http://researchspace.auckland.ac.nz

University of Auckland Research Repository, ResearchSpace

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
EFFICIENT RAY TRACING ON FPGAS

SAMUEl MICHAEL COLLINSON

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Systems Engineering, the University of Auckland, 2014.
ABSTRACT

Ray tracing is a computationally intensive task required by movie-makers to create the highly realistic images they require for motion pictures. GPUs currently dominate as hardware accelerators in the multi-billion dollar movie industry. However, the flexibility and power efficiency of FPGAs offer high potential to applications with many levels of parallelism, such as ray tracing.

This thesis investigates and proposes an FPGA platform to easily and efficiently explore parallelism in ray tracing on FPGAs, to decrease the productivity gap between the exponentially increasing amount of logic found on FPGAs and the number of man-hours taken to develop applications for them, fulfilling the first objective of this research. Various aspects and components of the platform that are critical for the ray tracing performance are then deeply investigated.

Two essential parts of ray tracing performance are the processes of determining if a ray intersects with a scene primitive, and determining if a ray intersects with an acceleration hierarchy bounding box. Deep, parallel floating-point pipelines for these ray-triangle and ray-box intersection tests are proposed. Both pipelines are evaluated and show potential for scalability and improved performance over other accelerated implementations.

Traversal of individual rays are independent and can be parallelised. The platform scales the number of traversal units implemented to exploit this, with a key component to their scalability the implementation of the priority queue contained within them. A novel hybrid hardware combination of a single-cycle queue and a heap is presented that combines the speed of a single-cycle queue with the scalability of a heap. An innovative method to simulate different priority queue workloads is developed to evaluate queue performance under different conditions. Optimisations are made to the heap for efficiency and scalability on the ray tracing platform.

Memory bandwidth on the proposed platform is identified as a bottleneck, with a cache being a good candidate to alleviate bandwidth requirements. Memory access patterns to the
acceleration hierarchy and scene primitives during ray tracing of four typical scenes on the platform are explored, leading to development of a novel node cache replacement policy. Evaluation shows that the replacement policy can provide a good performance increase for small cache sizes and may work well as a complimentary cache along side a direct mapped cache, and that the implementation of a node and primitive cache successfully alleviates the memory bandwidth bottleneck.

The presented floating-point intersection pipelines, heap implementation and optimisations, and ray tracing cache architecture fulfil the second objective of the research: to provide building blocks and optimisations that enable ray tracing efficiency on FPGAs.

An extensive experimental evaluation of the proposed platform on real hardware for four typical scenes is performed, including investigating the effect of priority queue type on performance and scalability, the impact of a cache on platform performance and the performance of the platform in relation to a CPU and GPU. The evaluation showed that scaling the number of traversal units with a cache provides an increase in performance over all scenes, the FPGA platform has better power efficiency than both the CPU and the GPU over all scenes and the FPGA platform has better bandwidth efficiency than both the CPU and the GPU over all scenes, with further improvements possible in all areas with the use of more current FPGA technology. This evaluation fulfils the final objective of the research: to demonstrate that FPGAs have a great potential for power and bandwidth efficiency when accelerating ray tracing.

Given the demonstrated potential of FPGAs as hardware accelerators for ray tracing, the proposed platform enables easy evaluation of newer FPGA technologies and ray tracing techniques, providing an important stepping-stone to researchers for further advances.
During the last year of my Bachelor of Computer Systems Engineering, in 2007, I decided to undertake my PhD after my Part IV project, under the supervision of Associate Professor John Morris, showed great potential in speeding up ray-triangle intersection tests by two orders of magnitude using FPGAs. At the time we believed ray-triangle intersection tests consumed the majority of ray tracing computation time [18].

My PhD started at the beginning of 2010 with Associate Professor John Morris as my supervisor. My first goal was to integrate the ray-triangle intersection pipeline created in my Part IV project with Pixie, an open-source rendering program. Progress initially was slow, as integration required me learning the structure of Pixie, Windows driver development and how to integrate the pipeline with a PCI express core. The summary of this work was presented, in part, at The International Conference on Field Programmable Logic and Applications (FPL) in 2012 [19] and is presented in this thesis in Chapter 5. Contributions presented in Chapter 5 are analysis of the standard software implementations of ray-triangle and ray-box intersection and development of deep, parallel floating-point pipelines for implementation on FPGAs.

At the beginning of 2012, John Morris resigned from The University of Auckland and Doctor Oliver Sinnen became my new PhD supervisor. As part of bringing Oliver up to speed on my research to date, he requested I take a step back and prepare a summary of ray-tracing and current hardware acceleration research. This fresh look showed that a refocus of my research was required, with acceleration of ray-triangle intersection alone not being the right approach, as acceleration hierarchies are able to replace large numbers of intersection calculations with less expensive hierarchy traversal. This led to the development of a flexible platform for ray-tracing on FPGAs that integrated the use of acceleration hierarchy and allowed easy, modular scaling and replacement of different components to investigate the performance of different ray-tracing techniques and parameters. Pixie was replaced by LuxRays as it used more current techniques for ray-tracing.
acceleration, was under current development, had more modular code allowing easier integration and a focus on hardware acceleration. The summary of the platform and integration with LuxRays was presented at The International Conference on Field-Programmable Technology (FPT) in 2013 [20] and is presented in this thesis in Chapter 4. The contributions in Chapter 4 are the flexible FPGA ray-tracing platform, providing a platform to research and explore the effectiveness of ray-tracing on FPGAs and enabling quick discovery and elimination of performance bottlenecks.

Over the 2012/2013 summer I supervised summer internship student Allan Bai together with Dr Sinnen. Together with Allan, a combined hardware implementation of a fast single-cycle priority queue and scalable heap queue was explored. An optimisation of the heapify routines for implementation on dual-port memory, that is present on FPGAs, was devised using a temporary register to allow each recursive call of the functions to complete in a single cycle. We created an evaluation method for priority queues that simulates queue workloads, modulating the number of items present on the queue to different means and standard deviations. Allan was wholly responsible for wrapping together the two priority queue implementations and writing simulation test benches for the evaluation methods we created. The summary of this work is presented in Chapter 6 and is in the submission process to the Transactions on Reconfigurable Technology and Systems (ACM). The significant contribution in Chapter 6 is the hybrid combination in hardware of fast single-cycle and scalable heap priority queues for applications requiring priority queues with the performance of a fast queue but with an unknown or highly variable working data set size. Other contributions are the devised methods for evaluation and queue workloads and an optimisation of the heap queue for workloads where a series of push operations are followed by a single pop operation, as is done in ray-tracing.

Once the platform was initially implemented and evaluated it became apparent that the bandwidth of the on-board memory was the main bottleneck to overall performance. A cache was the next logical step, but with the flexible nature of FPGAs we decided to investigate if an application specific cache would be of benefit. There are several factors that can affect the memory layout and access patterns during ray tracing, such as the maximum number of primitives that can be stored in a leaf, the width of the acceleration
hierarchy and the cost parameters for ray-triangle and ray-box intersection tests in the surface-area-heuristic. The exploration of these parameters as they are relevant to memory access patterns, the design and evaluation of node and primitives caches for FPGAs and a novel replacement policy are presented in Chapter 7 and, combined, are in the submission process to the *Journal of Parallel and Distributed Computing* (Elsevier). The contributions in Chapter 7 are memory access analysis of different types of data and different scene settings during ray tracing, design and evaluation of node and primitive caches for FPGAs and design and evaluation of a level based replacement policy for small node caches.

Throughout the previously mentioned chapters, the platform has been evaluated by running test renders of sample scenes and measuring the different areas of performance. All of these evaluations are presented in Chapter 8 along with a comparison against CPU and GPU performance. The contributions in Chapter 8 are methods for evaluating and optimising performance of the platform and performance analysis of the platform compared to a CPU for ultimate performance, energy per ray, and performance relative to the amount of memory bandwidth available to the device.
ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Dr Oliver Sinnon. You have provided exceptional advice, oversight, motivation and knowledge along the way and have superbly guided the focus of my study to significantly improve the quality of my research contributions.

Second, I would like to thank my mother, Dr Catherine Collinson. You have provided love, encouragement, financial support and excellent editorial skills, despite not understanding the technical content!

Third, I would like to thank the rest of my family and friends for their encouragement and support along the way.

Last, and most certainly not least, I would like to thank my partner and best friend, Gemma. It is hard to put into words what you mean to me, and how much I appreciate your unfailing love and support through this journey we have been on together. You have been kind, patient, forgiving, warm and caring, and it is easy for me to say that I could not have done this without you.
CONTENTS

1 INTRODUCTION 1
1.1 Ray Tracing 1
1.2 Field Programmable Gate Arrays 3
1.3 Motivation and Contributions 3
1.4 Thesis structure 5

2 RAY TRACING 9
2.1 Ray-Object Intersection 10
  2.1.1 Ray-Triangle Intersection 10
  2.1.2 Ray-Box Intersection 12
2.2 Acceleration Structures 14
  2.2.1 Space Partitioning 14
  2.2.2 Object Partitioning 17
2.3 Cornell Box 18
2.4 Rendering Software 20

3 HARDWARE ACCELERATION 23
3.1 Architectures 24
3.2 Field Programmable Gate Arrays 25
  3.2.1 Architecture 25
  3.2.2 Configuration 27
3.3 PCI Express 28
3.4 Hardware Acceleration of Ray Tracing 30
  3.4.1 Optimisations for GPUs 31
  3.4.2 SIMD/MIMD Ray Tracing 35
  3.4.3 Ray Tracing on FPGAs 37
  3.4.4 Caches for Ray Tracing 39

4 PLATFORM 41
4.1 Goals 42
4.2 Design 43
  4.2.1 Acceleration Platform 43
  4.2.2 Platform components 44
  4.2.3 Communication Controller 46
  4.2.4 Traversal Controller 50
  4.2.5 Intersection Unit 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.6 Memory Controller</td>
<td>52</td>
</tr>
<tr>
<td>4.2.7 Example for Cornell Box</td>
<td>53</td>
</tr>
<tr>
<td>4.3 Implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.3.1 Hardware components</td>
<td>55</td>
</tr>
<tr>
<td>4.3.2 Traversal unit</td>
<td>55</td>
</tr>
<tr>
<td>4.3.3 Main platform components</td>
<td>57</td>
</tr>
<tr>
<td>4.3.4 Data representation</td>
<td>58</td>
</tr>
<tr>
<td>4.4 Platform generation</td>
<td>59</td>
</tr>
<tr>
<td>4.5 Software integration</td>
<td>63</td>
</tr>
<tr>
<td>4.5.1 Operating-system integration</td>
<td>63</td>
</tr>
<tr>
<td>4.5.2 Integration with LuxRays</td>
<td>64</td>
</tr>
<tr>
<td>5 Intersection pipelines</td>
<td>67</td>
</tr>
<tr>
<td>5.1 Ray-Triangle Intersection</td>
<td>68</td>
</tr>
<tr>
<td>5.1.1 Software Implementation</td>
<td>69</td>
</tr>
<tr>
<td>5.1.2 Pipeline Design</td>
<td>71</td>
</tr>
<tr>
<td>5.1.3 Evaluation</td>
<td>79</td>
</tr>
<tr>
<td>5.2 Ray-Box Intersection</td>
<td>80</td>
</tr>
<tr>
<td>5.2.1 Software Implementation</td>
<td>80</td>
</tr>
<tr>
<td>5.2.2 Pipeline Design</td>
<td>82</td>
</tr>
<tr>
<td>5.2.3 Evaluation</td>
<td>84</td>
</tr>
<tr>
<td>6 Fast scalable queues</td>
<td>87</td>
</tr>
<tr>
<td>6.1 Single-cycle Priority Queue</td>
<td>89</td>
</tr>
<tr>
<td>6.2 Heap</td>
<td>92</td>
</tr>
<tr>
<td>6.3 Hybrid Queue</td>
<td>95</td>
</tr>
<tr>
<td>6.3.1 Implementation</td>
<td>97</td>
</tr>
<tr>
<td>6.3.2 Evaluation</td>
<td>104</td>
</tr>
<tr>
<td>6.4 Heap Optimisations</td>
<td>110</td>
</tr>
<tr>
<td>6.4.1 Optimisation for Burst-Heavy Workloads</td>
<td>110</td>
</tr>
<tr>
<td>6.4.2 Floating-Point Priorities</td>
<td>113</td>
</tr>
<tr>
<td>6.4.3 Smaller Heap Sizes</td>
<td>113</td>
</tr>
<tr>
<td>7 Efficient object caching</td>
<td>115</td>
</tr>
<tr>
<td>7.1 Memory Access Analysis</td>
<td>117</td>
</tr>
<tr>
<td>7.1.1 Scene Setup for Analysis</td>
<td>118</td>
</tr>
<tr>
<td>7.1.2 Node versus Leaf Bandwidth</td>
<td>118</td>
</tr>
<tr>
<td>7.1.3 Level Effect on Node Bandwidth</td>
<td>119</td>
</tr>
<tr>
<td>7.2 Caches for Ray Tracing</td>
<td>124</td>
</tr>
<tr>
<td>7.2.1 Design</td>
<td>124</td>
</tr>
<tr>
<td>7.2.2 Implementation</td>
<td>124</td>
</tr>
</tbody>
</table>
7.2.3 Evaluation 126
7.3 Level Replacement Policy 130
   7.3.1 Design 130
   7.3.2 Implementation 131
   7.3.3 Evaluation 132

