to be. As such our next step is to determine this colour and “shade” the pixel. We do this by tracing additional rays from our point of intersection to each light source in the virtual scene. These rays are often called “shadow rays” as if there is another primitive blocking the path of the ray then the point will be shadowed by that object. Once each of these shadow rays have been traced to the light sources we know what the incident illumination at the intersection point is. We can then use this in conjunction with the material properties of the object on which the intersection point lies to produce a limited solution to the rendering equation. This is described by Kajiya as [20]:

$$I(x, x') = g(x, x')[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''] \tag{2.1}$$

where:

$I(x, x')$ is related to the intensity of light passing from point x' to point x.

$g(x, x')$ is a geometry term.

$\epsilon(x, x')$ is related to the intensity of emitted light from x' to x.

$\rho(x, x', x'')$ is related to the intensity of light scattered from x" to x by a patch of surface at x'.

The ray casting solution is extremely simple. It completely ignores the whole recursive integral which accounts for light not directly incident on the surface.

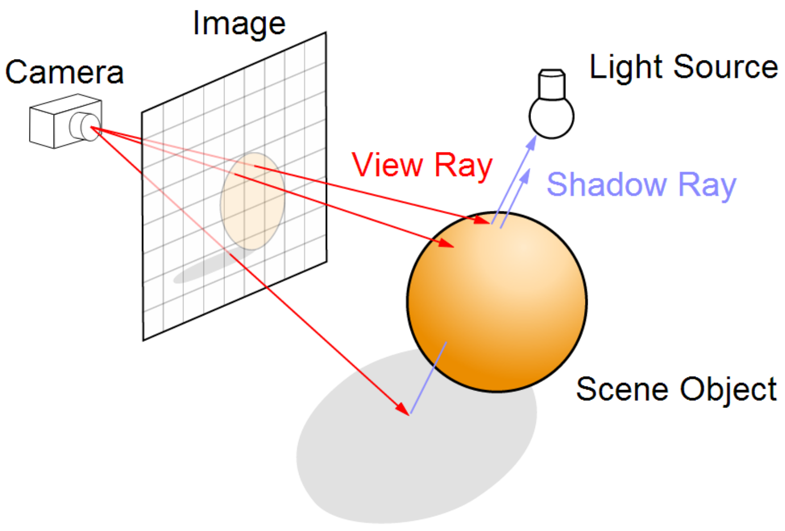


Figure 2.1: An Illustration of Ray Casting.

2.1.2 Ray Tracing

As mentioned previously, the ray casting method as described is capable of taking into account direct illumination only, meaning that while it can produce shadows, it is incapable of producing other effects that we would expect to see in the real world; such as reflections and refractions. With this in mind an extension to ray casting was proposed by Whitted [21] which is referred to as ray tracing or "Whitted ray tracing."

Consider how the ray model of light behaves in the real world. Light is emitted from various sources and is cast out into the environment. It impinges on objects and is reflected off those surfaces. Some of those rays will reflect directly into the eye (as we considered with ray casting) however many others with continue through the environment to impinge on other objects and reflect again. This can happen many times with energy lost upon each reflection.

Whitted ray tracing aims to model this effect using backwards ray tracing (from the camera) rather than forward ray tracing (from the light). It differs from ray casting only once we reach the initial intersection. Here instead of just casting shadow rays to the light sources, additional rays referred to as secondary rays are cast. If the surface is highly reflective then a reflection ray will be cast into the scene. Additionally if the surface is considered to be transparent then a refraction ray will be computed in accordance with Snell's Law.

A further difference between ray casting and Whitted's approach is the recursive nature of ray racing. Each of these secondary rays will be traced until they intersect with another piece of scene geometry whereupon the whole process starts again. However, as previously stated upon each intersection energy is lost, accordingly we limit the number of recursive secondary ray traces. Choosing the number of recursive bounces to allow leads to a trade off in image fidelity against performance. The more steps allowed the better the visual result however each ray traced comes with an associated cost and allowing too many would impact on the frame-rate.

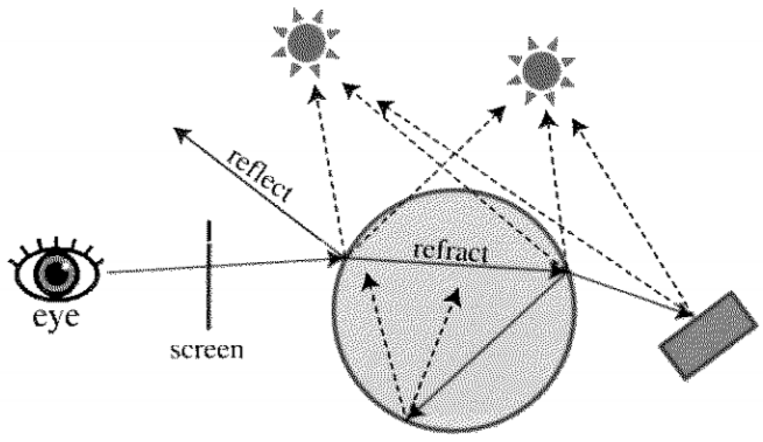


Figure 2.2: Illustration of Ray Tracing [2].

2.2 Acceleration Data Structures

In addition to the potentially very large number rays which must be traced, we must also consider the complexity of the scene itself. It is not uncommon for scenes to consist of thousands or even millions of triangles. If the geometry is left unstructured, each primary and secondary ray would have to run an intersection test with every triangle viewable in the scene, in order to determine which ones it interacts with. This would make ray tracing impossible to compute in real time if we were to take the naive approach. Indeed, in his paper, Whitted determined that between 75% and 95% of the total compute time of his ray tracing method was consumed by ray intersection tests [21]. Clearly it is desirable to find methods to improve how rays find their intersection points.

A solution is to use acceleration data structures (ADS) to organise scene geometry in such a way that allows for a significant reduction in the number of intersection tests that need to be conducted. There are a number of spatial data structures that are used to achieve this such as Octrees, Binary Space Partitioning (BSP) trees or k-d trees. These are all referred to as spatial index structures as they divide the scene into certain spatial sections which are then navigated rather than the scene geometry itself.

Each of the above listed structures are so called hierarchical structures as they use multilevel spatial subdivisions. However there also exists another class of spatial index structure called a flat data structure where no multilevel hierarchy is employed. An example of such a structure is Fujimoto's uniform grid [22]. The uniform grid divides the scene into a regular lattice structure of rectangular volume elements. A ray can be traced through the grid and only those primitives overlapping with those grid cells with which the ray intersects need be tested for intersection, potentially culling a huge amount of geometry. Wald et al. present an efficient means of traversing these uniform grid structures [23]. There are a number of beneficial features of the flat spatial data structure, such as its relative simplicity, constant memory usage and ease of update for handling dynamic scenes. However they typically perform poorly in many scene types due to an inefficient handling of primitive-dense areas, it is possible for a very large proportion of scene primitives to end up in only a few cells of a uniform grid.

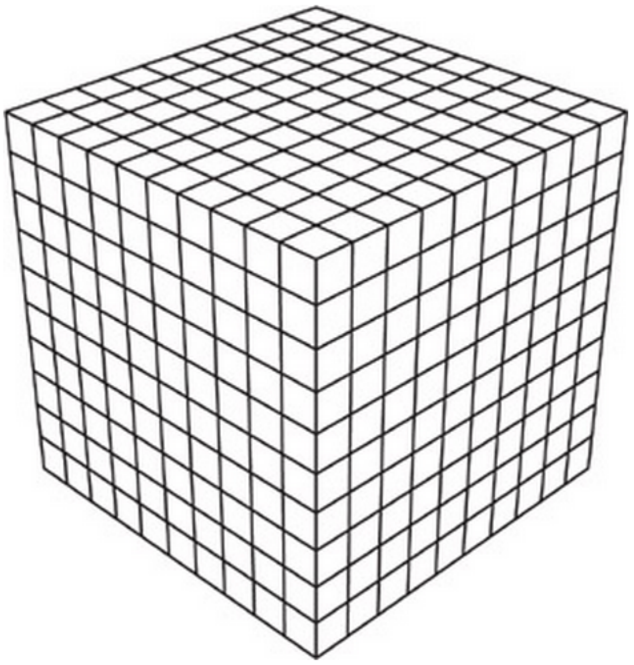


Figure 2.3: A perspective view of the uniform grid [3].

2.2.1 Spatial Subdivisions

Amongst the hierarchical structures there are two main methodologies by which they are conceptually organised. The BVH with which this project is primarily concerned is an object partitioning structure [24], another example is the bounding interval hierarchy [25]. In these methods the spatial divisions are made so as to partition the scene geometry.

On the other hand we have structures like the kd-tree which uses spatial partitioning [26]. The kd-tree is a binary spatial data structure that is defined not by regions but by splitting planes along the major axes. At each node level the scene is further divided in space and the scene geometry is mapped to the side of the plane with which it overlaps. This makes traversal incredibly simple. At each node on the tree an intersection test is perforedwith the relevant splitting plane. If the intersection occurs before or after the scene boundary then we know which child node must be tested. However if it occurs within the scene boundary then both child nodes must be checked.

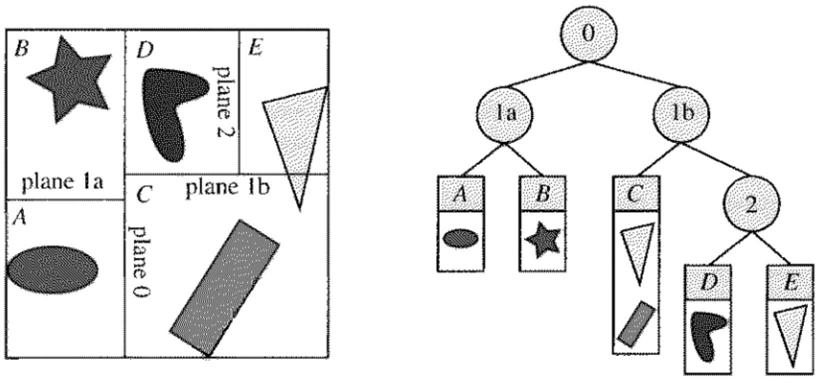


Figure 2.4: A sample kd-tree [2].

2.3 Bounding Volume Hierarchy

As previously stated, the main ADS with which this investigation is concerned is the bounding volume hierarchy. A bounding volume is considered to be a volume that encloses an object or set of objects, in this case geometric primitives. We want the bounding volume to be a more simple shape than the geometry it contains to facilitate easy intersection tests. For this purpose the Axis Aligned Bounding Box (AABB) is a popular choice. This is simply a cuboid bounding volume that is axis aligned with the world's coordinate system. The AABB is popular as there exist simple, efficient ray-AABB intersection algorithms [27], it is also much less expensive to compute than other geometric proxies like the oriented bounding box (OBB) [28] or discrete oriented polytrope (K-DOP) [29].

The BVH is a tree structure of bounding volumes where each parent node encloses their child nodes. The consequence of this is that the root node of the tree structure is a bounding volume that encloses all of the geometry in the scene. Organising triangles in this way significantly reduces the number of intersection tests required for each ray. Rather than testing each triangle in turn it can now simply test first against the root node of the structure. If the test returns true, it can then begin to recursively traverse the tree. Each time it will test against the child bounding volumes of a successfully tested parent volume until a leaf node representing a single triangle or small subset of triangles is reached.