8 Platform Evaluation 135
   8.1 Effect of queue type on performance and scalability 137
   8.2 Cache Impact on Platform Performance 139
   8.3 Performance compared to a CPU and GPU 143
      8.3.1 Raw Performance 144
      8.3.2 Energy Efficiency 146
      8.3.3 Available Bandwidth 149

9 Conclusion 153

Bibliography 159
Beginning with Pixar’s *Toy Story* in 1995, computer-based 3D animation has met with both critical and commercial success. It has grown into a multi-billion dollar industry, producing either entirely computer-generated movies, such as *Avatar*, or enhancing real, filmed scenes in movies through the digital rendering of special effects, such as the Battle of the Pelennor Fields in *The Lord of the Rings: The Return of the King* [2]. Digital rendering of special effects is now a routine requirement for movie-makers, allowing them to create scenes that would otherwise be too dangerous, too costly or physically impossible to fabricate while still giving a realistic look and feel to the film [6].

1.1 RAY TRACING

The ray tracing method of rendering can create the photo-realistic digitally rendered images required by movie makers, but requires massive computational power, even by today’s standards, as increasingly complex scenes have to be rendered frame-by-frame at very high resolutions. Ray tracing can keep a several thousand server cluster occupied for months rendering a full-length animated movie [65].

The ray tracing method traces a path of light from the camera through each pixel in the image plane and simulates the light interaction with objects in the scene. Ray tracing can simulate a wide variety of optical effects, such as reflection and refraction, scattering and chromatic aberration. A light ray - projected from camera to the scene - is tested to see if it intersects with any of the objects in the scene and from there whether it will intersect with any of the scene’s light sources. Certain illumination algorithms and reflective or translucent materials may require further rays to be cast into the scene. Using ray tracing, a shader recursively traces rays until a light source is hit, or a predefined maximum recursion depth is reached, to calculate the total light contribution for each pixel, which is used
Figure 1.1: The ray tracing algorithm, where rays are traced from the camera through the image plane into the scene.

to determine the colour [26]. After the maximum recursion depth is reached, it is assumed, even if a light source will be reached, that the shading contribution is insignificant, and ray tracing terminates with no light contribution for the given pixel.

This concept is illustrated in Figure 1.1. The figure shows a camera and primary rays (in red) projected through the image plane into the scene, which contains a large blue sphere. Secondary rays (in green) are created from the interaction between primary rays and scene objects, depending on the properties of the given object. Secondary rays that eventually make their way to a light source contribute to the brightness of the pixel.

While it may seem counter-intuitive to cast rays into a scene rather than from a source (a process known as photon mapping), it is far more efficient as the overwhelming majority of light rays from a source do not reach the camera. Time is therefore not wasted computing paths for rays that will never reach the camera.

GPUs are typically used to accelerate ray tracing, providing performance of up to millions of rays per second, but use significant power to do so. Because of the long run times and significant
computation requirements, air-conditioning and power costs can be over 50% of the cost of running a render farm [74].

1.2 FIELD PROGRAMMABLE GATE ARRAYS

A field-programmable gate array (FPGA) is essentially a programmable integrated circuit (IC). FPGAs were invented by Xilinx in 1984 and fill the gap between general purpose processors (GPPs) and application-specific integrated circuits (ASICs), offering parallelisation and performance similar to what an ASIC can provide while being flexible and easily programmable like a GPP.

FPGAs can be used to design application-specific functional units that are very efficient for a given task by parallelising several operations. The functional units themselves can also be parallelised by replicating them and fitting multiple instances on the FPGA at once. The parallelism in reconfigurable hardware is also potentially much higher than in software and can be exploited on a much lower level so even greater speed-ups are possible if applications can be formulated in a parallel way. The abundant parallelism in ray tracing can be exploited in this way.

The memory architecture of FPGAs is also configurable, allowing creative data-path bandwidth optimisation. The inflexible and small amount of memory available to computation threads on GPUs can limit ray tracing performance, but could be side-stepped using a customized FPGA memory architecture.

FPGAs show greater power efficiency than GPUs and CPUs [16] due to their architecture and lower clock rate (which are typically hundreds of MHz, compared to GHz for CPUs and GPUs), so may be an alternative to GPUs when ray tracing power consumption is a concern to render farm operators or for embedded environments.

1.3 MOTIVATION AND CONTRIBUTIONS

Motivation  Previous research into the use of FPGAs in ray tracing has shown underwhelming results compared to GPUs, which dominate the field. However, that previous work was completed on now dated hardware (i.e. a Xilinx Virtex-2), three generations older than the FPGA used in this thesis (Xilinx Virtex-5)
and six generations older than is current at the time of writing (Xilinx UltraScale). FPGA architecture development has been rapid over the generations, delivering faster speeds, exponentially more logic, lower power usage and integration with faster memory and communication interfaces [88]. It is puzzling as to why more ray tracing research has not been undertaken that utilizes FPGAs, as although a large portion of ray tracing is parallel floating-point operations, that GPUs excel at, there is abundant parallelism that may be exploited by FPGAs in ways that GPUs can not. The productivity gap between the exponentially increasing amount of logic found on FPGAs and the number of man-hours taken to develop applications for them is one explanation, and an area which is currently a hot topic in FPGA research. The power efficiency of FPGAs in ray tracing acceleration has not been investigated sufficiently, and efficiency improvements are becoming more and more relevant with even larger rendering installations. The key motivations and objectives of this research are:

1. providing a platform to easily and efficiently explore parallelism in ray tracing on FPGAs to decrease this productivity gap,
2. providing building blocks and optimisations that enable ray tracing efficiency on FPGAs, and
3. to demonstrate that FPGAs have a great potential for power and bandwidth efficiency when accelerating ray tracing.

Contributions Given these motivations, the following contributions were made while undertaking the research presented in this thesis:

* Chapter 3: Exploration into the current state of the art of ray tracing hardware acceleration, identifying areas of improvement and new areas to exploit parallelism.

* Chapter 4: Investigation and proposal of an innovative truly MIMD FPGA ray tracing platform to easily and efficiently explore hardware acceleration of ray tracing on FPGAs. The presented platform is agnostic to different acceleration hierarchies, allows traversal of programmable widths of acceleration hierarchies, allows traversal of multiple rays at a time and intersection of each ray with multiple objects, and is
modular where units need to be replicated or interchanged (i.e. the generic and scalable traversal unit with interchangeable priority queue).

- **Chapter 6**: Investigation and proposal of a **hybrid hardware combination of a single-cycle queue and a heap**, combining the speed of the single-cycle queue with the scalability of the heap. Development of a **method to simulate different priority queue workloads**, to evaluate queue performance under different conditions. **Optimization of heap FPGA implementation** to handle ray tracing type workloads.

- **Chapter 7**: Exploration and **analysis of memory access patterns** to the acceleration hierarchy and scene primitives during ray tracing on the platform, leading to development of a **novel node cache replacement policy**, based not on recent use, but on a nodes level on the acceleration hierarchy. **Design and development of a cache for ray tracing**, alleviating the memory bottleneck to performance.

- **Chapter 8**: Extensive **experimental evaluation of the proposed platform on real hardware**, investigating the effect of priority queue type on performance and scalability, the impact of a cache on platform performance and the performance of the platform in relation to a CPU and GPU.

1.4 **THESIS STRUCTURE**

This thesis begins, in Chapter 2, by detailing the basic concepts of ray tracing. The general principles of ray-object intersection are outlined, followed by a detailed explanation of ray-triangle and ray-box intersection, two ray-object intersection tests commonly required during ray tracing. Spatial and object hierarchy acceleration structures, that restrict the number of ray-triangle intersection calculations along a ray path, are then presented. The simple Cornell Box scene is also presented, which will be used throughout the thesis to demonstrate ray tracing concepts. The chapter finishes by presenting an outline of typical rendering software and the process of creating a scene.

Chapter 3 outlines hardware acceleration and presents a review of current research into the hardware acceleration of ray tracing.
Hardware acceleration architectures are first described, followed by a more detailed examination of FPGA architecture and development. PCIe express, a high-speed interconnect that typically connects hardware accelerators to systems, is then detailed. Finally, previous hardware acceleration of ray tracing is presented, including algorithms to overcome the memory limitations of GPUs, ray tracing on SIMD/MIMD architectures, ray tracing implementations on FPGAs and ray tracing specific cache architectures.

Chapter 4 presents the primary design goals of a flexible FPGA ray tracing platform that aims to solve the first motivation of the thesis: to decrease the productivity gap between the exponentially increasing amount of logic found on FPGAs and the number of man-hours taken to develop applications for them. The design of how the hardware and software components of the platform integrate and interact with each other is then outlined and finally the implementation and workflow for development and debugging of the platform is presented.

The remaining chapters aim to solve the second motivation of the thesis: to provide building blocks and optimisations that enable ray tracing efficiency on FPGAs.

Chapter 5 evaluates deep, floating-point pipelines that implement ray-triangle and ray-box intersection tests on the platform. The pipelines are designed using the data-flow of their respective intersection equations, with optimisations made to reduce the latency of the pipelines. The pipeline’s resource utilization and performance is evaluated, with both pipelines showing potential for scalability and improved performance over other accelerated implementations.

Chapter 6 details the implementation and operation of fast, single-cycle queues and slower, logic-efficient heaps. The two queue types are combined into a hybrid queue that exploits the speed of a single-cycle queue while maintaining the scalability of the heap. Optimizations to the heap are presented for efficient performance and implementation on the ray tracing platform.

Chapter 7 presents four test scenes and analyses memory access patterns of the acceleration hierarchy and scene primitives during their ray tracing on the platform. Caches for scene nodes and primitives are presented and implemented in BlockRAM on the FPGA. A novel node cache replacement policy is presented that replaces nodes based on their level in the acceleration hierarchy instead of by recent use.
Chapter 8 presents an evaluation of the platform performance. First, the effect of priority queue type on performance and scalability is investigated. Second, after memory bandwidth is found to be a bottleneck to platform performance, the impact of a cache on platform performance is investigated. Finally, the performance of the platform is put into relation with CPU and GPU performance, in terms of: raw performance, energy efficiency, and the efficiency of each platform relative to their available memory bandwidth.

Finally, Chapter 9 concludes this thesis. The outcomes of this work are recapped and reconsidered within the context of the thesis motivations and a number of directions and suggestions for future research are presented.
Ray tracing creates very visually realistic images by accurately simulating the interaction of light with objects within a scene. As shown previously in Figure 1.1, ray tracing works by tracing a path of light from a virtual camera through each pixel in the image plane to calculate the colour of the object visible through it. Whenever a ray is created to be cast into the scene from the camera, the first task of a ray tracing engine is to determine which object, if any, the ray intersects and where the intersection occurs. The location of the intersection is the visible point along the ray and where the interaction of light with the object is simulated. To determine the intersection point, a naive approach is to test the ray for intersection with each object in the scene.

To determine intersection, the ray, \( \mathbf{R} \), is first put in the parametric form shown in Equation (2.1).

\[
\mathbf{R}(t) = \mathbf{O} + t\mathbf{D}
\]  

where \( \mathbf{O} \) is the ray origin, \( \mathbf{D} \) is the unit vector of the ray direction and \( t \) is a parameter with legal range \([0, \infty)\). Any point along the ray can be obtained by specifying a value for \( t \) and solving Equation (2.1). The value for \( \mathbf{O} \) is the position of the camera for primary rays and the unit ray direction vector, \( \mathbf{D} \), is created using the origin and the position of the pixel the ray is to pass through.

Then, using Equation (2.1), it is often easy to find the intersection of a ray with a surface defined by a function in the form \( F(x, y, z) = 0 \). The ray equation is substituted into the equation, producing an equation whose only parameter is \( t \). As an example, Equation (2.2) gives the equation for a sphere centred at the origin with radius \( r \), such as the one shown in Figure 1.1.

\[ x^2 + y^2 + z^2 - r^2 = 0 \]  

(2.2)

Then, substituting the ray equation gives Equation (2.3).

\[ (\mathbf{O} + t\mathbf{D})_x^2 + (\mathbf{O} + t\mathbf{D})_y^2 + (\mathbf{O} + t\mathbf{D})_z^2 - r^2 = 0 \]  

(2.3)
This produces an easily solvable quadratic equation for \( t \). If there are no real roots, then the ray must miss the sphere. If there are real roots, then the smallest positive root gives the closest intersection.

This chapter outlines, in Section 2.1, two essential ray-object intersection tests required during ray tracing. Acceleration hierarchies that trade expensive ray-triangle intersection tests for traversal of their structure are then described in Section 2.2. In Section 2.3, a common test scene, the Cornell Box, is presented as an example scene to explain ray tracing concepts in later sections. Finally, software implementations of ray tracing are detailed in Section 2.4.

### 2.1 Ray-Object Intersection

This section outlines two ray-object intersection tests required during ray tracing. Ray-triangle intersection, described in Section 2.1.1, is used to determine if a ray intersects with one of the scene primitives and ray-box intersection, described in Section 2.1.2, is used to determine if a ray intersects with a bounding box, required during acceleration hierarchy traversal, detailed later in Section 2.2.

#### 2.1.1 Ray-Triangle Intersection

A key ray tracing operation is ray-triangle intersection (RTI), which tests whether the back-projected ray intersects a scene triangle and requires several repetitive vector calculations.

Möller and Trumbore’s algorithm \([55]\) for RTI tests is commonly used in digital rendering programs, including LuxRender, where scene objects are represented by a mesh of triangles. The algorithm defines a ray, \( \mathbf{R}(t) \), as shown in Equation (2.1), by its point of origin, \( \mathbf{O} \), and normalised direction vector, \( \mathbf{D} \), and a triangle by its three vertices, \( \mathbf{V}_0, \mathbf{V}_1 \) and \( \mathbf{V}_2 \). \( \mathbf{O} \) is the location of the camera lens, \( \mathbf{D} \) is the direction from the lens through a pixel in the image and the triangle is part of an object’s triangle mesh.

Barycentric coordinates provide a way to parametrise a triangle in terms of two variables, \( u \) and \( v \):

\[
\mathbf{T}(u, v) = (1 - u - v)\mathbf{V}_0 + u\mathbf{V}_1 + v\mathbf{V}_2
\]  \hspace{1cm} (2.4)
where $u$ and $v$ must satisfy $u \geq 0$, $v \geq 0$ and $u + v \leq 1$.

Finding the intersection of a ray, $R(t)$, and triangle, $T(u, v)$, is equivalent to $R(t) = T(u, v)$. Substituting for Equation (2.1) and Equation (2.4) gives:

$$O + tD = (1-u-v)V_0 + uV_1 + vV_2$$  \(2.5\)

Rearranging these terms gives:

$$[-D, V_1 - V_0, V_2 - V_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$  \(2.6\)

This means that the distance, $t$, to the intersection and the coordinates, $u$ and $v$, of the intersection can be found by solving the linear equation above.

This process can be thought of, geometrically, as translating the triangle to the origin and transforming it into a unit triangle in $y$ and $z$, with the ray aligned along $x$. This process is shown in Figure 2.1, where $M = [-D, V_1 - V_0, V_2 - V_0]$ is the matrix from Equation (2.6).

Using Cramer’s rule and the determinant identity $|a \ b \ c| = -(a \times c) \cdot b = -(c \times b) \cdot a$, Equation (2.6) can be rewritten as:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$  \(2.7\)
where

\[ E_1 = V_1 - V_0 \]  \hspace{1cm} (2.8a)  
\[ E_2 = V_2 - V_0 \]  \hspace{1cm} (2.8b)  
\[ T = O - V_0 \]  \hspace{1cm} (2.8c)  
\[ P = D \times E_2 \]  \hspace{1cm} (2.8d)  
\[ Q = T \times E_1 \]  \hspace{1cm} (2.8e)  

The coordinates of the intersection, \( u \) and \( v \), are barycentric, \textit{i.e.} an intersection has occurred only when Equations (2.9a) – (2.9c) are satisfied.

\[ u \geq 0 \]  \hspace{1cm} (2.9a)  
\[ v \geq 0 \]  \hspace{1cm} (2.9b)  
\[ u + v \leq 1 \]  \hspace{1cm} (2.9c)  

As this process is reasonably complex to compute, the acceleration structures described later in Section 2.2 are implemented to quickly discard whole groups of triangles during the ray intersection process.

2.1.2 Ray-Box Intersection

All of the acceleration structures described later in Section 2.2 store an axis-aligned bounding-box (AABB) that surrounds all the primitives in the scene. This box can be used to quickly determine if a ray intersects any of the primitives in the scene; if a ray misses the box, it must also miss all of the primitives inside it. This section outlines how to determine whether a ray intersects with a box and, if an intersection occurs, the distance to intersection.
Bounding-boxes can be thought of as the intersection of three slabs, where a slab is the region of space between two parallel planes, and the slabs are each aligned with one of the three axes. To intersect a ray with a box, the ray is intersected with the box’s three slabs in turn. As the slabs are axis-aligned, a number of optimisations can be made to the ray-slab tests. Figure 2.2 shows a 2-dimensional box, with the slab in the x dimension defined between \( b_{\text{Min}}^x \) and \( b_{\text{Max}}^x \) and the slab in the y dimension defined between \( b_{\text{Min}}^y \) and \( b_{\text{Max}}^y \).

The basic ray-box intersection algorithm works as follows:

- A parametric interval is created that covers the range of positions, \([t_{\text{min}}, t_{\text{max}}]\), along the ray \( t \) where intersection can occur. \( t_{\text{min}} \) is typically set to zero and \( t_{\text{max}} \) typically infinity.

- Then, successively for each slab, two slab intersection positions, \( t_{\text{Near}} \) and \( t_{\text{Far}} \), are calculated where the ray intersects with the given slab. \( t_{\text{Near}} \) and \( t_{\text{Far}} \) values are labelled for the x dimension on Figure 2.2.

- The per-slab intersection values (\( t_{\text{Near}} \) and \( t_{\text{Far}} \)) and current parametric interval are set intersected to create the new parametric interval. If the resulting interval is degenerate (i.e. \( t_{\text{Near}} > t_{\text{Far}} \)), the ray does not intersect with the bounding-box and failure is returned.

- If, after checking all three slabs, the interval is non-degenerate, the interval represents the range of \( t \) where the ray is inside the box.

Figure 2.2 shows this process for a 2-dimensional box surrounding a triangle. The red section of the ray shows the final computed interval inside the box.

Equations (2.10a) – (2.10b) shows the calculations required to determine ray-slab intersection for one axis.

\[
\begin{align*}
t_{\text{Near}} &= \frac{b_{\text{Min}} - O}{D} \quad (2.10a) \\
t_{\text{Far}} &= \frac{b_{\text{Max}} - O}{D} \quad (2.10b)
\end{align*}
\]

\( b_{\text{Min}} \) and \( b_{\text{Max}} \) each represent one of the coordinates of the slabs for the given dimension and \( O \) and \( D \) represent the ray origin and direction for the given dimension.
Acceleration of the rendering process can be achieved through spatial or object hierarchy structures. These structures aim to restrict the number of ray-triangle intersection calculations along a ray path by trading them for traversal of the structure. Use of these structures requires implementing two algorithms: construction and traversal of the data structure. Construction creates the acceleration hierarchy once, initially, and then the traversal process can be repeated as many times as required for the given hierarchy, amortising the cost of hierarchy construction.

2.2 ACCELERATION STRUCTURES

A two-dimensional \( k \)-D-tree and the corresponding graph representation, presented by Foley et al. [32], is shown in Figure 2.3. Internal nodes are labelled next to their split planes and leaf nodes are labelled inside their area.

The following sections outline typical \( k \)-D-tree construction and traversal. An efficient traversal algorithm is important as the process of determining the triangle a ray has hit can represent over 90% of total rendering time [64, 96].

Figure 2.3: A two-dimensional \( k \)-D-tree and corresponding graph representation.
Construction

A kD-tree is commonly constructed recursively in a top-down manner. A root node is created that represents a bounding volume containing all of the objects in a scene. Starting with the root node, and for each subsequent node created, a decision is made to either split the node into two internal nodes or to declare it a leaf node.

When splitting, a high quality kD-tree can be constructed using greedy cost optimisation based on a surface area heuristic (SAH) \[90]. The SAH estimates the cost of splitting the current volume on a particular plane according to the probability of intersection with the children created. For a volume \( V \) that is split into two sub-volumes, \( V_L \) and \( V_R \), the probability of a ray traversing into these sub-volume is given by:

\[
P(V_L|V) = \frac{SA(V_L)}{SA(V)} \quad (2.11a)
\]

\[
P(V_R|V) = \frac{SA(V_R)}{SA(V)} \quad (2.11b)
\]

where \( SA(V) \) is the surface area of volume \( V \).

Given this probability at a potential split position, the cost of splitting can be estimated by:

\[
C_{split} = C_{rbi} + C_{rti}N_LP(V_L|V) + C_{rti}N_RP(V_R|V) \quad (2.12)
\]

where \( N_L \) and \( N_R \) are the number of triangles in \( V_L \) and \( V_R \) respectively, \( C_{rti} \) is the cost of a ray-triangle intersection test, described in Section 2.1.1, and \( C_{rbi} \) is the cost of a ray-box intersection test, described in Section 2.1.2. This means that instead of just testing for intersection with the set of triangles (at a cost of \( C_{rti} \times (N_R + N_L) \)), a node will need to be traversed (with cost \( C_{rbi}\)), then possibly (with probability \( P(V_L|V) \)) intersect with the first sub-volume of triangles (with cost \( C_{rti}N_L \)) and then possibly (with probability \( P(V_R|V) \)) intersect with the second sub-volume of triangles (with cost \( C_{rti}N_R \)). A split will therefore only be made when the cost of splitting is less than the cost of simply intersecting with the remaining triangles. Greedy decisions are made at each step to avoid this search becoming non-polynomial \[64\].
1 Function Traverse(tree, ray) begin
2     (tmin, tmax) = IntersectBBox(tree.root, ray);
3     SearchNode(tree.root, ray, tmin, tmax);
4 end

5 Function SearchNode(node, ray, tmin, tmax) begin
6     if IsLeaf(node) then
7         SearchLeaf(node, ray, tmin, tmax);
8     else
9         SearchInternal(node, ray, tmin, tmax);
10    end
11 end

12 Function SearchInternal(node, ray, tmin, tmax) begin
13     thit = IntersectBBox(node, ray);
14     (near, far) = Order(node.left, node.right);
15     if thit ≥ tmax or thit < 0 then
16         SearchNode(near, ray, tmin, tmax);
17     else if thit ≤ tmin then
18         SearchNode(far, ray, tmin, tmax);
19     else
20         Push(pq, (far, thit, tmax));
21         SearchNode(near, ray, tmin, tmax);
22    end
23 end

24 Function SearchLeaf(node, ray, tmin, tmax) begin
25     foreach triangle in the node do
26         thit = IntersectTriangle(triangle, ray);
27         if thit < tmax then
28             Succeed(thit);
29     end
30    end
31    ContinueSearch(node, ray, tmin, tmax);
32 end

33 Function ContinueSearch(node, ray, tmin, tmax) begin
34     if IsEmpty(pq) then
35         Fail();
36     else
37         (node, tmin, tmax) = Pop(pq);
38         SearchNode(node, ray, tmin, tmax);
39     end
40 end

Algorithm 2.1: Recursive kD-tree traversal.

Traversal

Pseudo code of a typical kD-tree traversal algorithm for ray tracing is shown in Algorithm 2.1. Inputs to the algorithm are the tree and a
ray. Starting at the root, the ray is walked down the tree so that intersection is tested with triangles in front-to-back order. A priority-queue (a priority ordered list of nodes left to visit) is used to ensure each node is traversed at most once and each necessary node exactly once. When traversing a node the algorithm must determine whether any child nodes can be skipped and what order to traverse the children (i.e. which node is near and which is far). These steps are shown on lines 13 and 14 of Algorithm 2.1, respectively.