While the use of a BVH will undoubtedly improve ray tracing performance they do

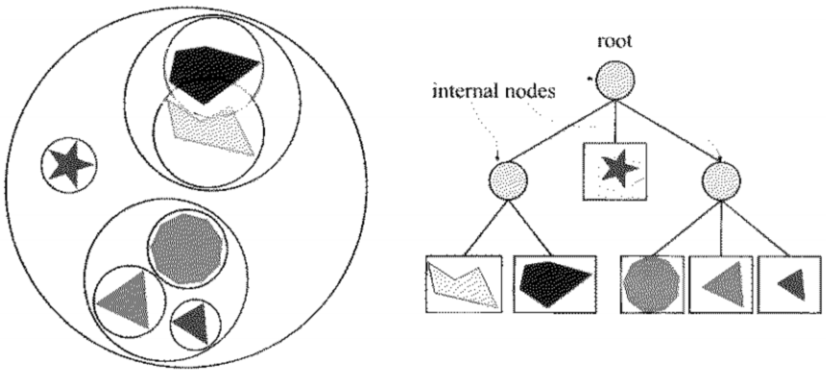


Figure 2.5: Sample BVH [2].

not come without a cost. It must be remembered that the BVH is geometry-dependant. This means that in dynamic scenes where any movement of scene geometry occurs the previous BVH can no longer be used. The two most common ways of handling scene dynamics are:

- The existing BVH can be refitted to accommodate the changes in scene geometry. This has the notable advantage of being relatively simple to compute. However it limits the amount and types of movement allowed in the scene as the quality of the tree can degrade over time if there is too much movement and this will have an impact of ray tracing compute time [30]. The operation is of the order O(n).
- The BVH can be rebuilt entirely each frame. This method does not restrict scene dynamics in any way but takes longer to execute. Rebuilding trees is an O(n log n) operation.

2.3.1 BVH Construction

There are a number of methods that have been developed to construct BVHs however each is not without their deficiencies. There are two important properties of a BVH construction process, build time and tree quality. Tree quality is typically considered as the average number of ray/scene intersections required to find the appropriate primitive. The different approaches perform better in one area than the other as there is a trade off between quality and build speed.

Both divisive (top-down) and agglomerative (bottom-up) approaches tend to perform will in both metrics with divisive methods having a slight edge in terms of build time and agglomerative methods, slightly higher tree qualities. Other construction methodologies such as the linear BVH, strongly favour build time over quality. The system under investigation seeks to further improve the construction time of divisive BVH's using hardware acceleration.

Divisive Method

The top down method presented by Wald in 2007 is the current standard for top-down BVH construction [31]. Wald noticed certain optimisations that had occurred in relation to the construction of k-d trees and noticed that the same methods were at least as applicable to BVHs if not more so. Typically when using a divisive clustering method, the way to achieve an optimal tree structure in terms of traversal cost, is to use a greedy Surface Area Heuristic (SAH) partitioning method. The SAH, originally developed by MacDonald and Booth, is a reasonably simple concept [32]. It is a greedy cost function that estimates traversal cost if the volume is divided in a particular fashion. It generates this cost by considering the surface area of the two volumes that would be produced if the current volume V containing N primitives were to be split into two parts V_L and V_R, each containing N_L and N_R primitives respectively. K_T and K_L are implementation specific constants.

$$Cost(V \rightarrow [L, R]) = K_T + K_I(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R) \tag{2.2}$$

The issue with this method is that there are $2^N - 2$ partitions which must be tested which is an extremely costly process. A solution is to implement a so called binned SAH. Here instead of considering all possible split pairing Wald subdivides the volume into K equally sized volumes called bins equidistant with respect to their centroids. By subdividing the volume in this manner he reduces the number of possible partitions from $2^N - 2$ to $K - 1$, a dramatic improvement in efficiency.

It is this binned SAH method that is of particular interest for this investigation. Developements have been made that allow for a parallel implementation with improved performance [33]. A further development has come from research by Doyle et al. who presented a binned SAH construction microarchitecture that hardware simulations suggest will provide enormous speed up over software implementations [34]. Algorithm 1 shows generalised form of the SAH construction approach [4].

ALGORITHM 1: Top-Down BVH Construction [4]

```
$buildBVH(nodeN)$;
**if** *terminationCondition(N)* **then**
    $makeLeaf(N)$;
    return
**else**
    $bestCost \leftarrow \infty$
    **for** *all candidate partitions P of N* **do**
        **if** $expectedCost(P) < bestCost$ **then**
            $bestCost \leftarrow expectedCost(P)$;
            $bestPartition \leftarrow P$;
        **end**
    **end**
    $partitionNode(N, bestPartition)$;
    $buildBVH(N.leftChild)$;
    $buildBVH(N.rightChild)$;
    return;
**end**
```

Agglomerative Method

The leading work in relation to the agglomerative approach to tree construction was carried out by Walter et al in 2008. Their efforts attempted to accelerate the greedy agglomerative construction algorithm for which the $O(N^3)$ solution is shown in Algorithm 2.

This algorithm takes a set of singleton clusters C where each cluster is a single primitive from P, the set of all primitives. It then loops until it has reached the root node and only one large cluster remains. At each iteration, the algorithm checks every cluster against every other cluster to see which two are nearest neighbours. When the algorithm has identified the best pair, it combines the two into a new cluster C and adds it to the set of clusters while removing the components of C from the set.

The issue with this approach is that the search for the nearest neighbour takes an extremely long time during the early stages being an O(n^2) process. Consider that

ALGORITHM 2: Greedy Agglomerative BVH Construction [35]

```
Input: Scene Primitives $P = P_1, P_2, \dots, P_N$
Output: BVH root node
Clusters $C = P$;
while $Size(C) < 1$ do
    $Best = \infty$ ;
    for $C_i \in C$ do
        for $C_j \in C$ do
            if $C_i \neq C_j$ and $d(C_i, C_j) < Best$ then
                $Best = d(C_i, C_j)$ ;
                $Left = C_i; Right = C_j)$ ;
            end
        end
    end
    $ClusterC' = newCluster(Left, Right)$ ;
    $C = C - Left - Right + C'$ ;
end
return $C$;
```

at the beginning every primitive is its own singleton cluster, this means that initially every piece of geometry must be checked against every other piece of geometry.

This was later accelerated by storing the remaining clusters in a k-d tree created using a divisive approach [36]. This creates the ability to limit the number of candidates considered for the nearest neighbour thus accelerating the process. Further optimisations enabled development of two different methods for constructing a BVH based on this idea of using the kd- tree. The first is a heap based method that uses a min heap, to preserve dissimilarity information across outer loop iterations. The second method, referred to as Locally-ordered clustering, attempts to automatically cluster nearby nodes on a tree if it can be proven that the algorithm would eventually cluster them anyway.

Later improvements to this this greedy agglomerative method were developed, the new more efficient process is called approximate agglomerative clustering [35]. This method relaxes the constraints of the greedy approach and manages to significantly improve on build time with only a slight compromise in tree quality.

2.4 Ray Tracing Hardware

Considerable research has already proposed a number of custom micro-architectures for ray tracing applications. These can be broadly divided into three categories, fixed-function, fully programmable or hybrid fixed-function/programmable. Of these three categories the most relevant to this investigation is the fixed function approach which offers less versatility but more in terms of performance and power efficiency.

2.4.1 Fixed Function

In 2001 Kobayashi et al. presented their application-specific integrated circuit (ASIC) for photo realistic image synthesis [37]. Their system uses a hardware accelerated 3D line generator to find objects likely to intersect with traced rays. Functionally the unit calculates all view independent light first and then for subsequent frames the ray tracing hardware is activated to produce the view-dependent illumination. The spatial data structure utilised is a uniform grid.

Also in 2001 came a study by Todman and Luk who tested he feasibility of using reconfigurable hardware for real-time ray tracing [38]. They focus on accelerating the most expensive aspect of the ray tracing algorithm which is the acceleration algorithm. They conclude that based on their feasibility study, real-time interactive ray-tracing could be achieved on 10 concurrent FPGAs. This investigation will also seek to use FPGA technology but not specifically for ray tracing acceleration, so the required resource quantity suggested by this report should not be relevant, additionally the age of the paper is a factor to consider.

In 2002 Schmittler et al presented SaarCOR a system using a three part design for ray tracing [8]. The three core components are the ray generation and shading unit, the ray tracing core and the memory management unit. The system utilises the technique known as packet tracing to achieve further gains by allowing multiple rays to be grouped together into packets for efficient calculation [39]. Of particular interest is an extension of this concept brought by Schmittler et al. two years later where they were able to develop a complete ray tracing pipeline on a single FPGA chip [40]. In this study, with the board running at 90MHz they were able to achieve frame rates between 20 and 60 for a range of 3D scenes with support for texturing and multiple light sources. The spatial data structure utilised is similar to a system proposed by

Wald that utilises a multi kd-tree structure [41].

2.4.2 Fully Programmable

A good recent example of a fully programmable architecture is the TRaX system [42, 43]. TRaX (standing for Threaded Ray eXecution) is essentially a series of parallel general purpose processing cores which share an L2 cache. Each of these cores contains a number of thread processors each of which contains its own function unit. The motivation for this system approach is the highly parallelisable nature of ray tracing algorithms. Results from the system were extremely positive with good performance seen at 500MHz and scene geometry comparable to contemporary video games. The creators of TRaX, Spjut et al. have also presented a similar system suitable for use on mobile platforms [44].

Further work has seen the concept behind TRaX expanded to create the STRaTA architecture [9]. Additions to the design seek to reduce reduce bandwidth demands and also improve the systems power consumption. The most relevant of these additions from the perspective of this project is that the BVH spatial index is decomposed into treelets [45]. The purpose of this is to maintain smaller tree structures capable of fitting entirely on the efficent L1 caches for the different processors.

2.4.3 Hybrid Fixed Function - Programmable

In 2003 Sanchez-Elez et al. presented their mapping scheme of an optimised, octree based ray tracing algorithm [46]. They also implemented it on a SIMD reconfigurable architecture. This system operating at 300MHz consumes only 1Watt and supports primary, shadows and reflection rays. Octrees are used as the ADS.

Other approaches include the Mobile Ray Tracing Processor (MRTP) which seeks to solve issues with SIMD utilisation caused by diverging ray distributions [47, 48]. MRTP uses reconfigurable stream multiprocessors (RSMPs) for high data-path utilisation. These RSMPs are used to execute one of three kernels, ray traversal, ray intersection and shading. There is very little fixed function hardware in this system, the only prominent feature is the hardware ray generator, it is mostly a custom programmable ray tracing architecture like those mentioned in the previous section.

Lastly we have the Samsung Reconfigurable GPU based on Ray-Tracing (SGRT) GPU designed to enable ray tracing on Mobile devices [49, 10]. The system is composed of two main parts. The first is a traversal and intersection (T and I) unit based on the T and I engine presented by Nah et al in 2011 [15]. The fixed function T and I engine was adapted to support BVHs rather than kd-trees as BVHs are more suitable for dynamic scenes. The second main component is the Samsung Reconfigurable Processor (SRP). The SRP is composed of a combination of a very long instruction word processor and a coarse grained reconfigurable array. The SPT supports C languages and reconfigures hardware for computationally expensive tasks at compile time. Above the SRGT cores the top layer of the GPU is an ARM processor that in this instance is used to construct the BVHs. This aspect of the design could be enhanced by additional fixed function hardware.

2.5 The Hardware BVH Builder

As mentioned previously this investigation is centered around the hardware unit for BVH construction presented by Doyle in his doctoral thesis [4]. This custom microarchitecture provides fast and efficient construction of BVHs. It implements a divisive construction method which uses the binned SAH to determine the partitioning point. Evaluation of the system has demonstrated that it produces a high quality tree with good construction times and with significantly lower power consumption relative to the same operation performed on general purpose hardware

2.5.1 Top Level System Architecture

At the top level of abstraction the BVH builder is a unit composed of two major components, upper builder and subtree builder. The upper builder is designed to directly interface with external memory while the subtree builder units interface with the output from the upper builder. As its name suggest the upper builder constructs the top few nodes of the hierarchy until it generates a node consisting of a number of primitives less than a threshold value. This node is then delegated to one of the connected subtree builders for further construction.