The order of nodes, i.e. near to far, is determined by ordering by ascending intersection distances along the given ray. Methods using the ray origin or ray direction can be used to estimate the order if intersection distance is not available or expensive to compute. After node classification, there are three possible cases for further traversal: (i) visit only the near node, (ii) visit only the far node and (iii) visit the near node followed by the far node. The priority-queue stores the far node when both need to be visited. Lines 15 and 17 of Algorithm 2.1 show the node traversal decision using the ray origin to determine the order of child node traversal [37].

Traversal continues down a tree until a leaf node (containing triangles) is encountered. RTI tests are performed on all triangles within the leaf and if the ray intersects any triangles, the closest intersection is guaranteed to be the first intersection along the ray and traversal terminates. If no intersection is found, a node is popped from the priority-queue and traversal continues. If the priority-queue is empty then traversal terminates without intersection.

The worst case performance of the algorithm for \( n \) leaf nodes is \( O(n) \), where the ray may visit a number of nodes linear to the size of the tree. However, in practice it is expected that the ray will find an intersection within one of the first leaf nodes visited and performance is typically \( O(\log n) \) [39].

2.2.2 Object Partitioning

Bounding Volume Hierarchies (BVHs), instead of splitting the scene space like kD-trees, recursively split the set of scene primitives until the leaf size (number of primitives) meets a given criterion. Each internal node stores a bounding box surrounding all child nodes. The advantage of this technique is that each primitive is stored in
the hierarchy exactly once, but traversal can become inefficient when volumes overlap each other.

Construction

BVHs are constructed top-down recursively by dividing the scene’s primitives into two subsets. To determine the split position, the cheapest cost of traversal is determined using the SAH, as is done in kD-tree construction. Once the split is determined, an AABB is created around the subset of primitives and assigned to the child node. This process continues until the cost of splitting is higher than testing intersection with all the primitives, or the number of primitives is below a user defined criterion. Stich et al. [82] presented a technique that uses spatial splits, similar to those in kD-tree construction, to construct significantly more efficient BVHs than previous techniques.

Traversal

A BVH is traversed in the same way as a kD-tree. Given a ray, a typical priority-queue based traversal algorithm starts by intersecting the ray with the bounding boxes contained at the root node. The pointers of the children with a non-empty bounding box intersection are sorted and then pushed on to the priority-queue. The routine is repeated by popping the next element, performing ray-triangle intersection if it is a leaf or ray-box intersection if it is an internal node, with the same criteria for completion as priority-queue-based kD-tree traversal.

2.3 Cornell Box

The Cornell Box, shown in Figure 2.4, was a test originally aimed at determining the accuracy of rendering software by comparing the rendered scene with an actual photograph of the same scene, and has now become a commonly used test model [36]. The basic environment consists of: a light source in the centre of a white ceiling, a green right wall, a red left wall, a white back wall, a white floor and two white boxes on the floor within the walls.

An example setup of this scene is included within LuxRender to render at a $512 \times 512$ resolution with a maximum ray path depth of five, i.e. for each primary (coherent) ray that intersects with a
Figure 2.4: The simple Cornell scene used as an example to explain ray-tracing concepts.
primitive, the secondary (incoherent) ray paths generated have a maximum recursion depth of five. As the geometries of the scene objects are rather simple, the scene can be represented by only 32 triangle primitives. Such a basic scene is not representative of what would typically be ray-traced by movie-makers, but is perfect for use as an example to illustrate ray-tracing concepts throughout the remainder of this thesis.

### 2.4 Rendering Software

Rendering is the process of generating an image from a set of models that collectively make a scene, by means of computer software. The results of this process are also often referred to as a rendering. Rendering software is generally made up of two parts; 1) the modelling interface to create the scene, often referred to as the rendering front-end, and 2) the rendering engine that transforms the model into a high quality image, often referred to as the rendering back-end. A scene is defined by modelling software using strictly defined data structures that contain the geometry, viewpoint, texture, lighting, and shading information that together compose the virtual scene. The data contained in the scene file is then passed to a rendering engine to be processed and output to a digital image file using techniques such as ray tracing.

Blender is a common open-source modelling program used for creating high quality scenes for animated films and visual effects \[33\]. Figure 2.5 shows the Cornell Box scene from Section 2.3 as setup in Blender. Within the scene the wire-frame representation of the cubes and surrounding walls that make up the room can be seen. The pink square at the top of the room represents the light source and the camera viewpoint can be seen as the dashed rectangle near the perimeter of the walls.

A typical rendering engine implements all the algorithms presented in this chapter efficiently in software for a given platform. A wide range of software exists and notable among them are RenderMan (proprietary), Maya (proprietary), MentalRay (proprietary), OptiX (proprietary), Pixie (GPL) and LuxRender (GPL).

LuxRender is a modern, hardware-acceleration-focused and currently developed rendering engine \[52\]. LuxRays, a subsection of
the LuxRender suite dedicated to hardware acceleration, is an open-source mixed C++ and OpenCL library. It implements ray tracing for both QBVH and $kD$-tree acceleration hierarchies and has an OpenCL implementation that enables ray-tracing on CPUs and GPUs that support version 1.1 or newer of OpenCL.
Hardware acceleration is the use of computer hardware to perform specific functions faster than is possible in software running on a more general-purpose CPU. Examples of hardware acceleration include shading pipelines in graphics processing units (GPUs) and physics processing units (PPUs) for fast physics simulations for video games.

Typical processors execute instructions one by one, sequentially. There are several techniques used to improve performance of sequential processors of which hardware acceleration is one. Hardware is able to more easily exploit concurrency than software, often allowing hardware to be much faster than software. Hardware accelerators are designed for computationally intensive software code, such as those with repetitive vector calculations. Depending on the granularity of the parallelism in a target application, hardware acceleration can vary from a small functional unit to a large functional block.

Given a hierarchy of general-purpose processors such as CPUs, more specialized processors such as GPUs, fixed-function implementations on FPGAs, and fixed-function implementations on ASICs; there is a trade-off between flexibility and efficiency, with efficiency increasing when an application is implemented higher up that hierarchy, while flexibility decreases.

This chapter first outlines the classification of typical architectures used for hardware acceleration in Section 3.1. The architecture of FPGAs, which offer parallelisation and performance similar to what an ASIC can provide while being flexible and easily programmable like a GPP, is outlined in Section 3.2. Then PCIexpress, a high-speed serial expansion bus used to attach accelerators like GPUs and FPGAs is outlined in Section 3.3. Finally, ray tracing algorithms developed to overcome the memory limitations of GPUs, algorithms for ray tracing on SIMD/MIMD architectures, previous ray tracing implementations on FPGAs, and ray tracing specific cache research is presented in Section 3.4.
Computer architectures can be characterised using Flynn’s taxonomy \([39]\), shown in Table 3.1, based on the parallelism in both the instruction and data streams.

**SISD** represents a sequential processor which exploits no parallelism in either the instruction or data streams. The classic CPU can be classified as a SISD device, however most modern processors typically also include an SIMD component, like the MMX/SSE units found in Intel’s Pentium and Core architectures.

**MISD** represents an architecture where multiple instructions operate on a single data stream. It is an uncommon architecture which is generally used for fault tolerance, where multiple heterogeneous systems operate on the same data stream and must agree on the result. An example application could be a space-shuttle flight computer that requires high reliability.

**SIMD** describes an architecture which processes multiple data streams against a single instruction stream to perform operations which may be naturally parallelised. SIMD architectures are commonly used as hardware accelerators, with wide SIMD units available on many modern hardware architectures such as 16-wide AVX on Intel CPUs \([42]\), 16-wide SIMD on Intel Many Integrated Core (Intel MIC) \([44]\), and 16-or-more-wide SIMD on GPUs \([1]\). SIMD provides high performance by sharing the hardware for caches and instruction decode over many ALUs. However, using wide SIMD hardware efficiently can be challenging, since algorithmic changes must be made to occupy all SIMD lanes through the majority of execution.

**MIMD** represents an architecture where multiple autonomous processors simultaneously execute different instructions on different data. Distributed systems are generally MIMD architectures; either exploiting a single shared memory space or a distributed memory space. The Intel MIC architecture is also MIMD as it implements multiple processors, each with 16-wide SIMD capabilities \([44]\).
Flynn’s taxonomy is unable to be generally applied to the FPGA architecture as it is completely customizable and thus can implement any taxonomy. But as the advantage of FPGAs is their ability to implement parallel constructs, a SIMD or MIMD architecture would typically be employed, depending on the application.

3.2 FIELD PROGRAMMABLE GATE ARRAYS

A field-programmable gate array (FPGA) is essentially a programmable integrated circuit (IC). FPGAs fill the gap between general purpose processors (GPPs) and application-specific integrated circuits (ASICs), offering parallelisation and performance similar to what an ASIC can provide while being flexible and easily programmable like a GPP.

FPGAs can be used to design application-specific functional units that are very efficient for the given task (and only the given task) by parallelising operations. The functional units themselves can also be parallelised by replicating them and fitting multiple instances on the FPGA at once. FPGAs are good candidates for acceleration of applications with high degrees of parallelism. Further, the lower clock rates of FPGAs (which are typically hundreds of MHz, compared to GHz for CPUs and GPUs) mean they have low power requirements.

This section briefly describes the FPGA architecture in Section 3.2.1 and programmable configuration of FPGAs in Section 3.2.2.

3.2.1 Architecture

FPGAs contain a matrix of configurable logic blocks (CLBs), which each contain “slices”, connected via a hierarchy of reconfigurable interconnects that allow the blocks to be “wired together”. The contents of a slice is the core of FPGA functionality. A typical slice in a modern FPGA, such as those found in Xilinx FPGAs, contains:

- multiple 6-input look-up tables (LUTs); able to implement any arbitrarily defined six-input boolean function,
- multiple flip-flops (FFs) per LUT; also referred to as registers, that synchronously hold LUT output values,
- distributed memory and shift register logic capability,
a dedicated carry-chain; to perform fast arithmetic addition and subtraction in a slice, and

dedicated multiplexers, allowing LUTs to be combined.

FPGAs also contain additional general features like BlockRAM, FIFOs, DSP blocks and clocking resources (such as PLLs). Families of FPGAs may also have device specific features like dedicated PCIe express cores, dedicated Ethernet cores and dedicated DDR2/3 memory interfaces. Figure 3.1 shows an example layout of a Xilinx Spartan-6 FPGA, labelling one of the dedicated PCIe express cores, one of the memory interfaces, one of the BlockRAM columns and one of the DSP columns. The CLBs are shown in blue. The FPGA used for the target platform in Chapter 4 is much larger than the example shown, but the regularity of the layout and features are the same.
3.2.2 Configuration

The structure, design and operation of digital logic circuits can be programmed using a hardware description language (HDL), typically VHDL or Verilog. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and verification of an electronic circuit. It also allows for the compilation of an HDL program into a lower level specification of physical electronic components, such as the set of masks used to create an IC.

A HDL looks much like any other programming language; it is a textual description consisting of expressions, statements and control structures. The important difference from most programming languages is that HDLs explicitly include the notion of time, a primary attribute of IC design. This notion of time allows HDL code to describe register-transfer level designs, modelling the flow of data between synchronous storage elements (registers in the case of FPGAs).

After writing the description of an application in a HDL, it must then be synthesized (analogous to the word “compiled” for GPP software), implemented and programmed onto the FPGA. The steps in the following three paragraphs outline the process of configuring the target FPGA with an application design using the Xilinx Integrated Software Environment (ISE). The steps are specific to the Xilinx software, however other vendors use similar processes.

**Synthesis** During synthesis, the synthesis engine compiles the design to transform HDL sources into an architecture-specific design netlist. The Xilinx synthesizer is called XST, which synthesizes VHDL, Verilog, or mixed language designs to create a Xilinx-specific netlist file, known as an NGC file, which contains both logical design data and constraints. The NGC contains the design in terms of logic elements, optimised for the target architecture or “technology”, *i.e.* in terms of LUTs, carry logic, I/O buffers, and other technology-specific components.

**Implementation** After synthesis, design implementation is run, which converts the logical design into a physical file format that can be downloaded to the selected target device. The implementation process consists of the following named steps:
- **Translate**: the incoming netlists and constraints are merged into a Xilinx design file.

- **Map**: the design is fit into the available resources on the target device, and optionally, the design is partially placed on the device.

- **Place and Route**: the design components are placed and routed on the target device, according to the timing constraints.

- **Generate Programming File**: a bitstream file is created in a format that can be downloaded to the device.

**Device Configuration and Programming** After generating the programming file, the FPGA can be configured by transferring the bitstream either directly to the FPGA or into a programmable read-only memory (PROM) which is attached to the FPGA. This is done by connecting a (typically USB-JTAG) cable to an appropriate port on the host computer and to the correct pins on the target board and running the device configuration software within the Xilinx ISE. If the bitstream is programmed directly to the FPGA, it must be reprogrammed each time after power to the device is cycled. If programmed to a PROM then the bitstream is read from the PROM when power is applied to the device.

### 3.3 PCI EXPRESS

PCIexpress is a packet-based high-speed serial expansion bus standard designed to replace the older PCI, PCI-X, and AGP bus standards [71]. PCIexpress is based on point-to-point topology, with separate serial links connecting every device to the host, and offering extremely high bandwidth, such as 15.75 GB/s for a 16 lane PCIexpress 3.0 connection. PCIexpress operates in consumer, server, and industrial applications, as a motherboard-level interconnect (to link motherboard-mounted peripherals), a passive backplane interconnect and as an expansion card interface for add-in boards. In virtually all modern computers, from consumer laptops and desktops to enterprise data servers, the PCIexpress bus serves as the primary motherboard-level interconnect, connecting the host system-processor with both integrated-peripherals (surface-mounted ICs) and add-on peripherals (expansion cards).
The PCIexpress standard defines slots and connectors for multiple widths: $\times 1$, $\times 4$, $\times 8$, $\times 12$, $\times 16$ and $\times 32$. This allows PCIexpress bus to serve both cost-sensitive applications where high throughput is not needed, as well as performance-critical applications such as 3D graphics, networking (e.g. 10-gigabit Ethernet), and enterprise storage (e.g. SAS or Fibre Channel). Hardware acceleration devices such as GPUs and FPGA development boards are typically connected to systems through their PCIexpress bus.

PCIexpress is a layered protocol, consisting of a physical layer, a data link layer, and a transaction layer, each described in the following paragraphs.

**Physical Layer** The physical layer (PHY) specification is divided into two sub-layers, corresponding to electrical and logical specifications. The logical sub-layer is sometimes further divided into a media access control (MAC) sub-layer and a physical coding sub-layer (PCS), which are terms borrowed from the IEEE 802 networking protocol model [8], although this division is not formally part of the PCIexpress specification. At the electrical level, each lane consists of two unidirectional differential pairs, for a total of four data wires per lane. The electrical level also contains hardware such as serializers/deserializers and clocking resources.

**Data Link Layer** The DLL primarily performs three services for the PCIexpress link:

- sequence the transaction layer packets (TLPs) that are generated by the transaction layer,

- ensure reliable delivery of TLPs between two endpoints via an acknowledgement protocol (ACK and NAK signalling) that explicitly requires replay of unacknowledged/bad TLPs, and

- manage link flow control (such as rate-limiting).

On the transmitting side, the data link layer generates an incrementing sequence number for each outgoing TLP. It serves as a unique identification tag for each transmitted TLP, and is inserted into the header of the outgoing TLP. A 32-bit cyclic redundancy check code (known on the DLL as a Link CRC or LCRC) is also appended to the tail of each outgoing TLP.
**Transaction Layer** PCIe express implements split transactions, where the request and response are separated by time, allowing the link to carry other traffic while the target device gathers data for the response. There are four PCIe express transaction types:

- I/O reads and writes; legacy transactions supported only for backward compatibility with previous standards,
- configuration reads and writes; used by the host to set-up the device configuration space, which shows the device information (such as vendor and product information), capabilities (such as power modes) and configuration (enabling/disabling said power modes),
- memory reads and writes; used to transfer data to and from memory on the host, and
- messaging; used for error signalling, hot-plug signalling and interrupt signalling.

The above transaction types are each implemented through different TLPs, which the transaction layer passes to the DLL to be sent to the host. A hardware application that interfaces with a host through PCIe express, such as the platform presented in Chapter 4, uses the transaction layer to send and receive data through memory read and write TLPs. There are three typical memory TLP types used by an application for transmitting and receiving data over the PCIe express link:

- **Memory Write Request** (MWr): used to write data of specified size, from the transmitting side to a specific address on the receiving side.
- **Memory Read Request** (MRd): used by the transmitting side to request data at a specific address and specified size from the receiving side.
- **Completion with Data** (CplD): used by the transmitting side to return requested data (using MRd) by the receiving side.

3.4 **Hardware Acceleration of Ray Tracing**

The abundant amount of parallelism in ray tracing, long required rendering times and use of ray tracing in the financially-large
industry of movie making has lead to much research into the hardware acceleration of ray tracing. This section outlines ray tracing algorithms developed to overcome the memory limitations of GPUs in Section 3.4.1, followed by algorithms for ray tracing on SIMD/MIMD architectures in Section 3.4.2, then outlines previous ray tracing implementations on FPGAs in Section 3.4.3 and finally outlines ray tracing specific cache research in Section 3.4.4.

3.4.1 Optimisations for GPUs

This section briefly outlines techniques developed to optimise ray tracing on GPUs. Techniques used in the presented platform are discussed in more detail later in Chapter 4. The majority of the techniques are developed to overcome the limited and inflexible amount of memory available to each execution thread on a GPU. These techniques show that an efficient priority queue implementation on GPUs is not achievable, and a stack is often used to remove complexity and the requirement for queue maintenance. If an FPGA were able to efficiently implement a priority queue it would avoid needing implementation of these techniques.

LONDEST COMMON TRAVERSAL SEQUENCE The longest common traversal sequence (LCTS) of rays within a pyramidal shaft (created from the camera through a pixel) can be found from the traversal of the four corner rays. At each node, the next traversal step (which node to traverse or set of primitives to intersect) is determined for each of the four rays and compared. If the next step is the same for each ray, and if only one node is to be visited, the four rays proceed to the next step. If the next step for any of the rays differs from the others, or if the next step is the same but more than one node is to be visited, the traversal stops. This node marks the end of the LCTS and it is guaranteed that all rays within the shaft traverse to this node. It is then used as the root node for traversal of all rays within the shaft, decreasing the size of the tree that has to be traversed for each ray. This technique only works well for coherent rays, and as the platform presented in Chapter 4 initially favours neither coherent or incoherent rays, this technique may be explored later if coherent ray performance need be optimized.
Figure 3.2: After failing to find an intersection in a leaf node, \textit{kd-restart} advances \texttt{tmin} and \texttt{tmax} so that the modified range now starts in the next leaf to be traversed.

33 \textbf{Function} ContinueSearch\( (\text{node}, \text{ray}, \text{tmin}, \text{tmax}) \) \textbf{begin}
34 \hspace{1em} \textbf{if} \texttt{tmax} == \texttt{global\_tmax} \textbf{then}
35 \hspace{2em} Fail();
36 \hspace{1em} \textbf{else}
37 \hspace{2em} \texttt{tmin} = \texttt{tmax};
38 \hspace{2em} \texttt{tmax} = \texttt{global\_tmax};
39 \hspace{2em} SearchNode(tree.root, ray, \texttt{tmin}, \texttt{tmax});
40 \hspace{1em} \textbf{end}
41 \textbf{end}

Algorithm 3.1: \textit{kd-restart} modifications to the \texttt{ContinueSearch} function.

\textbf{RESTART} The \textit{kd-restart} algorithm, presented by Foley \textit{et al.} [32], modifies the traditional traversal algorithm to eliminate the use of a stack altogether. If the \texttt{push} operation was removed from the \texttt{Search Internal} function on line 20 of Algorithm 2.1, the algorithm would proceed directly to the first leaf node intersected by the ray. With this modification, the only difference is the change to \texttt{tmax} if a node were to be placed on the stack. In such a case, the new value of \texttt{tmax} is exactly the value pushed onto the stack as \texttt{tmin}. When we reach a leaf node, the value of \texttt{tmax} is either the global \texttt{tmax} value, or the \texttt{tmin} value for the rays entry into the new leaf. When traversal reaches a leaf node and fails to find a hit, it simply restarts the search at the root of the tree with the value of \texttt{tmin} advanced to the end of the leaf. By repeating this process, all leaves along the ray are visited in the same order as the standard traversal algorithm at the expense of restarting traversal for each leaf node. The modifications to the \texttt{ContinueSearch} function are shown in Algorithm 3.1.

The cost of traversal to \(n\) leaf nodes is \(O(n \cdot h)\), where \(h\) is the height of the tree. This gives a worst case of \(O(n \log n)\) in a balanced tree. In practice, if we assume the number of leaf nodes visited is bounded by a small constant, then the expected cost will be \(O(\log n)\).
3.4 Hardware Acceleration of Ray Tracing

**Figure 3.3:** After searching a leaf node, *kd-backtrack* resumes the search at the first ancestor that intersects the modified range.

_Backtrack_ In the traditional algorithm, a node that has been pushed onto the stack is always a sibling of one of the current nodes ancestors. The _kd-backtrack_ algorithm, also presented by Foley _et al._ [32], modifies _kd-restart_ to store parent links at each node, making it possible to follow these links back up the tree to find the parent node atop the stack, removing the need for expensive restarts. Using the same tactic of advancing tmin to the end of the last leaf visited, the appropriate parent can be recognised as the closest ancestor that has a non-empty intersection with the remaining range, shown in Figure 3.3. The intersection can be determined using the AABB of each node, so this additional information must be stored at each internal node. The _ContinueSearch_ function to implement these changes is shown in Algorithm 3.2. These changes allow the algorithm to maintain the linear worst-case bounds for cost of traversal, at the expense of the additional per-node storage required.

_Push-down_ Horn _et al._ [40] presented _push-down_, a technique that moves the restart position down the tree to the lowest sub-tree enclosing the remaining nodes. As a ray descends from the root node through the tree, as long as it encounters nodes whose splitting planes it does not cross (i.e. it traverses only the near or only the far node), the restart location can be moved down the tree, reducing these redundant steps after a restart.

This optimisation is no longer beneficial when the ray intersects both near and far nodes, no matter how unlikely it is that the ray will make it to the far node before terminating. The benefits of this optimisation are inconsistent, as when a ray crosses one of the early splitting planes there will be no benefit at restart and a slight penalty for the extra bookkeeping.
Function ContinueSearch(node, ray, tmin, tmax) begin
  if tmax == global_tmax then
    Fail();
  else
    tmin = tmax;
    tmax = global_tmax;
    BackTrack(node.parent, ray, tmin, tmax);
  end
end

Function BackTrack(node, ray, tmin, tmax) begin
  \( (t_0, t_1) = \text{IntersectBBox}(node, ray) \);
  if \( t_0 \geq tmax \) then
    BackTrack(node.parent, ray, tmin, tmax);
  else
    SearchNode(node, ray, \( t_0, t_1 \));
  end
end

Algorithm 3.2: kd-backtrack modifications to the ContinueSearch function.

Short-stack Horn et al. [40] also presented short-stack, using a small, fixed-size stack and falling back to stack-less traversal if the stack should underflow. Pushing a new node onto the stack when the stack is full will cause the bottom-most entry to be discarded. Popping from an empty stack is no longer a cause for termination, but triggers a restart instead. The stack effectively acts as a cache, trading off the amount of per-ray cache for restart frequency. While short-stack is complimentary to push-down, it is more robust, only degrading when the stack fills, limiting the sensitivity to ray directions exhibited by push-down. Short-stack eliminates the overhead of a restart that would have happened within \( N \) pushes of the bottom of the tree.

The number of nodes visited for each algorithm was compared for three complex scenes on an ATI X1900XTX GPU. Push-Down visited 3 - 22% fewer nodes than kd-restart and the addition of short-stack reduced visits by a further 48 - 52%.

Stack-spilling Novák [64] presented another method for alleviating the impact of restarting called stack-spilling. Instead of having just a short-stack, another long-stack is added in local memory and a spilling technique is used to deal with overflows and underflows of the short-stack. If the short-stack is full and a new item is being pushed on top, the algorithm spills the bottom item and
Figure 3.4: Examples of stack spilling on stack overflow and underflow.

Stack spilling was shown to be on average 23% faster than short-stack with push-down on a GeForce 285 GTX GPU.

3.4.2 SIMD/MIMD Ray Tracing

Varying approaches have been developed to accelerate ray tracing on SIMD and MIMD architectures, and are commonly used on the wide SIMD or MIMD units available on many modern hardware architectures. This section outlines the varying ray tracing techniques used for efficient acceleration on SIMD and MIMD architectures.