The system is separated in these conceptual parts for a number of reasons. Firstly

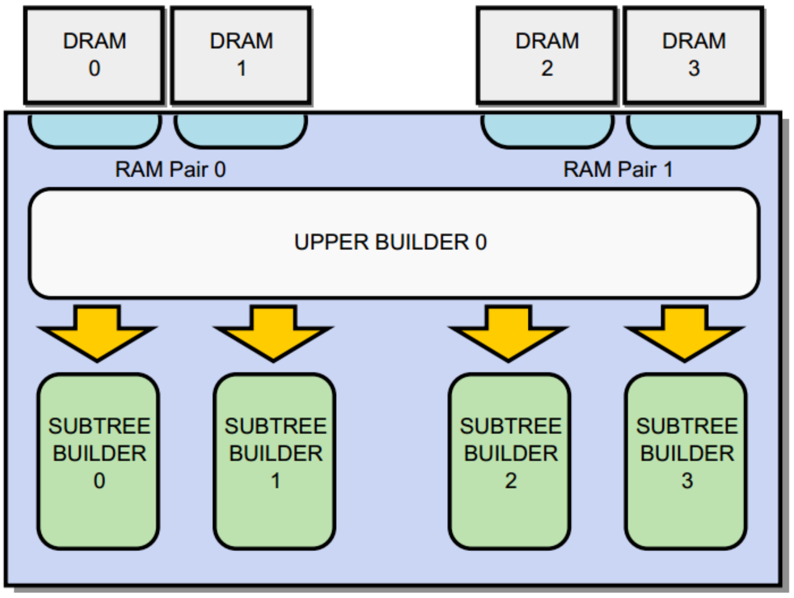


Figure 2.6: Top Level Architecture of the BVH construction unit [4].

because at each split the scene is wholly split into the two sub nodes the process is highly parallelisable, however the benefits of parallelisation are far greater at the bottom of the tree than at the top. As such a single threaded approach may be taken initially and then parallelised once the upper nodes of the tree have been constructed. Another reason for the design being laid out in this way arises from the nature of the BVH construction algorithm. In order to start computation of the SAH for a particular node we need access to all of the scene data associated with that node. Obviously this is going to be limited by the interface with external memory. However the subtree builders utilise high bandwidth internal memory to store primitives which gives superior performance to the upper builder, the node size that the subtree builders can accept is limited by the size of this internal memory.

2.5.2 The Subtree Builder

The subtree builder forms the main component of this overall system. It is also the component most directly relevant to the memory architecture that this thesis will outline. What follows is a detailed description of the interface to and architecture of this component.

Architecture

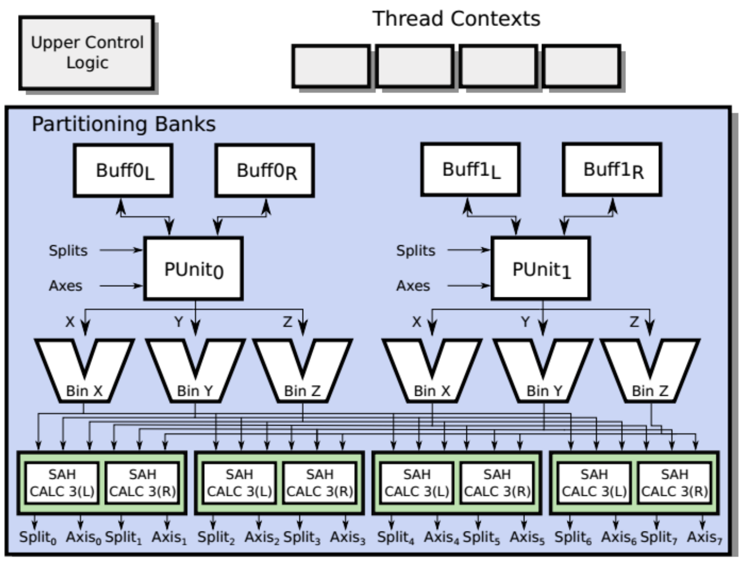


Figure 2.7: Top Level Architecture of the Subtree Builder [4].

As can be seen in the top level diagram shown in Figure 2.7 the subtree builder consists of a number of major components which dictate the operation of the unit. Certain features of the design may appear confusing at first glance, such as the buffer pairs. However it is important to recall that the subtree builder has been designed to execute the binned SAH based construction method outlined by Wald [31]. The recursive nature of this algorithm heavily influences the design of the hardware.

The first significant component of the subtree builder is the pair of primitive buffers, two such pairs are seen in Figure 2.7 namely Buff0_L - Buff0_R and Buff1_L - Buff1_R. These are composed of high performance on-chip memory similar to that used in an L1 cache. They are used to store scene primitives passed to the subtree builder. As such it is the size of memory allocated for these buffers that determines the threshold point at which the upper builder will delegate to the subtree builder. The size of these buffers is configurable but is generally considered to be of a size capable of storing a number of primitives in the order of thousands.

Each pair of buffers is then connected to a Partitioning unit (labeled PUnit$_0$ and PUNIT$_1$ in Figure 2.7) by a bidirectional channel. The logic within the partitioning unit will take a determined value for an SAH split and begin organising the primitive buffers into two separate data vectors depending on which side of the split they are located. The double buffering structure allows the partitioning unit to read from one buffer say Buff0$_L$ and write the reorganised vectors into its partner buffer Buff0$_R$. Then after the next SAH split has been calculated it will perform the reorganisation procedure in reverse this time reading from Buff0$_R$ and writing to Buff0$_L$.

Of course the partitioning unit needs a value for the split point and the axis along which it is found. This is determined by the split calculation hardware comprised of binning units and SAH calculators. The binning units take the scene primitive AABBs as input and determine which SAH bin the AABB belongs to along the particular axis. There is one binning unit for each the X, Y and Z axes. After this calculation the binning unit will pass the AABB along with its determined bin location to the SAH calculation units. These units will wait until all of the AABBs have been binned and passed on to them and will then calculate the cost value of each possible split by way of the surface area heuristic and output the lowest cost split along with its corresponding axis.

Interface

As previously explained the subtree builder interfaces primarily with the upper builder. This interface consists of a number of important components. Once upper builder construction is complete and the requisite node size has been reached the scene data corresponding to the node must be passed down to the subtree builder. This data consists of a series of 216 bit values. These 216 bits are comprised by a 24 bit integer index and six 32 bit floats corresponding to vertex coordinates. It is important to note that only six float values are needed as primitive are not stored directly but rather as their AABBs. Hence the primitives can be represented by two AABB vertices, the front-bottom-left corner and the back-top-right corner. If we consider the gains relative to directly storing a triangular primitive then 96 bits are saved for every one of the thousands of primitives expected to be stored in the subtree builder. The 24 bit index is used to reference to the scene primitive corresponding to a particular AABB. This data is delivered to the subtree builder through a 216 bit channel.

Additionally the subtree builder has an output channel intended to communicate directly with memory that will write the results of the calculations into memory. Certain control parameters which connect to the subtree builder in the form of 24 bit channels represent memory addresses to which the output of the subtree builder will be directed. These control parameters allow for multiple subtree builders to function in parallel but still write their results into the correct locations in memory to produce a valid tree structure.

Lastly there are a number of constant and control signals shared between the control logic which dictates the operation of the unit. The constant signal tells the subtree builder how many primitives are about to be sent to it. When it then detects that it has received the number of primitives corresponding to that signal, it sets high it's READY channel indicating to the logic that it is ready to begin executing. The control logic will then set the units GO channel high and the subtree builder will begin calculation.

Chapter 3

Hardware Platform

The goal of this investigation was to develop a memory architecture to enable efficient communication between a traditional processing system and a custom microarchitecture in the form of the hardware BVH builder. In order to design and evaluate such a system a suitable platform was needed. Considering that a primary project goal was to develop a system to make this kind of communication function in a practical environment it was decided that a hardware platform that would allow for real hardware implementations would be required. The development platform that most closely met the conditions imposed by the project goals was the Xilinx Zynq-7000 Series

3.1 Zynq-7000 Series

The 7000 series is a recent series of chips from Xilinx that are referred to as "All Programmable System on Chips" (AP SoCs). The phrase "all programmable" refers to fact that the user is able to program the chips both in terms of hardware and software. The main feature of the 7000-series chips is that they are composed of a Processing System (PS) with a dual core ARM Cortex A9 processor surrounded by a Programmable Logic (PL) subsystem using a Field Programmable Gate Array (FPGA). These two systems are capable of running independently and can communicate by high speed Advanced eXtensible Interface (AXI) connections. There are over 3000 internal interconnects between the processing system and programmable logic which allows for up to 100Gb of available bandwidth.

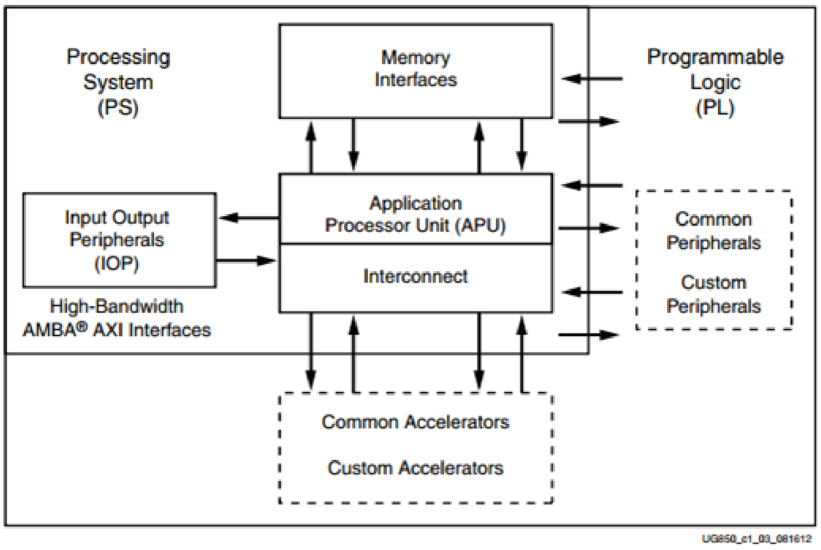


Figure 3.1: High level block diagram of Zynq system [5].

The block diagram shown in Figure 3.1 shows the general high level structure of the Zynq chip. We can see the processing system consisting of an APU, memory interfaces and IO peripherals. We also see that the PS and PL are very tightly connected. One interesting thing to note is the distinction made between common peripherals and accelerators and custom peripherals and accelerators in the PL subsystem. A number of common systems we would typically expect from a processor, like the ability to output video, are absent from the basic chip design. These common peripherals may be implemented instead using the FPGA and are very simply instantiated using the software provided by Xilinx. Custom hardware on the other hand may be directly designed by the user or a 3rd party and integrated into the Zynq system through the FPGA.

3.1.1 Processing System

Figure 3.2 shows an overview of the features of the Zynq processing system's APU. It can be seen that the system features two ARM Cortex A9 cores each with their own single and double precision floating point unit, a NEON media processing engine which supports Single Instruction Multiple Data (SIMD) vector operations and 32KB

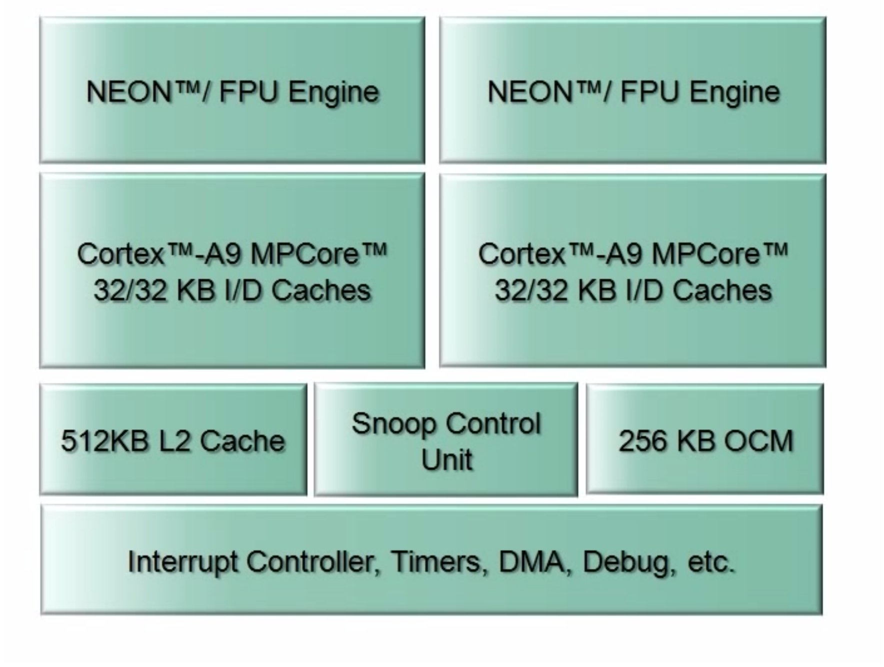

NEON™/ FPU Engine
NEON™/ FPU Engine
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
512KB L2 Cache
Snoop Control Unit
256 KB OCM
Interrupt Controller, Timers, DMA, Debug, etc.