Packet tracing When tracing coherent rays, that is, rays that share a similar origin and direction, high SIMD utilization can be easily achieved through techniques such as packet tracing [13, 66]. Packet tracing uses a single traversal stack and performs the same node/triangle intersection tests for multiple rays, forcing all rays to follow the same traversal sequence by all descending into a subtree if even only a single ray needs to traverse it, using masks to track which rays are active during each test. However, packet tracing performance degrades poorly when divergence becomes significant, where few SIMD lanes are active per test, making the technique unsuitable for secondary rays due to their incoherence. The platform presented in Chapter 4 initially favours neither coherent or incoherent rays, however this technique may be explored later if coherent ray performance need be optimized.
**n-ary bounding volume hierarchy** An alternative method of parallelism, instead of testing multiple rays against a single node or triangle, is to test a single ray against multiple nodes or triangles. In this approach, a bounding volume hierarchy (BVH) is used with a branching factor and leaf size equal to the SIMD width (an MBVH acceleration structure) [27, 28, 92]. This approach uses $M$-wide SIMD to perform $M$ node or triangle intersection tests in parallel for a single ray and does not rely on ray coherence at all. However, this approach quickly loses algorithmic efficiency for branching factors greater than four and with branching factors of 16 or greater is significantly worse than packet tracing if there is even a small amount of ray coherence [92]. Because of inefficiencies above a branching factor of four, and as the typical SIMD width of modern processors is four (in terms of single precision floating-point numbers), MBVH is typically implemented with a width of four and called QBVH (Quad-BVH), however the concept is not limited to this width. This approach is implemented in the platform presented in Chapter 4 for programmable values of $M$.

**Comparing simulated maximum performance to measured performance** Aila and Laine [1] compare GPU traversal and intersection acceleration performance using real measured values versus the simulated theoretical maximum to find inefficiencies causing discrepancies. Their simulator first dumps the sequence of traversal, leaf, and triangle intersection operations required for each ray. Knowing, from native assembly code, the sequence of instructions that need to be issued for each operation their simulator can then run the ray dump with a given SIMD width and report SIMD efficiency and estimated execution speed. Their results identified inefficiencies in hardware work distribution methods because of their optimisation for homogeneous units of work. Unfortunately, in ray tracing, the execution time of individual rays varies wildly and causes starvation issues when long running rays keep the distribution unit hostage. Aila and Laine proposed using persistent threads instead of hardware work distribution methods, bringing performance within 10-20% of the theoretical maximum for the given GPU. After this improvement, memory bandwidth begins to affect performance. This research shows that memory bandwidth is clearly an area that special attention must be paid, and an application specific cache architecture may be of some benefit.
Hybrid Ray Tracing for MIMD Architectures

As rays are independent of each other, the single ray multiple box or triangle method can be parallelised independently. On a MIMD architecture each ray is treated as a separate thread, meaning improved hardware utilization when processing incoherent rays. This approach is used in several other papers [49, 51, 60, 80] and is the approach implemented in the platform presented in Chapter 4.

Multiple threads of incoherent memory accesses are challenging for memory systems and are the cause of significant performance degradation for MIMD implementations on GPUs as rays diverge [1]. Benthin et al. [10] present a hybrid system that combines packet and single ray tracing for wide SIMD pipelines on the Intel MIC architecture. Their approach begins using packet tracing and falls back to single ray tracing when rays diverge. SIMD efficiency for single ray tracing is also improved by exploiting four-wide SIMD in each box and primitive intersection test, expanding to 16-wide SIMD by always performing four such node or primitive tests in parallel. Their results show greater performance than single or packet ray tracing for both coherent and incoherent rays. This research shows the abundance of parallelism that exists in ray tracing, and that ray tracing architectures must be designed to exploit many aspects of parallelism at a time, which is something FPGAs do well and can exploit.

3.4.3 Ray Tracing on FPGAs

FPGAs have also been considered for ray tracing. Schmittler and Woop from the computer graphics laboratory at Saarland University presented three FPGA based ray tracing prototypes, which implement the entire ray tracing process, including shading, using replicable SIMD shading and traversal processing units. The first prototype was a fixed function ray tracer called SaarCOR [77, 78]. The second was a programmable ray processing unit (RPU), which allowed for higher quality in the rendered image at the cost of area or performance [99]. It was implemented on a Xilinx Virtex-2 FPGA and performed similarly to a CPU. The third was an optimization of the RPU for dynamic scenes [98] which was implemented on an Xilinx Virtex-4 FPGA. The FPGA version outperforms the software implementation by 40% to 70%.
Cameron [15] presented a method for using FPGAs to supplement ray-tracing computations on the Cray XD-1. Preliminary simulations estimated the system’s performance to be $1.2 \times 10^7$ ray-triangle intersection tests per second.

Fender and Rose [29] presented a Virtex-2 based ray-tracing prototype that outperformed a 2GHz Pentium 4 processor by an order of magnitude. The prototype was capable of processing $5 \times 10^7$ ray-triangle intersection tests per second, however the implementation of the ray-triangle intersection data-path is not detailed.

Nery et al. [63] presented a parallel ray-tracing architecture for application-specific hardware based on a uniform spatial subdivision of the scene and exploiting an embedded computation of ray-triangle intersections. The architecture was implemented on both FPGA and General Purpose GPU systems. The FPGA implementation achieved $1.6 \times 10^5$ ray-triangle intersection tests per second while the GPU system achieved $4.0 \times 10^6$ ray-triangle intersection tests per second.

All of the above mentioned FPGA research aims to accelerate ray-triangle intersection alone, which is investigated in Chapter 5.

Wei and Gen [95] presented the acceleration of ray generation and acceleration hierarchy generation on FPGAs with limited resources. These aspects are not considered for acceleration in this thesis as we first focus on traversal acceleration, however this work may become relevant if this part of the ray tracing process begins to consume significant time, or the entire rendering process is to be implemented on a single chip.

Lee [50, 79] presented a mobile ray tracing system using the Samsung Reconfigurable Processor, implementing intersection tests and hierarchy traversal, as done with the presented platform in Chapter 4. The architecture was implemented on eight Xilinx Virtex-6 LX760 FPGAs each with dual-channel DDR2 RAM. The implementation traced $3.2 \times 10^6$ primary rays per second on a $1.7 \times 10^5$ triangle scene.

Nah et al. [59] presented RayCore, a ray-tracing architecture for mobile devices. The architecture focuses on ray-tracing on mobile devices, with scene sizes limited to $6.4 \times 10^4$ elements, but shows consistent performance over all scenes tested, up to $2.7 \times 10^7$ rays per second.

Few of the above mentioned implementations integrate with high-quality rendering software suitable for use with real industrial renderers. This is addressed by the platform presented in Chapter 4.
by implementing the platform with the same interface as a GPU accelerator and integrating it with LuxRender. The above research uses ray tracing techniques mostly borrowed from GPUs. This thesis discusses optimisations specific to FPGAs (e.g. caches, flexible acceleration hierarchies and priority queue implementation).

3.4.4 Caches for Ray Tracing

Very little cache specific ray tracing work has been published. The only directly related paper being the work of Nah et al. [61] but few details are published. They propose a split node cache, storing upper levels of scene nodes in a separate cache to exploit their temporal locality. Their method uses a breadth-first memory layout for the upper portion of the tree to be cached and then a depth-first layout below that. Their simulations showed a reduction in cache miss rate by up to three percent, but the tested scenes were very small and of similar size making it hard to draw general conclusions. The little-known requirements for efficient ray tracing caching are investigated and explored in Chapter 7.
With the amount of inherent and exploitable parallelism in ray-tracing and the flexible and parallel nature of FPGAs with lower power requirements, it is surprising that not more research has been done into the use of FPGAs for ray-tracing. One possible reason for this is the high barrier to entry into making an FPGA design. It has long been visible that there is a large productivity gap growing between the exponentially increasing amount of logic found on FPGAs and the number of man-hours taken to develop applications for them [88]. In this chapter, a flexible ray-tracing platform for FPGAs is presented which aims to reduce this gap by allowing easy configuration, development, testing and evaluation of ray-tracing concepts without having to design or redesign the application each time.

Ray-tracing is typically implemented for GPUs and CPUs using the Open Computing Language (OpenCL), which is a framework for writing programs that execute across heterogeneous platforms. OpenCL views a computing system as consisting of a number of compute devices, which might be CPUs or GPUs, attached to a host processor (a CPU). It defines a C-like language for writing programs, called kernels, that execute on the compute devices. A single compute device typically consists of many individual processing elements (PEs) and a single kernel execution can run on all or many of the PEs in parallel.

An OpenCL kernel is quickly compiled in seconds before being run on a PE, allowing easy development of parallel applications and a fast debug cycle. The same is not true for FPGA development, where a single change to a line of code requires reimplementation of the entire system at a cost of many hours to days for larger FPGAs [88]. ModelSim is a program that allows simulation and debugging of an FPGA system or individual components within it without implementing and debugging the required configuration on the physical FPGA and allowing visibility of all the signals within the design. Creating test benches for large, complete simulations is also time consuming, but is still quicker and easier than debugging
on an FPGA, where integrated logic analysers (ILAs) such as Xilinx’s ChipScope use extra logic, are unable to view all signals at once and can be tricky to set-up to trigger for complicated debugging tasks [101].

This chapter first presents the goals of an FPGA platform to solve the above problems in Section 4.1. The design of how the hardware and software components of the platform integrate and interact with each other is presented in Section 4.2 and the implementation and work-flow for development and debugging of the platform is presented in Section 4.3. The overall evaluation of the platform performance and scalability is presented later in Chapter 8, after:
1) intersection pipelines are presented in Chapter 5, 2) scalability of priority queues required for traversal are investigated in Chapter 6 and 3) a ray tracing specific cache hierarchy is investigated in Chapter 7.

4.1 GOALS

With the previously mentioned factors that hinder ray tracing development on FPGAs taken into consideration, the ray-tracing platform was carefully designed to smartly exploit parallelism in ray-tracing with the following goals:

- **having the same API as GPU platforms to aid software integration,**
  _i.e._ having two callable functions with the same arguments as the GPU API (1) to transfer the scene to the device and (2) to trace a bundle of rays on the device.

- **integrating with a modern, acceleration-focused and currently developed rendering engine,**
  _i.e._ enabling easy hardware integration, using modern acceleration techniques and allowing easy evaluation and comparison of performance with GPUs and CPUs.

- **easily generating complete VHDL or Verilog test benches for the entire system or individual components using real ray-tracing data,**
  _i.e._ easing simulation use by being able to write simple templates that generate multiple thousand line test benches in an easy templating language.
• being agnostic to different acceleration hierarchies,
  \textit{i.e.} to traverse both $k$D-tree and QBVHs without the need for
  reimplementation - the only difference between the two being
  their layout in memory.

• \textbf{traversing programmable widths of acceleration hierarchies},
  \textit{i.e.} varying $M$ for MBVH or increasing the maximum number
  of primitives per leaf to evaluate memory requirements and
  performance.

• \textbf{traversing multiple rays at a time and intersecting each ray
  with multiple nodes}, and
  \textit{i.e.} using a MIMD architecture with no assumptions about ray
  coherency to be efficient for both coherent and incoherent rays.

• \textbf{being modular where units need to be replicated or
  interchanged.}
  \textit{i.e.} the module that preforms traversal of the acceleration
  hierarchy can be easily interchanged to explore the performance
  of different techniques (\textit{i.e.} $kd$-restart etc.) or replicated to
  explore the efficiency of parallelisation. Priority-queues within
  the traversal unit may also be interchanged to explore their
  utilisation and replicability versus their performance.

4.2 \textsc{Design}

This section outlines the design of how the hardware and software
components of the platform integrate and interact with each other to
achieve the goals presented in Section 4.1.

4.2.1 Acceleration Platform

The target acceleration platform of this thesis is a PCI\textit{Express} attached
FPGA with on-board memory - the same interface used for GPU
accelerators - shown in Figure 4.1. The FPGA acts as a coprocessor
during ray-tracing the same way that a GPU does, with the CPU
handing off bundles of rays to be traced, leaving it free for other
computation and interrupted once the coprocessor has finished
tracing the rays. The development board’s on-board memory is used
to store scene primitives and the acceleration hierarchy, making it
available at lower latency and allowing the data to stream directly into the ray tracing computation pipelines.

The FPGA platform will be integrated with LuxRays - a subsection of the LuxRender suite dedicated to hardware acceleration. LuxRays is an open-source mixed C++ and OpenCL library that implements ray tracing for both QBVH and kD-tree acceleration hierarchies and has an OpenCL implementation that enables ray-tracing on CPUs and GPUs that support version 1.1 or newer of OpenCL.

4.2.2 Platform components

If we consider tracing a single ray through an acceleration hierarchy as described earlier in Chapter 2, there are three things that must be implemented:

- the control flow to determine when to continue or end traversal of the acceleration hierarchy for a given ray,
- a priority-queue to store intermediary results when more than one child node needs to be traversed, and
- a method to determine ray-box and ray-triangle intersection.

On the platform, shown in Figure 4.2, these functions are split into separate modular units to allow easy replicability, replacement and individual testing.
The control flow and priority-queue for each ray is combined into a traversal unit (TU). The TU is programmably replicated in the traversal controller (TC), detailed in Section 4.2.4, which distributes batches of rays from the host amongst the TUs during operation. Each TU traces a single ray at a time which results in a single triangle hit (or miss) result which the TC sends back to the host. When a TU is tracing a ray it needs to determine if the ray intersects with nodes and triangles in the acceleration hierarchy. This is done by sending a request to the intersection unit (IU), described in Section 4.2.5, which is responsible for performing box and triangle intersection tests. The IU returns hit (or miss) results for each of the child nodes if the requested node is internal or triangles if the requested node is a leaf, after which the TU continues with traversal. Supplementary units are required to interface with PCIe to co-ordinate the flow of nodes, primitives, rays and hits to and from the host computer using DMA, handled in the communication controller (CC) detailed in Section 4.2.3, and to interface with the on-board memory to store nodes and primitives and to access them later for intersection tests, handled in the memory controller (MC) described in Section 4.2.6.
4.2.3 Communication Controller

The communication controller (CC) is responsible for interfacing with the PCIe\textsuperscript{express} bus and:

- maintaining memory mapped configuration registers that control the operation of the platform from the software on the host computer,
- implementing a DMA controller to make transfers to and from the memory on the host computer via the PCIe\textsuperscript{express} bus with control passed from the host computer CPU to the FPGA,
- completing the transfer of nodes, primitives and rays to the FPGA using DMA when requested by the host computer,
- transferring hits back to the host computer once tracing of each ray has completed using DMA and
- generating interrupts to send to the host computer when each DMA operation has completed.

A \textit{node} is an internal node in the acceleration hierarchy, a \textit{primitive} is a scene triangle and a \textit{leaf} is a node without any children, containing a reference to one or more primitives. A node is of a predetermined width (set by the user before construction) and contains a reference to and bounding box of all children nodes or primitives. A primitive is represented by three points in three-dimensional space. A hit contains the resulting primitive hit or miss for a given ray.

\textbf{Listing 4.1:} Pseudocode explaining the transfer process in relation to computation on the host.

```
1 def RayTraceSceneOnFPGA(scene):
  hierarchy = ConstructAccelerationHierarchyOnHost(scene)
  SendAccelerationHierarchyFromHostToFPGA(hierarchy)

  ray_packets = GenerateRayPacketsOnHost(scene)
  for ray_packet in ray_packets:
    SendRayPacketFromHostToFPGA(ray_packet)
    // The FPGA traces the rays through the hierarchy,
    // writes the hit/miss results back to the host
    // memory, and fires an interrupt when complete.
    WaitForFPGAInterrupt()
  RenderResults(ray_packet)
```
Once the host computer software has constructed the acceleration hierarchy it initiates traversal by writing the memory locations and numbers of the nodes and primitives to the configuration registers in the CC and by signalling the CC to start by writing to a control register. The CC then transmits the appropriate PCIe express read request packets to the host computer memory management unit (MMU) which replies with the requested node and primitive data.

The CC forwards any node and primitive completion data to the memory controller to be written to the FPGA development board memory. Once complete, an interrupt is sent to the host computer and the host software initiates traversal of a set of rays. The host computer writes the address and number of the rays to be intersected and an address to write the results to the configuration registers in the CC and signals the CC to initiate a DMA transfer by writing to a control register. The CC then transmits the appropriate PCIe express read request packets to the host computer MMU which replies with the requested ray data. The CC forwards the ray data to the traversal controller and waits for it to return hits. When hits arrive the CC encapsulates them as PCIe express write packets and transmits them to the host computer MMU.

An example of the PCIe express requests required to transfer the Cornell Box scene, described earlier in Section 2.3, are shown in Figure 4.3. Following transaction sequence numbers, shown at the origin of the transaction in red:

- Transactions 1–4 set the node memory address (0x80000000), node buffer size (0x580), primitive memory address (0x88000000) and primitive buffer size (0x800). The node and primitive buffer sizes represent their size in bytes and are determined by the scene. Sizes of high quality scenes are typically in the order of hundreds of megabytes.

- Transaction 5 signals the DMA controller on the FPGA to start the transfer of the scene.

- Transactions 6–8 show the memory read requests for the scene nodes, from the node memory address (0x80000000) and totalling the size of the node buffer (0x580).

- Transactions 9–11 shows the completion packets containing node data from the previous memory reads.
Figure 4.3: PCI express transactions to transfer the scene nodes and primitives to the development board from the host computer using DMA. The number of each transaction in the sequence is shown at the origin of the transaction in red.

- Transactions 12–15 show the memory read requests for the scene primitives, from the primitive memory address (0x88000000) and totalling the size of the primitive buffer (0x800).

- Transactions 16–19 shows the completion packets containing primitive data from the previous memory reads.
Host Computer | Development Board
---|---
1 |
2 | MWr | RAY_ADDR | 0x98000000
3 | MWr | RAY_SIZE | 0x4
4 | MWr | HIT_ADDR | 0x98800000
5 | MWr | RT_INITIATOR | 0x1
6 | MRd | 0x98800000 | 0x30
7 | CPD | 0x30 | RAY_DATA
8 | MWr | 0x98800014 | HIT_DATA_1
9 | MWr | 0x98800000 | HIT_DATA_0
10 | MWr | 0x98800028 | HIT_DATA_2
11 | MWr | 0x9880003C | HIT_DATA_3

**Figure 4.4:** PCI express transactions to transfer a bundle of four rays to the development board from the host computer and for the development board to write the four hits found during ray tracing to the host computer via DMA. The number of each transaction in the sequence is shown at the origin of the transaction in red.

- Once all the required scene data has been read from host memory and stored on the FPGA, transaction 20 signals completion to the host software with an Message Signalled Interrupt (MSI).

An example of the PCI express requests required to trace four rays for the Cornell Box scene, described earlier in Section 2.3, are shown in Figure 4.4. Following transaction sequence numbers, shown at the origin of the transaction in red:

- Transactions 1–3 set the ray memory address (0x98000000), number of rays (0x4), hit memory address (0x98800000). Note that the number of hits does not need to be set as it is the same as the number of rays. Typically, several thousand rays are transferred per transaction and the total size of the data transferred is the number of rays multiplied by the size of a ray: six floating-point numbers plus a 32 bit ray identifier ($r \times (6 \times 32 + 32)$).
• Transaction 4 signals the DMA controller on the FPGA to start the transfer of rays.

• Transaction 5 shows the memory read requests for the rays, from the node memory address (0x98000000) and totalling the number of rays times the size of a ray (0x30).

• Transaction 6 shows the completion packet containing ray data from the previous memory read. After this packet, the platform begins tracing the transferred rays through the scene in the platform onboard memory.

• Transactions 7–10 show the memory write packets for the ray hits, to the hit memory address (0x98800000) plus an offset calculated by the hit index multiplied by the size of a hit (0x14).

• Once all the required rays have been traced and hits written to the host memory, transaction 11 signals completion to the host software with an MSI.

The steps of reading rays and writing results can be repeated as many times as necessary for different sets of rays for the same scene, without having to resend the nodes and primitives each time.

### 4.2.4 Traversal Controller

The traversal controller (TC) buffers rays received from the CC when they are available and removes a ray from the buffer to be stored and traced by a traversal unit (TU) as they become available (by signalling completed traversal of a previous ray). A TU is a programmably replicable module that implements the control flow and intermediary result storage for acceleration hierarchy traversal, described earlier in Algorithm 2.1. To maximise the throughput of the deep ray-object intersection pipelines, described later in Chapter 5, more than just the single intersection test a single traversal unit can request at a time needs to be processed. As the traversals of different rays are independent of each other, this unit is able to be freely replicated when configuring the platform and being able to process multiple rays at once is one of the big sources of parallelism in ray-tracing. To process multiple rays at once, instead of replicating the entire platform as was done in previous work, only the traversal unit is replicated to allow the intersection pipelines to be shared - increasing
their efficiency - and reducing wasted resource utilisation, allowing it to be used for further performance gains.

When a TU receives a ray from the TC it starts traversal from the root node of the acceleration hierarchy. For each node, the TU makes a request to the intersection unit to determine if the ray intersects with any of the nodes’ children. The intersection unit returns hit/miss results for each child node. The traversal unit stores references to intersected children in a priority-queue with the closest intersection having highest priority. Efficient implementation of the priority queue in the TU is of great importance to its replicability and is discussed in more detail in Chapter 6. Traversal continues down a tree until a leaf node (containing triangles) is encountered. RTI tests for all triangles within the leaf are requested from the intersection unit which responds with a hit or miss flag and the distance to intersection along the ray if intersection has occurred. If intersection occurs, the closest intersection is guaranteed to be the first intersection along the ray and traversal terminates, freeing the TU to process the next available ray and discarding any remaining intermediary nodes in the priority queue without processing them. If no intersection is found, a node is popped from the queue and traversal continues. If the priority-queue is empty then traversal terminates without intersection.

4.2.5 Intersection Unit

The intersection unit (IU) buffers intersection requests from the traversal units when they are available and passes them onto the memory controller when requested. As there are multiple traversal units which each make intersection requests asynchronously to each other, it is possible that multiple intersection requests can be received simultaneously and yet only one can be forwarded to the memory controller per cycle. To solve this, the IU uses a fixed-priority scheduler to determine what intersection request to process and any traversal unit that does not have its pending request processed in the given cycle must stall and hold the request signal high until it is processed and acknowledged by the IU. The fixed-priority scheduler may starve some traversal units of intersection requests while they stall, but as all rays are independent, during normal operation it does not matter what order they are processed in. All requests will be processed eventually, at latest when the higher priority traversal units...
have completed their work. The objective of multiple traversal units is to keep the IU busy, for performance it does not matter in which order rays are processed.

In response to an intersection request, the memory controller streams the nodes or primitives for the request to the IU. The IU implements two separate, deep pipelines for ray-box and ray-triangle intersection - into which the responses from the memory controller are fed. These pipelines, outlined in more detail in Chapter 5, were designed by analysing the data-flow of Equations (2.7) – (2.9c) for ray-triangle intersection and Equations (2.10a) – (2.10b) for ray-box intersection to exploit the parallelism in their many vector calculations.

When the input data rate to the intersection pipelines exceeds their throughput, replicating the pipelines, reducing precision of calculations or using fixed-precision calculations may be explored with the flexibility the platform and the FPGA provide to increase throughput. The latency of the pipelines, although not immediately important, must also not be neglected, as reducing the latency could potentially increase performance if logic utilization limits the number of traversal units that can be created.

4.2.6 Memory Controller

The memory controller (MC) is responsible for interfacing with the platform on-board memory. At the beginning of ray tracing when the scene is transferred to the platform, described previously in Section 4.2.3, the memory controller (MC) accepts the scene nodes and primitives from the CC and writes them to consecutive addresses in local memory. During ray tracing, intersection requests are taken from the IU and translated into the appropriate local memory read requests for the nodes or primitives they relate to. When the requested data is returned from the on-board memory it is streamed to the appropriate IU intersection pipeline.

Creating this module keeps the memory interface abstracted from the other modules who just pass node and primitive data and requests to the module, which means that swapping any vendor specific memory interface controllers (i.e. when upgrading to a newer board or moving to a different vendor) does not effect the rest of the platform, just the small amount of logic that interfaces with it to create the memory access addresses. As the nodes and primitives
are stored consecutively in memory, the memory access address and access length of a node or primitive reference is calculated by multiplying the reference value by a constant based on the address and data width of the on-board memory interface. Changing the memory interface type between different vendors means changing these values, but the method of translating a node or primitive reference into a memory address is still a simple multiplication.