Figure 3.2: Zynq Processing System Components [6].

instruction and data caches. These processors can operate in a number of different modes allowing each of them to run separate operating systems if desired.

It can also be seen that there is a shared 512KB L2 cache with a snoop control unit to maintain cache coherency. An interesting feature of the APU is the 256KB dedicated on-chip memory. This is larger than one would normally expect and is connected separate to the L2 cache. This is to enable operation of a real time OS such as FreeRTOS without the need to deal with cache misses [6].

Lastly we see the APU peripheral connections, two of which, are of particular interest to this investigation, the DMA and interrupt controllers. The DMA (direct memory access) engine allows for a high speed delivery of data from memory to another part of the system. The interrupt controllers allow for processes executing in the programmable logic to interrupt and trigger routines in the APU.

A feature of the Zynq chip is that it is processor centric. At power-up the processor will start and will then determine how much of the rest of the board will be powered on. The processor can choose to function independently or can choose to power up any of the programmable logic or IO peripherals that it needs. This a useful feature in terms of power consumption as only the parts of the board that are being used will draw power.

3.1.2 Programmable Logic

The programmable logic portion of the chip will consist of a Xilinx Artix-7 or Kintex-7 FPGA depending on the model of board. However both of these systems contain the same basic components, just in different quantities. Four primary components make up the Xilinx FPGA systems, Logic Cells, DSP Slices, Block RAMs and Clocking Resources

Logic Cells are the basic building blocks of custom logic in an FPGA. They are composed of a six input Look-up Table with a dual flip flop. The logic cells in Xilinx FPGAs can also be utilised as distributed memory or shift registers [1]. Logic cells also contain connections to adjacent cells allowing them to be combined together to produce complex logical systems. The main use of the Logic cell is in expressing combinatorial logic and for pipelining.

The DSP slices are small powerful units targeted at digital signal processing functions. They consist of a 25bit x 18bit multiplier, 25bit Pre-Adder, 48bit accumulator that can function as an arithmetic logic unit and a 96bit accumulator. The presence DSP slices is evidence of how the devices are tailored for signal and image processing applications however they could equally of use in graphics. Filtering, state estimation and other such processes would draw heavily on the DSP slice resources.

Block RAMs (BRAM) are the main memory component of the FPGA. They are essentially a configurable, dual port 36kb memory block with built-in first-in first-out (FIFO) logic. Their primary use is in local storage of data, multiple BRAMs can be combined together to create larger local storage capacity. However, due to their built-in FIFO logic they can also be used to handle re-timing between different system components which might be reading and writing data at different rates.

Finally, there are the built in clocking resources. This is to service a need that often arises where different components must function on independent clock domains. The FPGA achieves this using what are called mixed-mode clock managers which can be configured to meet a number of clocking criteria.

There exists multiple ways to implement custom logic on the FPGA. There is the traditional method which is to use a hardware descriptive language (HDL) such as VHDL or Verilog. These are languages which can by used to express logical statements and can be used to simulate computer hardware. If run through a synthesis tool then these HDL files will be used to configure the FPGA resources to reflect their logical expressions. Another way is to use prepackaged intellectual property (IP) provided by Xilinx or other third parties. This IP will typically be designed to operate on the Zynq system and to interface with AXI standard. Lastly Xilinx development software offers what it refers to as High Level Synthesis (HLS). This system will take an algorithm expressed in C/C++ and use it to synthesise hardware, foregoing the need for a direct implementation in a HDL.

3.1.3 ZC702 Evaluation Board

There are a number of different products produced which use a Zynq-7000 series chip, from the relatively low cost Zedboard and MicroZed produced by Avnet, to Xilinx's powerful, fan-cooled ZC706. The board which was utilised in this investigation is the ZC702 Evaluation Board. The board features the typical Dual ARM Cortex A9 processor capable of clocking at 866MHz and a Atrix-7 FPGA with 85,000 Logic cells, 560KB of BRAM and 220 DSP slices available [50]. In terms of available logic this equates to roughly 1.3 million gates in the equivalent application specific integrated circuit (ASIC).

In addition to its processing system and programmable logic resources the ZC702 has an array of common peripherals and connections available such as HDMI, USB UART, Micro USB and Ethernet which can all be seen in Figure 3.3. There are also a number of ways to boot the device if running an OS. It can boot from the 128Mb of QSPI flash memory available, from an inserted SD card or via a JTAG connection to a host machine. While the board does not natively support video without a HDMI controller and OS installation with frame buffer support, it is possible to establish a serial connection over USB UART and interface directly with the device in that fashion.

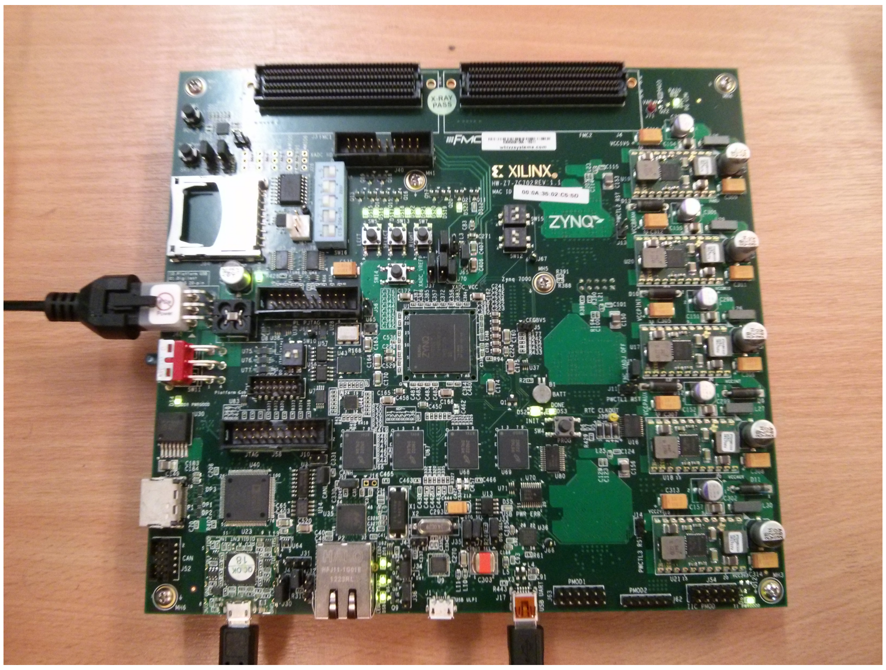


Figure 3.3: ZC702 Evaluation Board.

Chapter 4

Design and Implementation

Before beginning implementation, a significant amount of experimentation with sample system designs provided by Xilinx was carried out. The goal of this experimentation was to gain insight into the operation and limitations of the board and accompanying software which would inform the design process. It was important to try to envision how the proposed shared memory architecture could be facilitated on the ZC702. Ultimately it was concluded that the design would consist of three levels.

4.1 Design Elements

Hardware
: The hardware level design covers the necessary circuitry to deliver the data and control signals needed for the custom module to operate and also features a custom shim to allow the module to interface with the AXI standard on which the PS-PL communication is based. Hardware interrupts allow for the programmable logic to communicate back to the processing system when certain important events occur. This data delivery method and custom interface are then to be tested for resource utilisation and how it affects the clock speed limits and operating capacity of the BVH builder.

Software
: The software-level design would consist of a “bare metal” application which would be capable of basic interaction with the custom hardware. This application could

be used to measure the performance of the interface in practical rather than theoretical terms by measuring operation timings on the software side.

Operating System

The final stage of the design is at the OS level. It was proposed that custom OS install would be created. It would feature frame buffer and multithreading support as well as custom libraries to allow for video output and texture support in a ray tracing application. This application would then delegate the construction of the BVH to the custom hardware using the the hardware and software from the previous stages.

4.2 Design Environment

Designing for the Zynq system requires the use of a number of different software packages which together form a lengthy end-to-end work-flow which will produce a functioning Zynq system.

4.2.1 Vivado

The hardware portion will be primarily designed using Xilinx Vivado software. This program can be configured with the presets of the particular board one is using, which allows it to automatically generate a large number of circuit connections that would otherwise need to be specified manually. Is also contains an extensive catalog of hardware IP which can be integrated into one's own design. Many of these IP blocks are targeted at signal processing applications such as the Fast Fourier Transform block.

Vivado also contains an extensive set of analysis tools. Hardware expressed using IP blocks or HDL can be simulated and its behavior analysed on the signal level. This can be vital in applications where a single clock cycle delay in a particular operation could cause the system to fail. Additionally Vivado allows for design validation which will quickly pick up any errors or critical warnings in the design before it is processed. This is especially useful as the full bitstream generation flow can take a very long time, over an hour depending on the extent of the design. Vivado also supports on-chip debugging which allows for a live analysis of the signal values while the system is powered on and operating [51].

The most important functionality that Vivado provides is the ability to take a hardware bitstream that can be loaded onto the board. This is a multistage process that begins with a project specified at the register transfer level (RTL) and ends with a bitstream that describes the FPGA custom hardware. The first of these stages is called synthesis. Synthesis is essentially the process of taking the RTL description which may be a combination of IP blocks and HDL and convert this into a logic gate level representation [52]. Once the design has been synthesised and a gate level representation has been generated, it must be converted so as to be compatible with the FPGA. This stage which seeks to configure the FPGA resources to represent the gate level logic expression is called implementation [53]. With the design implemented it must be converted to a format that can be loaded onto the board, this process is called bitstream generation.

4.2.2 Xilinx SDK

After the bitstream has been generated in Vivado it can be exported to the Xilinx SDK. SDK is an integrated development environment (IDE) based on Eclipse which has been heavily adapted to support the particular needs of the Zynq system. The hardware description that has been exported from Vivado can be loaded onto the chip through SDK using the "Program FPGA" command. This hardware description can also be used by the SDK to generate a Board Support Package (BSP). The BSP contains drivers automatically generated for the hardware that has been implemented in the design. This allows the user to communicate with the board in C/C++ code using driver calls.

The main function of the SDK is software development. When creating a new application one can select between a bare metal, "standalone" application or a Linux application. The bare metal application will run directly on the hardware with no need for an OS layer. This gives excellent performance but is limited in what it can achieve. The Linux application option will create an application that when it builds will be cross-compiled using the GNU Linux compiler appropriate for the ARM processor on the chip in use. The cross compiled application can then be transferred via storage media to the board and executed from the OS running on the board.

Another important ability of the SDK is its debugging capabilities. Applications can be launched remotely on the hardware and a connection can be maintained that will allow SDK to manage the program flow control. Breakpoints and operation stepping function just like they would in a conventional IDE but the code is being executed remotely.

4.2.3 PetaLinux SDK

Xilinx provide a software package called the PetaLinux SDK which can build, develop, test and deploy an embedded Linux solution onto Zynq devices. With embedded processor systems like this, a lightweight OS is desirable and the PetaLinux SDK is configurable so that it can contain only those packages and capabilities an application requires. If a particular application does not need to output video then the PetaLinux SDK can be configured to produce a boot image that does not contain a frame buffer.

While applications can be developed in the SDK, cross compiled and transferred to a Linux install it is also possible to create applications within the PetaLinux SDK itself. Applications made in this way will be compiled when the boot image is being built. While this approach allows for applications to be part of the main Linux file system it does lack the debugging and syntax highlighting capabilities of the SDK. In similar fashion custom libraries can be added to the install which can greatly expand the capabilities of the OS.

4.3 Implementation

Implementation of the shared memory architecture followed the same three component structure outlined in the design phase. It was decided that the subtree builder, rather than the complete hardware BVH builder, would be the target for the interface design. This decision was made for a number of reasons. Results produced by Doyle in his doctoral thesis which presents the BVH builder indicate that the majority of performance gain arose from the parallel subtree builders and hypothesised that a processor system computing the upper nodes would be no less efficient and could prove more flexible [4]. Additionally the subtree builder has a slightly more simple interface making it easier to implement as a proof of concept.

4.3.1 Hardware

The hardware shared memory interface must be designed such as to communicate with the subtree builder. As described previously the most important important aspects of the subtree builder's interface are the 216 bit data in and data out channels, three 24 bit address channels and the single bit go channel. It was envisioned that the main data, the scene data in and the BVH data out would be managed by the shared memory architecture. Another, more direct solution would be better suited to manage the address channels and control signals.

AXI DMA Engine

The solution to achieving the desired shared memory architecture was found in the AXI DMA Controller. This IP core enables high bandwidth direct memory access between main memory and peripherals. The DMA controller connects to the target peripherals with an AXI4-Stream channel supporting bandwidths of 8, 16, 32, 64, 128, 256, 215 and 1024 bits [54]. Conceptually, the module sits between the DDR3 memory controller and the peripheral. The processor does not connect directly to the peripheral but rather will negotiate the data transfer between the DMA controller and the DDR3 controller and then return to it's normal operation. In this way the processor does not even have to wait for the transfer to the peripheral to be completed before it can return to other operations.

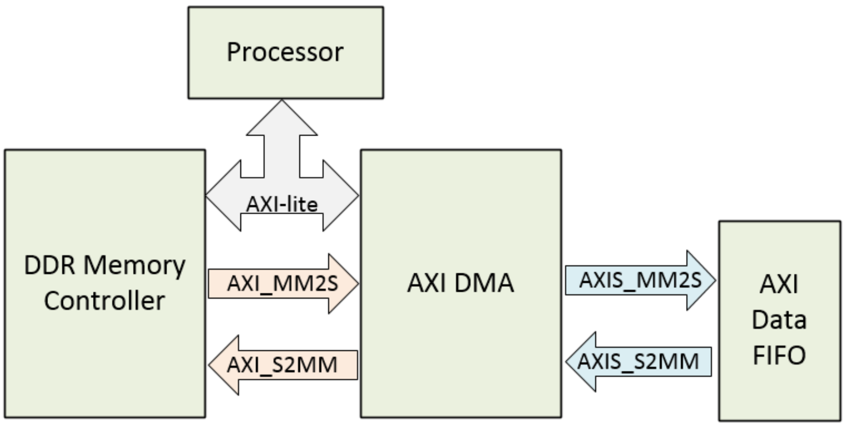


Figure 4.1: Dataflow Block Diagram [7].

One limitation of the DMA engine which affects the system design is its capacity to handle unaligned transfers. If there are two AXI4-Stream channels connecting the DMA engine and the custom peripheral with 64 bits or less of bandwidth, then the system can support unaligned transfers. That is, transfers of data not aligned on a 64, 32, 16, etc. bit boundary. However if the channel is increased above a 64 bit width then this ability is lost, meaning that addresses given to the DMA engine must be correctly aligned along the appropriate boundary.

Peripheral Interface

We can consider then that the peripheral interface will contain two AXI4-Stream interfaces which will connected to the data in and data out channels on the subtree builder. However, those channels on the subtree builder are a peculiar 216 bits wide, a size not supported by the DMA engine. This means that there are two main options for designing the interface between the subtree builder and the AXI4-Stream port.

- Proceed with the next closest channel width available, 256bit. This will result in 40 bits of additional wasted memory for every scene primitive, there will also be further data waste at the start and end of the memory block in order to ensure than the memory is aligned along 256bit boundaries.
- Utilise a 32bit channel and buffer the input up to 224 bits before releasing to the BVH builder. This will result in only 8 bits of waste memory per primitive but will be forced to take multiple clock cycles to deliver each complete set of primitive data.

Regardless of the approach taken there is an important factor to consider, namely the AXI4-Stream Protocol. The stream interface isn't simply a single channel that continuously outputs data, but rather it contains a number of important control signals which allow for communication between the master and slave devices. Consider the connection through which data is delivered to the peripheral. In this case the DMA controller is acting as the master device and the subtree builder as the slave device. The reverse is true for the other stream connection that writes the output of the peripheral back to memory. On the slave side there is the TREADY signal which informs the master device that the slave is ready to accept data. Similarly on the master side there is the TVALID signal which indicates the master device has valid data ready to be written. The AXI4-Stream transfer will only take place if both TVALID and TREADY are being asserted [55].

It is also of importance that rate of data flow be managed between the two devices. If the subtree builder reads data at a rate slower than the DMA engine can deliver it then it would be desirable to maintain a memory buffer next to the subtree builder to allow the DMA transfer to complete as swiftly as possible. To achieve this a custom FIFO unit is utilised which will moderate the dataflow rate and also manage the control signals. FIFOs essentially function as a data queue, where the first value written in will be the first value read out, they contain logic that will signal to the connected devices when they are full or empty. A separate FIFO will be maintained either side of the BVH builder with slightly different configurations for communicating with the the AXI4-Stream master and slave ports.

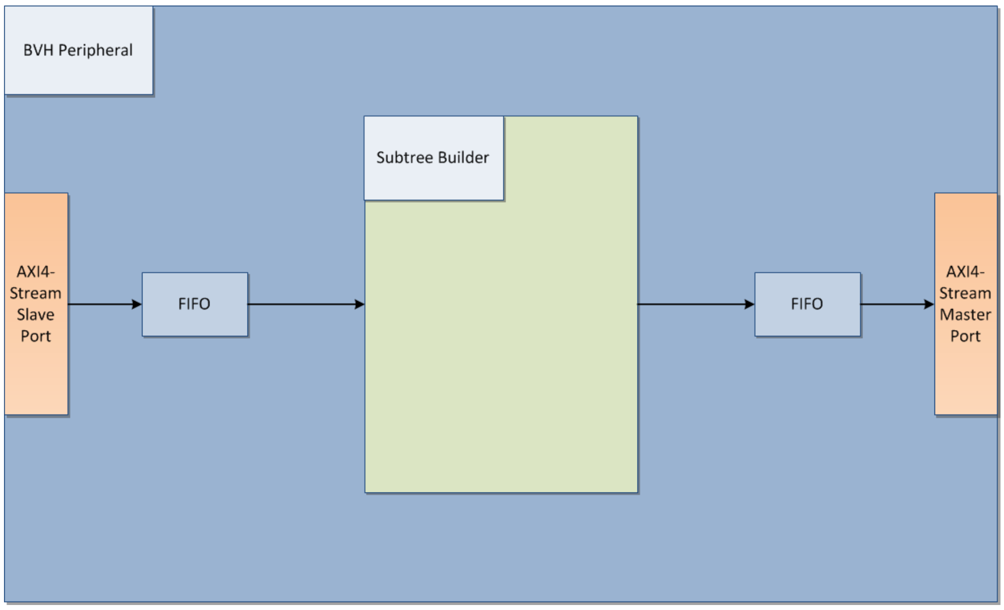


Figure 4.2: Dataflow block diagram with FIFO control.

In addition to the FIFO, logic if the approach is taken to use the 32bit data channel width in order to minimise data waste and enable unaligned transfers, further logic will be needed to allow for the data to be concatenated together into the required 216bit data in signal and similarly on the data out channel. This would be achieved using a small set of seven 32bit registers and a counter. The counter would direct each data read from the FIFO to the appropriate register then increment to the next one. When it reaches the seventh register it would enable a simultaneous write into the subtree builder from all of the registers with the remaining eight bits simply connected to ground, the counter would then reset to zero.

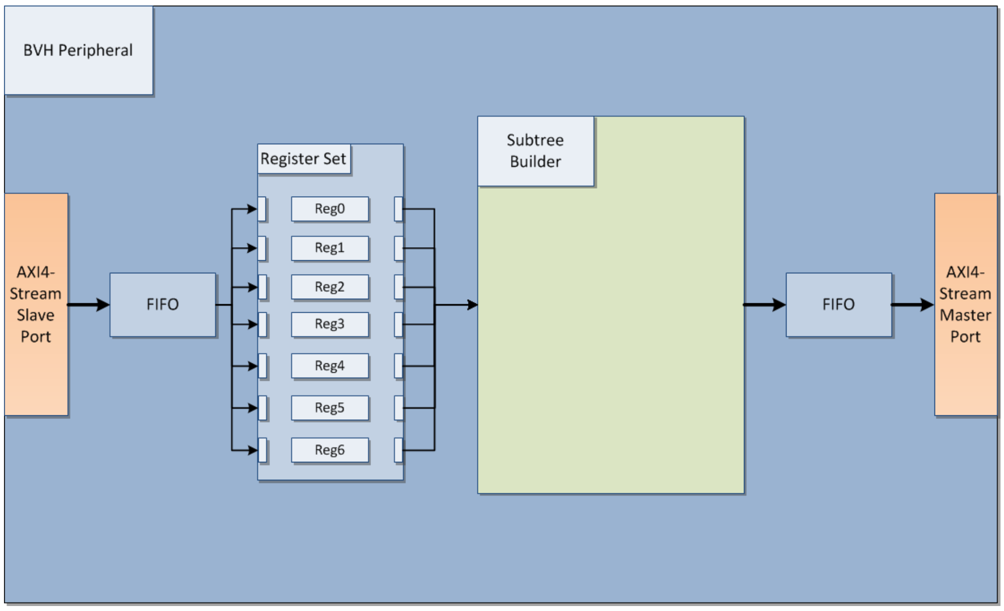


Figure 4.3: Dataflow block diagram with FIFO control and register set.

However data delivery is not the only aspect which must be considered. It is also necessary for the processor to communicate directly with the peripheral to allow for the delivery of constant and control signals to the BVH builder. For the subtree builder interface we need three 24-bit address values, two 16-bit values relating to the primitive count, two 8-bit values relating to the tree depth and a single bit GO signal. The solution to achieve this is through the AXI4-Lite interface. This will allow us to establish a connection directly between one of the processors AXI master ports and the slave port that we create on the peripheral.

Upon creation of the AXI4-Lite slave port in Vivado the user will be prompted to select the number of registers associated with the port. It is into these registers that the control signals will be written and the register output can then be connected to the appropriate port on the subtree builder unit. The total size of control signals is 121 bits which means that four 32 bit registers will be required. The remaining seven output bits can simply be connected to ground.

The processor is now trying to connect to both the DMA controller in order to negotiate transfers and directly to the peripheral via the AXI4-Lite interface. Because of this we can simply connect one of the processors general purpose AXI master ports to an AXI interconnect which will then connect to both the slave ports on the DMA controller and on the peripheral. The final connection which must be made is to connect the two interrupt ports on the DMA controller to a concatenation unit which is, in turn, connected to the interrupt input port on the processing system labelled IRQ_F2P. The complete hardware design is illustrated in Figure 4.4.

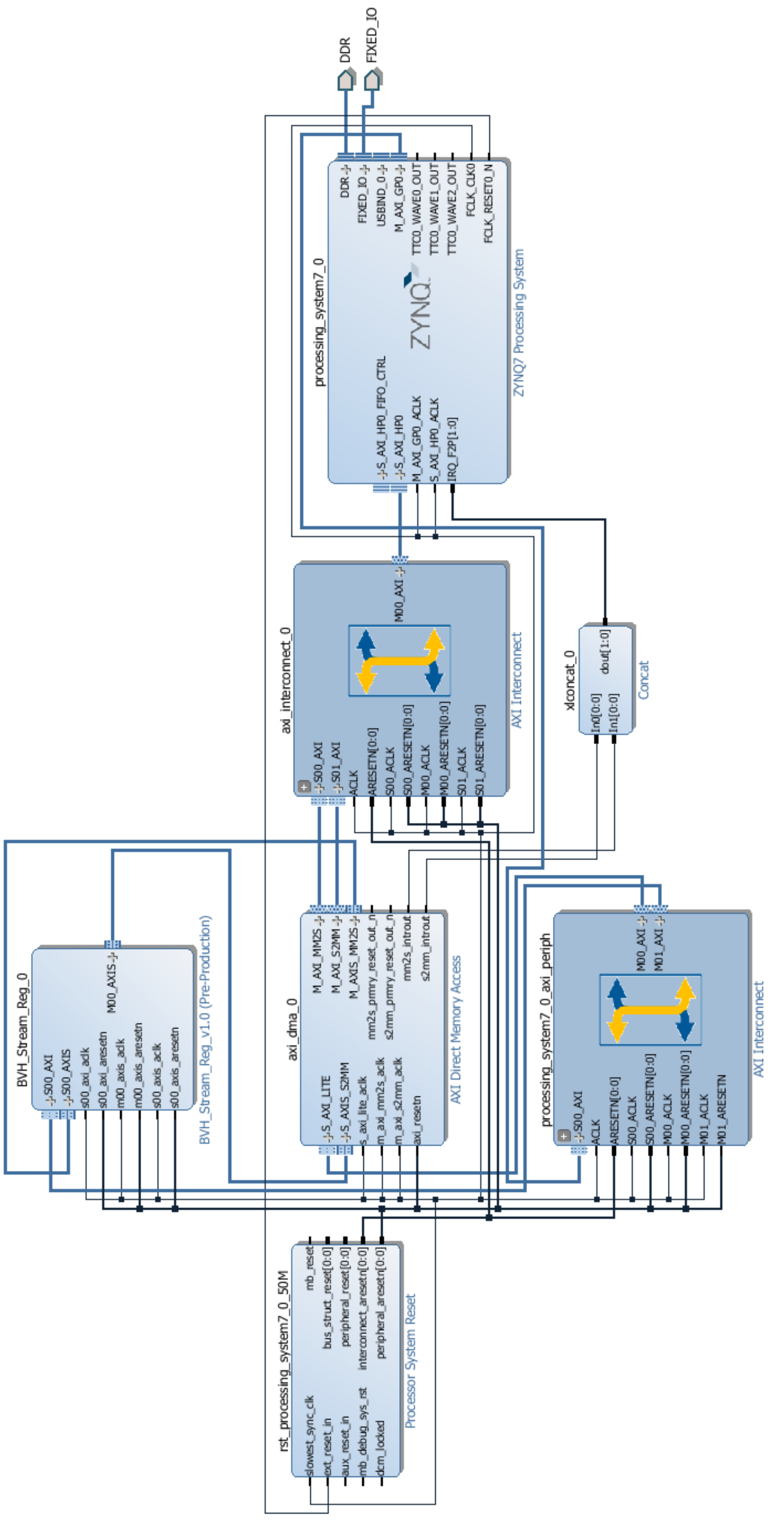


Figure 4.4: Complete Top Level System Diagram

4.3.2 Software

After the hardware design has been synthesised, implemented and converted into a bitstream, it can be used to generate a Board Support Package (BSP). This BSP will consist of a number of headers and libraries for controlling the hardware resources described in the exported hardware platform. The processor however still needs to know where exactly it's supposed to send data to, when communicating with the hardware. Because the whole system is connected by AXI interfaces when the hardware platform is generated the custom peripherals become memory mapped within the address space reserved for custom logic. The required addresses are found within the BSP.

Start Address	**Description**
0x0000_0000	External DDR RAM
0x4000_0000	Custom Peripherals
0xE000_0000	Fixed I/O Peripherals
0xF800_0000	Fixed Internal Peripherals
0xFC00_0000	Flash Memory
0xFC00_0000	On-Chip Memory

Table 4.1: Zynq Memory Map [1].

A software application communicating with the peripheral begins by initialising the driver for the DMA controller. It also initialises the interrupts for the DMA. This involves binding an appropriate interrupt service routine to be called when a particular interrupt is called. In the case of a full ray tracing application with BVH acceleration, these interrupt service routines might be used to dictate the operation of the device. The routine called when the data transfer to the device has completed for instance, might be used to write the GO signal into the appropriate register to start the BVH calculation. The routine that is triggered at the end of the return transfer could set a flag variable which informs the processor that the BVH has been constructed and is ready for use.

Making the requisite calls to begin the data transfer processes uses the ***AxiDmaSimpleTransfer***. This function, which can be found in the libraries of the BSP, takes a number of parameters:

- A reference to the DMA object which was used to initialise the hardware.

- The address in memory at which the transfer is to begin. In the case of the BVH builder this would is address of the array containing all of the scene primitive data.
- A length parameter indicating the amount of data to be transferred.
- A parameter indicating the direction of the transfer; ***XAXIDMA-DEVICE-TO-DMA*** will start the transfer to the peripheral and ***XAXIDMA-DMA-TO-DEVICE*** will start the transfer from the peripheral back to memory.

Similarly the call to write data into the control registers of the device can be found in the BSP. The function call is of the form Peripheral_Name_mWriteReg and takes the following parameters:

- A reference to the memory mapped address of the start of the peripherals register set. This reference will be found in the BSP.
- An offset to the base address for the particular register being written to. This is also found in the BSP files.
- The 32bit data value to be written to the register.

4.3.3 Operating System

Implementation of the operating system layer in the PetaLinux SDK requires a number files which must be generated either in Vivado or the SDK. The first is the hardware bitstream which is generated and exported to the SDK. The board support package which had previously been generated for use in software design is also needed as it contains vital headers and libraries for communicating with the peripherals implemented in the target design. The last thing needed to generate a boot image is a First Stage Boot Loader (FSBL). The FSBL can be created in SDK and takes the form of an ELF file. It is responsible for the early stages of the boot process including determining which pieces of hardware are to be used and need to be powered on [56].

With these files created and PetaLinux installed (Linux only) a project can be created using the BSP as a source. From here the desired kernel version can be selected, in this case linux-xlnx. With the kernel configured, a boot image can be generated using the petalinux-package terminal command. This command takes as parameters the FSBL and FPGA bitstream. This will generate the two files necessary to allow the system to boot; the boot binary, BOOT.bin and the boot image image.ub.

Before the PetaLinux OS has been built it is possible to add custom code and libraries. In this custom install a ray tracing application is created. This application is cross compiled for the Zynq when PetaLinux builds the boot image. Similarly this application requires the SDL libraries to function [4]. As there are no officially supported SDL implementations for Zynq it is necessary to cross compile the libraries from source and copy them across at build time. Unfortunately the lack of windowing system support in the OS means that the ray tracer fails when it attempts to start rendering an image. Work on this aspect of the project is ongoing.

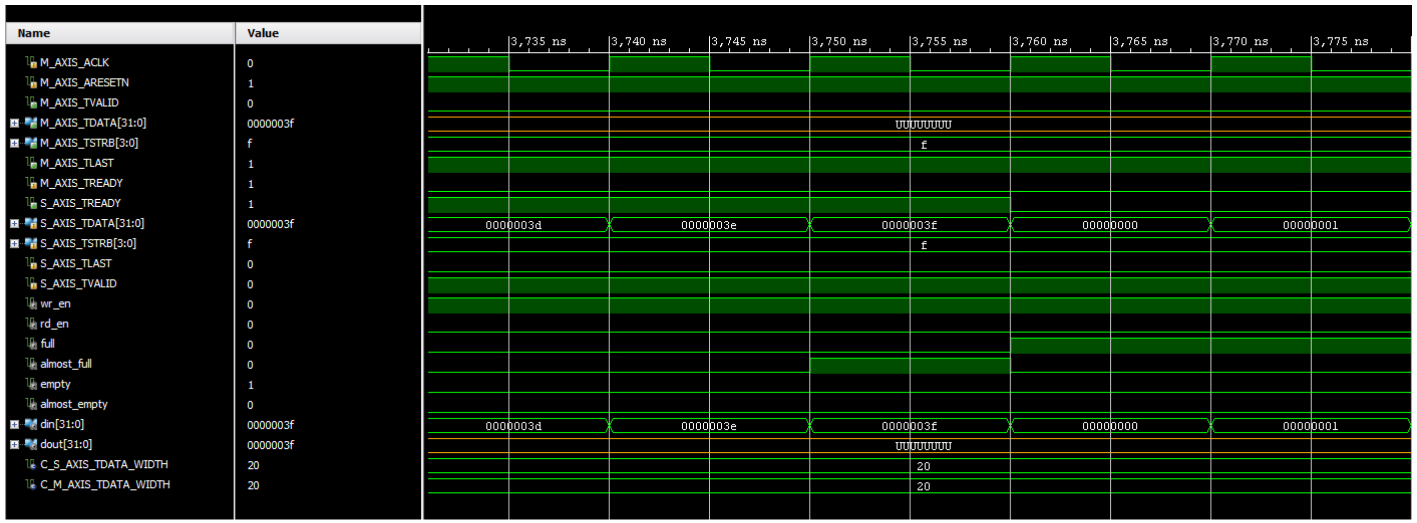
Chapter 5

Results & Evaluation

This section will present an evaluation of the outlined shared memory architecture. This analysis will cover the simulation results for the custom peripheral and how its interface with the AXI4-Stream protocol. It will also discuss factors such as the effect of the system throughput on the BVH builder and also the resource utilisation of the interface.

5.1 Interface Simulation

Vivado allows the user to simulate hardware for the purposes of testing. There are a number of way to apporach this, for instance a VHDL testbench. However, this system was evaluated using Tool Command Language (TCL) code. TCL commands drive the simulation behaviour setting signals high or low, simulating clock signals and instructing the system to run for set periods A script was written to simulate the behaviour of the DMA controller when performing a transfer to and from the custom module. The signal characteristics associated with these processes were obtained from the DMA product guide [54].



Name	Value
M_AXIS_ACLK	0
M_AXIS_ARESETN	1
M_AXIS_TVALID	0
M_AXIS_TDATA[31:0]	0000003f
M_AXIS_TSTRB[3:0]	f
M_AXIS_TLAST	1
M_AXIS_TREADY	1
S_AXIS_TREADY	1
S_AXIS_TDATA[31:0]	0000003f
S_AXIS_TSTRB[3:0]	f
S_AXIS_TLAST	0
S_AXIS_TVALID	0
wr_en	0
rd_en	0
full	0
almost_full	0
empty	1
almost_empty	0
din[31:0]	0000003f
dout[31:0]	0000003f
C_S_AXIS_TDATA_WIDTH	20
C_M_AXIS_TDATA_WIDTH	20

Figure 5.1: Simulation of DMA memory mapped to stream transfer for the BVH builder interface.

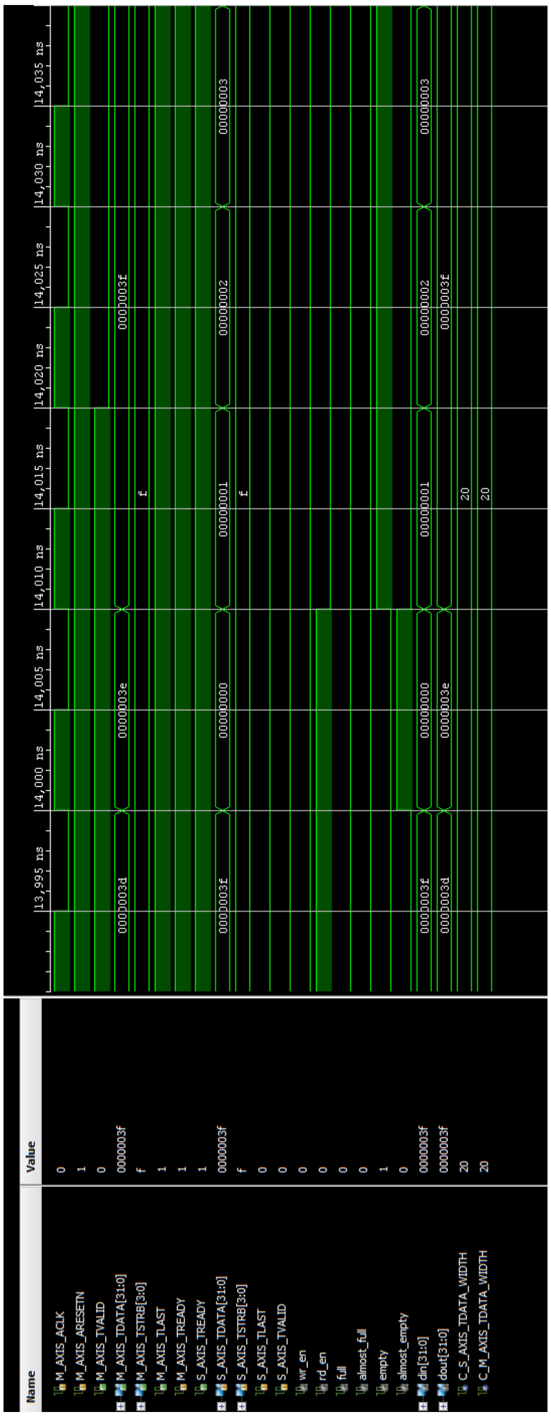


Figure 5.2: Simulation of DMA stream to memory mapped transfer for the BVH builder interface.

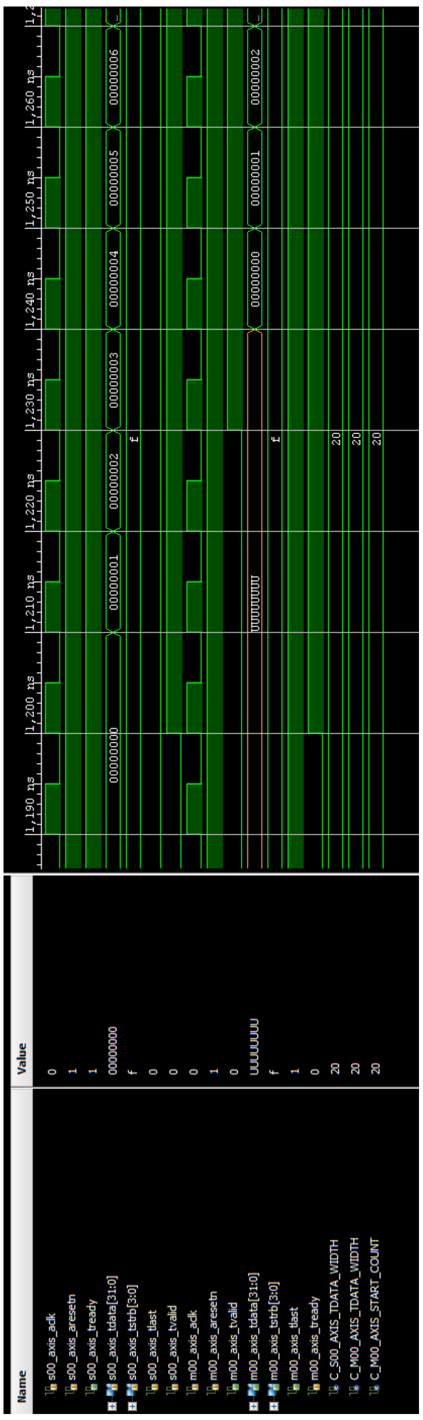


Figure 5.3: Simulation of simultaneous reading and writing.

In Figure 5.1 we see a simulation of a DMA memory mapped to stream (MM2S) transfer. The S_AXIS_TDATA channel, connected to the FIFO's *din* channel, is delivering a stream of data that changes value every cycle for clarity. The DMA engine would constantly be delivering valid data so the S_AXIS_TVALID signal will be asserted for the duration of the transfer. The moment of particular interest that is captured in Figure 5.1 relates to the FIFO reaching it's full state and how it influences the transfer. It can be seen from examination of the signals that on the same rising edge that the $full$ signal gets set high indicating that there is no room left in the FIFO the S_AXIS_TREADY signal is set low which indicates to the DMA controller that the device is no longer able to accept data thus halting the transfer.

Similarly in Figure 5.2 we see a simulation of a DMA stream to memory mapped (S2MM) transfer. In this case the *dout* channel of the FIFO is connected to the M_AXIS_TDATA master AXI port. As this side of the design is writing to the AXI4-Stream port rather than reading from it, it will need to control the valid signal rather than the read signal. This behaviour can be seen in in Figure 5.2 where we see the last available valid data being written from the FIFO to the AXI port. Note how then the *empty* signal goes high to indicate that the last piece of data has been read from the FIFO, the M_AXI_TVALID signal remains high for 1 additional clock cycle to allow it to read the value on the M_AXIS_TDATA channel before going low. If the FIFO were to be refilled with data the M_AXI_TVALID would return to a high state until all of the data had been read again.

5.2 Throughput Analysis

The two main properties of the system interface, which must be considered when analysing how it would affect the performance of the subtree builder hardware are; bandwidth and clock frequency. The main performance metric of the subtree builder is the total time to build (TTTB) which is the sum of the fill time (the time required to deliver the data into the builder) and execution time (the time required to compute the BVH). The bandwidth of the interface affects the fill time and the clock frequency affects both the fill time and the execution time.

A simulation of the subtree builder was carried out, on a number of standard scenes using the same method as Doyle et al. [4]. Only portions of the standard scenes are taken as the capacity of the subtree builder is insufficient for the whole scene. This simulation calculates the number of clock cycles it would take for the subtree builder to construct the BVH on a given scene. Using this data and figures for the clock frequency and bandwidth we can determine the TTTB

Scene	**No. Primitives**	**Leaf Size**	**Cycles to Complete**
toasters_p0_2.obj	4838	4	75199
marbles_p1_2.obj	4408	4	58334
cloth_subtree0.obj	7488	4	87983
armadillo_subtree30.obj	6407	4	83196
dragon_subtree50.obj	7093	4	96178

Table 5.1: Results of subtree builder simulation.

The data in Table 5.1 is generated from a simulation of the subtree builder using a particular configuration. The simulated builder had a width value of 4. Width refers to the number of partitioning units that are available in the system. It also has 8 available thread contexts which are used to hide the latency of certain operation and ensure optimal pipeline utilisation.

From these results the fill time can be calculated by taking the number of primitives, multiplying it by the number of bits per primitive of the interface (256 or 224), dividing by the bandwidth and then multiplying by the period of the clock signal. The execution time can be found by simply multiplying the number of cycles to complete. We can sum them together to determine the TTTB. Applying this to the two system configurations at 250MHz gives the following results.

Scene	**Fill Time(ns)**	**Ex. Time(ns)**	**TTTB(ns)**	**Pad(bits)**
toasters_p0_2.obj	19352	300796	320148	193,520
marbles_p1_2.obj	17632	233336	250968	176320
cloth_subtree0.obj	29952	351932	381884	299520
armadillo_subtree30.obj	25628	332784	358412	256280
dragon_subtree50.obj	28372	384712	413084	283720

Table 5.2: Performance results for the 256bit system variant @250MHz.

The results in Table 5.2 and Table 5.3 reveal a trade off between performance and data efficiency. While the limitations of working with the DMA engine mean that both approaches have significant padding, the 256bit system wastes five times as much data

Scene	**Fill Time(ns)**	**Ex. Time(ns)**	**TTTB(ns)**	**Pad(bits)**
toasters_p0_2.obj	135464	300796	436260	38704
marbles_p1_2.obj	123424	233336	356760	35264
cloth_subtree0.obj	209664	351932	561596	59904
armadillo_subtree30.obj	179396	332784	512180	51256
dragon_subtree50.obj	198604	384712	583316	56744

Table 5.3: Performance results for the 32bit system variant @250MHz.

per frame as the 32bit system. However, the price for this more efficient data usage is a significant hit in performance.

On average the 32-bit system took 145123.2ns longer in terms of TTTB. That means that the 32-bit system takes on average 42.05% longer than the 256-bit system, with fill time comprising 34.55% of TTTB, as opposed to only 7% for the 256-bit approach. Considering the scale of the performance difference between the 256-bit system is clearly the preferable option despite is inferior efficiency in terms of data. It would be possible to reduce the large amount of padding in the 26-bit system by making the spare 40 bits for each primitive the first 40 bits of the next primitive data, however this would add significant complexity to the hardware design and may cause the design to meet timing. A thorough evaluation of such a system would be worthy of investigation but was beyond the scope of this project.

5.3 Resource Utilisation

An important factor to consider in relation to these interfaces is the amount of resources that are required to implement them. In the case of an ASIC this might be measured in terms of number of gates or consumed chip area. However as this implementation is for an FPGA it is convenient to discuss resource utilisation in terms of the amount of available resources on the FPGA, such as logic cells and memory blocks, that must be used to implement the interface.

5.3.1 256bit Data Channel

The following results present the FPGA Logic and Memory resources consumed by the 256bit channel design. While it lacks the register logic of the 32 bit variant, its increased bandwidth means that the memory allocated for the FIFO will be larger. The figures are reported from Vivado after the design has been synthesised.

Site Type	**Used**	**Loced**	**Available**	**Util%**
Slice LUTs	6293	0	53200	11.82
LUT as Logic	6078	0	53200	11.42
LUT as Memory	215	0	17400	1.23
LUT as Distributed RAM	106	0		
LUT as Shift Register	109	0		
Slice Registers	8103	0	106400	7.61
Register as Flip Flop	8103	0	106400	7.61
Register as Latch	0	0	106400	0.00

Table 5.4: Logic resource utilisation for 256bit data channel configuration @ 50MHz.

Site Type	**Used**	**Loced**	**Available**	**Util%**
Block RAM Tile	13	0	140	9.28
RAMB36/FIFIO	12	0	140	8.57
RAMB36E1 only	12			
RAMB18	2	0	280	0.71
RAMB18E1 only	2			

Table 5.5: Memory resource utilisation for 256bit data channel configuration @ 50MHz.

From the results in Table 5.4 and Table 5.5 we can see that this interface consumes a non negligible amount of system resources. However these results consider the board to be operating at the default 50MHz FPGA clock. If the frequency is increased to the maximum supported by this clocking structure Vivado will seek to optimise the design during synthesis, in order to meet the imposed timing constraints. This may involve adding additional registers or other necessary logic in order to shorten certain pipeline stages. The result is a system capable of operating at higher frequencies but consuming a greater amount of FPGA resources. No change to the memory requirements of the FPGA is caused by the frequency increase.

5.3.2 32bit Data Channel

The tables below show the level of resource utilisation of the 32 bit data channel interface solution at 50MHz and 250MHz. This implementation consumes significantly

Site Type	Used	Loced	Available	Util%
Slice LUTs	6927	0	53200	13.02
LUT as Logic	6712	0	53200	12.61
LUT as Memory	215	0	17400	1.23
LUT as Distributed RAM	106	0		
LUT as Shift Register	109	0		
Slice Registers	8103	0	106400	7.61
Register as Flip Flop	8103	0	106400	7.61
Register as Latch	0	0	106400	0.00

Table 5.6: Logic resource utilisation for 256bit data channel configuration @ 250MHz.

fewer system resources.

Site Type	Used	Loced	Available	Util%
Slice LUTs	3387	0	53200	6.36
LUT as Logic	3242	0	53200	6.09
LUT as Memory	145	0	17400	0.83
LUT as Distributed RAM	18	0		
LUT as Shift Register	127	0		
Slice Registers	4174	0	106400	3.92
Register as Flip Flop	4174	0	106400	3.92
Register as Latch	0	0	106400	0.00

Table 5.7: Logic resource utilisation for 32bit data channel configuration @ 50MHz.

These results considered in light of the performance analysis suggest that increased performance will also require a significantly greater amount of system resources. It should be considered that the ZC702 has a lower grade Zynq chip containing fewer FPGA resources than more expensive boards. As such the percentage of resources consumed would be less for the same hardware. Another point of note is that the FIFOs utilised are very large each having a depth of 256. Simulations seen in Figure 5.3 suggest that such a large FIFO depth is unnecessary. However the utilisation results in Table 5.10 show that a significant reduction in FIFO depth from 256 to 20 had an almost negligible effect on the resources consumed. This indicates that the DMA engine is responsible for the majority of the consumed resources.

Site Type	**Used**	**Loced**	**Available**	**Util%**
Block RAM Tile	3.5	0	140	2.50
RAMB36/FIFIO	2	0	140	1.42
RAMB36E1 only	2			
RAMB18	3	0	280	1.07
RAMB18E1 only	3			

Table 5.8: Memory resource utilisation for 32bit data channel configuration @ 50MHz.

Site Type	**Used**	**Loced**	**Available**	**Util%**
Slice LUTs	3585	0	53200	6.73
LUT as Logic	3440	0	53200	6.46
LUT as Memory	145	0	17400	0.83
LUT as Distributed RAM	18	0		
LUT as Shift Register	127	0		
Slice Registers	4174	0	106400	3.92
Register as Flip Flop	4174	0	106400	3.92
Register as Latch	0	0	106400	0.00

Table 5.9: Logic resource utilisation for 32bit data channel configuration @ 250MHz.

5.4 Overall Evaluation

The stated goal at the beginning of this investigation was to create a complete hardware/software hybrid ray tracing system running on the ZC702. While much of each of the stages has been completed they do not fully work together. The hardware interface has been developed, tested and evaluated but has yet to be successfully integrated with the subtree builder hardware. In spite of this the implementation is at a level where the results presented can be considered a valid representation of a final, fully functioning system.

Site Type	**Used**	**Loced**	**Available**	**Util%**
Slice LUTs	3500	0	53200	6.57
LUT as Logic	3355	0	53200	6.30
LUT as Memory	145	0	17400	0.83
LUT as Distributed RAM	18	0		
LUT as Shift Register	127	0		
Slice Registers	4165	0	106400	3.91
Register as Flip Flop	4165	0	106400	3.91
Register as Latch	0	0	106400	0.00

Table 5.10: Logic resource utilisation for 32bit data channel configuration @ 250MHz and FIFO depth of 20.

Chapter 6

Discussion

6.1 Conclusion

This thesis proposed a design for a shared memory architecture to enable efficient communication between a processor system and a BVH accelerator. The system was designed and implemented for the Xilinx ZC702 evaluation board.

The shared memory architecture, composed of Vivado IP blocks and custom VHDL modules, was evaluated by variety of metrics. Two different configurations of the design were examined with each producing superior performance than the other in certain areas. Ultimately it was determined that the proposed architecture is feasible for implementation on a device featuring an integrated processor and programmable logic subsystem.

Ray tracing will continue to be a goal towards which the computer graphics community strives. The dark silicon problem and the rise of mobile computing may drive chip manufacturers to provide power efficient, fixed function hardware alongside the traditional CPU and GPU oriented architectures. This investigation has demonstrated that, fixed function hardware sharing memory with a processor, can provide efficient delivery of large quantities of data without occupying processor time. This research could be a starting off point, a platform on which further research into heterogeneous graphics architectures could build.

6.2 Future Work

Following on from this project the next obvious goal would to complete the system initially planned and allow for a software ray tracing system to delegate the construction of a BVH for small scenes to the programmable hardware. The existing system is very close to this point and would require only a few notable issues to be fixed. Moving beyond that then, the next step would be to augment the interface to support the complete BVH builder hardware. This would enable support for significantly large scenes.

Ultimately no amount of hardware acceleration will make the ZC702 a suitable device for ray tracing. It's unique hardware capabilities made it ideal for testing this proof of concept memory architecture. The bulk of research opportunities arising from this investigation lie with the BVH builder hardware. Recent research by Fowler et al. [57] considers the dual role of the BVH in ray tracing and physics simulations. Their analysis shows that the optimal number primitives per leaf node of the BVH changes is different for the two applications. As such they propose an adaptive BVH which inner nodes flagged as leaf nodes for ray tracing traversal while the final leaf nodes consisting of only one primitive would function normally for physics simulation.

From a hardware perspective the current iteration of the BVH builder would be unable to achieve this structure. This is due to the binned SAH algorithm utilised in the design which is unable to guarantee a primitive count of one per leaf node. Further research in this area might consider an addition of a third conceptual layer to the hardware. Much like the upper builder delegates nodes of an appropriate size to the subtree builder; the subtree builder could delegate nodes of a certain size to a sweep builder. This unit would implement the more complex and computationally expensive SAH sweep algorithm. This approach would allow for hardware generation of an adaptive BVH suitable for both physics simulation and ray tracing.

Bibliography

[1] Xilinx, "Zynq programmable logic highlights," 2012.

[2] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-Time Rendering.* CRC Press, 2008.

[3] H. Nguyen, *Gpu Gems 3.* Addison-Wesley Professional, first ed., 2007.

[4] M. Doyle, "Hardware support for power and area efficient construction of high-quality bounding volume hierarchies," 2014.

[5] Xilinx, *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide*, v1.3 ed., June 2014.

[6] Xilinx, "Zynq processing system highlights," 2012.

[7] J. Johnson, "Using the axi dma in vivado," 2014.

[8] J. Schmittler, I. Wald, and P. Slusallek, "Saarcor: A hardware architecture for ray tracing," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, (Aire-la-Ville, Switzerland, Switzerland), pp. 27–36, Eurographics Association, 2002.

[9] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "An energy and bandwidth efficient ray tracing architecture," in *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, (New York, NY, USA), pp. 121–128, ACM, 2013.

[10] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "Sgrt: A mobile gpu architecture for real-time ray tracing," in

Proceedings of the 5th High-Performance Graphics Conference, HPG '13, (New York, NY, USA), pp. 109–119, ACM, 2013.

[11] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), (New York, NY, USA), pp. 37–45, ACM, 1968.

[12] P. H. Christensen, G. Harker, J. Shade, B. Schubert, and D. Batali, "Multiresolution radiosity caching for global illumination in movies," in *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, (New York, NY, USA), pp. 47:1–47:1, ACM, 2012.

[13] C. Hery and R. Villemin, "Physically based lighting at pixar," July 2013.

[14] S. Laine and T. Karras, "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, (New York, NY, USA), pp. 79–88, ACM, 2011.

[15] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T and i engine: Traversal and intersection engine for hardware accelerated ray tracing," in *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, (New York, NY, USA), pp. 160:1–160:10, ACM, 2011.

[16] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[17] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, pp. 256–268, Oct 1974.

[18] Qualcomm, "Qualcomm snapdragon 810 processor product brief," 2014.

[19] NVIDIA, "Bringing high-end graphics to handheld devices," 2011.

[20] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, (New York, NY, USA), pp. 143–150, ACM, 1986.

[21] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, pp. 343–349, June 1980.

[22] A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated ray-tracing system," *Computer Graphics and Applications, IEEE*, vol. 6, pp. 16–26, April 1986.

[23] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, (New York, NY, USA), pp. 485–493, ACM, 2006.

[24] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," *Commun. ACM*, vol. 19, pp. 547–554, Oct. 1976.

[25] C. Wächter and A. Keller, "Instant ray tracing: The bounding interval hierarchy," in *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR'06, (Aire-la-Ville, Switzerland, Switzerland), pp. 139–149, Eurographics Association, 2006.

[26] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, Sept. 1975.

[27] A. Williams, S. Barrus, R. K. Morley, and P. Shirley, "An efficient and robust ray-box intersection algorithm," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.

[28] J. O'Rourke, "Finding minimal enclosing boxes," *International Journal of Computer Information Sciences*, vol. 14, no. 3, pp. 183–199, 1985.

[29] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 4, pp. 21–36, Jan 1998.

[30] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, "Fast, effective bvh updates for animated scenes," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, (New York, NY, USA), pp. 197–204, ACM, 2012.

[31] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, pp. 33–40, 2007.

[32] D. J. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *Vis. Comput.*, vol. 6, pp. 153–166, May 1990.

[33] I. Wald, "Fast construction of sah bvhs on the intel many integrated core (mic) architecture," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, pp. 47–57, Jan 2012.

[34] M. J. Doyle, C. Fowler, and M. Manzke, "A hardware unit for fast sah-optimised bvh construction," *ACM Trans. Graph.*, vol. 32, pp. 139:1–139:10, July 2013.

[35] Y. Gu, Y. He, K. Fatahalian, and G. E. Blelloch, "Efficient bvh construction via approximate agglomerative clustering.," in *High Performance Graphics*, pp. 81–88, ACM, 2013.

[36] B. Walter, K. Bala, M. Kulkarni, and K. Pingali, "Fast agglomerative clustering for rendering," in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pp. 81–86, 2008.

[37] H. Kobayashi, K. Suzuki, K. Sano, Y. Kaeriyama, Y. Saida, N. Oba, and T. Nakamura, "3dcgiram: an intelligent memory architecture for photo-realistic image synthesis," in *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pp. 462–467, 2001.

[38] T. Todman and W. Luk, "Reconfigurable designs for ray tracing," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pp. 300–301, March 2001.

[39] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," in *Computer Graphics Forum*, pp. 153–164, 2001.

[40] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime ray tracing of dynamic scenes on an fpga chip," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, (New York, NY, USA), pp. 95–106, ACM, 2004.

[41] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes," in *Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003. IEEE Symposium on*, pp. 77–85, Oct 2003.

[42] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "Trax: A multicore hardware architecture for real-time ray tracing," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, pp. 1802–1815, Dec 2009.

[43] J. Spjut, S. Boulos, D. Kopta, E. Brunvand, and S. Kellis, "Trax: A multi-threaded architecture for real-time ray tracing," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 108–114, June 2008.

[44] J. Spjut, D. Kopta, E. Brunvand, and A. Davis, "A mobile accelerator architecture for ray tracing," *3rd Workshop on Socs, Heterogeneous Architectures and Workloads*, Feb 2013.

[45] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proceedings of the Conference on High Performance Graphics*, HPG '10, (Aire-la-Ville, Switzerland, Switzerland), pp. 113–122, Eurographics Association, 2010.

[46] M. Sanchez-Elez, H. Du, N. Tabrizi, Y. Long, N. Bagherzadeh, and M. Fernndez, "Algorithm optimizations and mapping scheme for interactive ray tracing on a reconfigurable architecture.," *Computers and Graphics*, vol. 27, no. 5, pp. 701–713, 2003.

[47] H.-Y. Kim, Y.-J. Kim, and L.-S. Kim, "Reconfigurable mobile stream processor for ray tracing," in *Custom Integrated Circuits Conference (CICC), 2010 IEEE*, pp. 1–4, Sept 2010.

[48] H.-Y. Kim, Y.-J. Kim, and L.-S. Kim, "Mrtp: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization," *Solid-State Circuits, IEEE Journal of*, vol. 47, pp. 518–535, Feb 2012.

[49] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung, "Sgrt: A scalable mobile gpu architecture based on ray tracing," in *ACM SIGGRAPH 2012 Posters*, SIGGRAPH '12, (New York, NY, USA), pp. 44:1–44:1, ACM, 2012.

[50] Xilinx, "Zynq 7000 combined product table," may 2014.

[51] Xilinx, *Vivado Design Suite User Guide, Programming and Debugging*, v2014.1 ed., May 2014.

[52] Xilinx, *Vivado Design Suite User Guide, Synthesis*, v2013.2 ed., June 2013.

[53] Xilinx, *Vivado Design Suite User Guide, Implementation*, v2013.4 ed., December 2013.

[54] Xilinx, *LogiCORE IP AXI DMA Product Guide*, 7.1 ed., April 2014.

[55] ARM, *AMBA 4 AXI4-Stream Protocol Specification*, 1.0 ed., 2010.

[56] Xilinx, *Zynq-7000 All Programmable SoC Software Developers Guide*, 9.0 ed., June 2014.

[57] C. Fowler, M. Doyle, and M. Manzke, "Adaptive bvh: An evaluation of an efficient shared data structure for interactive simulation," in *Proceedings of the Spring Conference of Computer Graphics*, SCCG '14, (New York, NY, USA), ACM, 2014.