### 4.2.7 Example for Cornell Box

This section will walk through tracing an example ray for the Cornell Box scene, presented earlier in Section 2.3, to show the flow of requests between all of the components in the platform. Figure 4.5 shows the QBVH acceleration hierarchy created for the Cornell Box scene. Internal nodes are shown in grey, leaves in blue and empty leaves in red.

The nodes and primitives are transferred to the FPGA on-board memory as described previously for Figure 4.3. Then a bundle of rays is transferred to the CC as described previously for Figure 4.4.

Figure 4.6 shows the flow of requests between the platform components when tracing a ray. The flow of requests that then take place are as follow:

- A single ray from the bundle (RAY 0) is assigned to TU0 by the CC.
The TU begins by requesting intersection from the IC for the root node.

The IC requests the data for the root node from the MC.

The MC translates the root node data request from the IC into a memory read address and presents the address to the platform specific memory interface. The memory interface returns the requested data and the MC passes it through to the IC.

The IC performs intersection tests with the root node and RAY 0 and returns the results to the TU. In this example the ray only intersects with the bounding-box surrounding the leaf with reference 1E.

The TU then requests intersection from the IC for leaf 1E.

The IC requests the data for leaf 1E from the MC.

The MC translates the leaf 1E data request from the IC into a memory read address and presents the address to the platform specific memory interface. The memory interface returns the requested data and the MC passes it through to the IC.

The IC performs intersection tests with the primitives within leaf 1E and RAY 0 and returns the results to the TU.

The TU then returns the closest of the triangle intersection results, if there are any, to the CC as the resulting “hit” for RAY 0. After this step TU0 is now ready to begin traversal for another ray.
This very simple example only tests for intersection with a single node and a single leaf, but shows the complete flow of requests between all components.

4.3 IMPLEMENTATION

This section outlines the hardware and software components required to implement the platform and the workflow for development and debugging.

4.3.1 Hardware components

The implementation platform used for this thesis is a HiTech Global Xilinx Virtex-5 PCIe Development Board [35] in an Intel Core i5 host computer running Ubuntu 14.04 Linux. The board contains a Xilinx Virtex-5 LX330T-2 FPGA with 192 DSP48E slices, 331,776 logic cells and 11,664 Kbits of block RAM. Connected to the FPGA is a 200-pin 2GB DDR2 SO-DIMM. The FPGA connects to the host through an 8-lane PCIe 1.1 bus capable of 16Gb/s end-to-end bandwidth.

4.3.2 Traversal unit

The traversal unit is implemented in VHDL and consists of control logic to implement acceleration hierarchy traversal, as described in Algorithm 2.1, and instantiation of a priority queue to store intermediary results.

The priority queue’s pq_out output vector shows the current item at the head (with highest priority) of the priority queue. The pop operation of the priority queue is triggered by driving the pq_rd signal high for a cycle, which removes the item from the top of the queue and outputs the next highest priority item to pq_out. The push operation of the priority queue is triggered by driving the pq_wr signal high while presenting a new item to the pq_in input vector for a cycle, which inserts the item onto the queue and positions it based on its priority. Driving the pq_reset signal resets the queue, removing any items currently stored. More operation and implementation details of the priority queue push and pop operations are given later in Chapter 6.
REST
Wait for a ray to begin traversal.
Outputs:
ray_wait = 1
int_req_ref = root

INT_ACK_WAIT
Wait for the IC to acknowledge the request.
Outputs:
int_req = 1

INT_RESULT_WAIT
Wait for the intersection results from the IC.
Outputs:
pq_in = int_result
pq_wr = 1

CHECK_DONE
Check to see if traversal of the tree has completed.

SEND_RESULT
Send the result to the CC.
Outputs:
hit_ready = 1

INT_REQ
pop a value from the PQ to intersect.
Outputs:
int_req_ref = pq_out
pq_rd = 1

hit > pq_out
hit ≤ pq_out

result_ack = 1
ray_ready = 1
int_req_ack = 1

Figure 4.7: The traversal unit FSM that implements priority-queue based ray-tracing from Algorithm 2.1.
Figure 4.7 shows the finite-state machine (FSM) that controls the input and output values on the priority queue to implement hierarchy traversal. The TU enters the REST state upon reset where it waits for the TC to assign it a ray. Once a ray has been assigned, the TU presents the root node reference to the IC for intersection with the current ray before transitioning to the INT_ACK_WAIT state. In this state, the TU waits for the IC to acknowledge the intersection request made in the previous state. Once acknowledgement is received the TU transitions to the INTRESULT_WAIT state, where it waits for the IC to return intersection results for the previously requested reference. Any successful intersection results received are written to the queue using the pq_in and pq_wr signals. Once the last intersection result is received, shown by the int_res_last signal, the TU transitions to the CHECK_DONE state. In this state, the distance to the current hit (which is initially set to infinity) is compared with the distance to the intersection of the item at the head of the priority queue (which is also infinity if the queue is empty). If the queue is empty, or the distance to the item on the top of the queue is farther than the distance to the current intersection, the TU transitions to the SEND_RESULT state, where the traversal result passed to the TC, before emptying the queue (using the pq_reset signal) and transitioning back to the REST state, ready to begin traversal for another ray. If the queue is not empty and the distance to the item on the top of the queue is closer than the distance to the current intersection, the TU transitions to the INT_REQ state, where it presents the item on the top of the queue to the IU for intersection, pops the value using the pq_rd signal, then transitions to the INT_ACK_WAIT state to continue traversal.

4.3.3 Main platform components

The four platform components are written in a mixture of VHDL and Verilog hardware description languages.

- The CC is implemented from the Xilinx Endpoint Block Plus PCIeExpress core and a modified Bus Master DMA reference design in Verilog [100].

- The MC interfaces with the platforms on-board DDR2 using a Xilinx MIG controller in Verilog [103]. The MIG interface
presents address and data signals for performing memory reads and writes and a 64 bit wide data bus to output read data.

- The IU is written in VHDL and uses Xilinx DSP floating-point cores with the maximum available pipeline length for all floating-point calculations and comparisons, with the maximum available pipeline length sacrificing latency but enabling them to run at higher frequency [102]. More details of the IU implementation are given later in Chapter 5.

- The TC wraps multiple TUs, detailed previously in Section 4.3.2, using VHDL and uses FIFOs for buffering rays as they arrive from the CC and hits as they wait to be sent back to the host by the CC.

The MC, IU and TC run with a 150MHz clock frequency, a frequency that allows easy scalability with timing closure, while the CC runs with a 250MHz clock, required to interface with the Xilinx PCIexpress core. It is possible to clock the MC, IU and TC at higher clock frequencies but manual layout work would be required to meet timing when the FPGA approaches capacity. The clocks are generated from a 100MHz reference clock with one of the FPGA’s PLLs. FIFOs and BlockRAMs are generated using the Xilinx Core Generator [104] where required to buffer data or cross clock domains.

4.3.4 Data representation

All node and primitive references are encoded using 32 bit unsigned integers. The most significant bit (bit 31) is used to differentiate between nodes and primitives, with 1 representing a primitive and 0 representing a node. All objects must be arranged consecutively in memory during tree construction.

For a node, the remaining 31 bits (bits 30 down to 0) represent the node address, shown in Figure 4.8. The root node has the address 0x00000000 and is the first node placed on the priority-queue in a traversal unit when traversal starts for a ray.

For leaves, bits 30 to 27 represent the number of primitives in the leaf and the remaining 27 bits (bits 26 down to 0) represent the
starting address of the first primitive, shown in Figure 4.9. This means a leaf can reference a maximum of 16 primitives and the tree must continue splitting the primitives until this constraint is fulfilled.

A node contains a reference to and bounding box of all children nodes or primitives. Each child referenced therefore requires six single-precision floating-point numbers, defining the bounding-box minimum and maximum values in three dimensions, and a reference. This makes a single node in a MBVH require $M \times (7 \times 32)$ bits of memory, which means 896 bits ($4 \times (7 \times 32)$) for a QBVH node. Each primitive (triangle) in a leaf is represented by three single-precision floating-point points in three-dimensions, meaning 288 bits ($9 \times 32$) of memory per primitive and allowing each leaf to reference a maximum of 4,608 bits ($16 \times 288$) for sixteen primitives.

### 4.4 Platform Generation

Platform generation is implemented using Python and the Jinja2 templating library. Python is a widely used general-purpose, high-level programming language with a design philosophy that emphasizes code readability and with syntax that allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java [76]. Jinja2 is a modern and designer-friendly templating language for Python, which is fast, widely used and secure [75].

The directory layout of the platform generation scripts and templates is shown in Figure 4.10. The main script is the generator.py file, which when run:

- assembles configuration settings from the settings.py files,

- reads structured data files from the data directory,

- sets up a Jinja2 environment,

- adds helper functions that generate VHDL and Verilog constructs,
Figure 4.10: The file structure of the platform generator.
• generates all files in the templates directory using the Jinja2 environment, and finally

• writes the generated files into a unique platform project folder in the build directory.

Using this program structure it is possible to:

• write template files in a mix of Jinja2 and either VHDL or Verilog that implement the functionality of the platform,

• use Jinja2 constructs such as for loops and if statements to write generic templates,

• change platform settings in the settings.py file that are passed into the Jinja2 environment and influence the output generated by the templates,

• use a TRAVERSE_UNITS setting to easily scale the number of traversal units and a TRAVERSE_UNIT_PQ_TYPE setting to easily change the type of queue within the traversal units generated by the platform,

• use a VENDOR setting to render different cores based on the target vendor (i.e. Altera or Xilinx specific PCIe, memory and floating-point cores), and

• dump real ray-tracing data from LuxRays into files in the data folder and use the data to generate full system or individual module test benches to verify the correctness of the platform.

Listing 4.2 shows an example of a templated VHDL file, using a Jinja2 for loop (lines 1 and 13) to instantiate as many traversal units as are specified by the TRAVERSE_UNITS setting. Template variables, such as the loop index i, can be printed in-place using double curly braces, as shown on lines 2, 4 and 9.

When combined with tools such as Xilinx’s ChipScope for FPGA debugging, the platform generator enables the work-flow shown in Figure 4.11 with easy configuration, full system verification and scalability over different platforms to help quickly make large designs and reduce the previously mentioned productivity gap.
Modify platform templates and configuration variables

Run Python script to generate platform output directory and test benches

Test individual components with component simulation test benches.

Test complete platform with ray-tracing simulation test bench.

Implement the platform on the FPGA.

Render real scenes on the FPGA using LuxRays.

Debug FPGA platform using ChipScope.

Figure 4.11: The work-flow for platform development.
Listing 4.2: Example of templated VHDL code.

```vhdl
{% for i in range(ctx.TRAVERSE_UNITS) %}
tu_{{i}} : traverse_unit
generic map (UNIT_ID => {{i}})
port map (clk => clk,
          rst => rst_i,
          traverse_ray_wait_o => traverse_ray_wait({{i}}),
          ...
          ...
);%
{% endfor %}
```

4.5 SOFTWARE INTEGRATION

This section outlines the Linux driver, written to provide a software interface to the platform, in Section 4.5.1 and the modifications to LuxRays to enable ray tracing on the platform in Section 4.5.2.

4.5.1 Operating-system integration

A Linux driver was implemented for this thesis that allows ray-tracing software to interact with the FPGA and was written using the Linux Kernel architecture for PCIexpress device drivers [24]. The driver creates functions for:

- transferring scene data to the device from main memory (TransferScene),
- initiating intersection of a bundle of rays (IntersectRays), and
- an interrupt handler to let the FPGA signal completion of any DMA operations (HandleInterrupt).

The TransferScene function accepts the respective address and quantity of the scene nodes and primitives, writes the values to the memory mapped registers of the CC and signals the FPGA to start transferring. The FPGA then reads the nodes and primitives from the main memory over the PCIexpress bus and stores them via the MC in the on-board memory.
The **IntersectRays** function accepts the address and quantity of the rays to be intersected and the address which the FPGA must write the resulting hits to. The function then writes these values to the memory mapped registers of the CC and signals the FPGA to start intersection. The FPGA then reads the rays from the main memory over the PCI*express* bus, sends them to the TC to start intersection, and writes the resulting hits from the TC over the PCI*express* bus to the main memory.

When either of the **TransferScene** or **IntersectRays** functions initiates a PCI*express* DMA transfer to or from the FPGA, the calling thread then sleeps, waiting on a semaphore to be re-awoken once the transfer has completed and freeing up the CPU for any other required computation. The completion of a transfer is signalled by the FPGA through a MSI, which triggers a call to the **Handle Interrupt** function. **HandleInterrupt** then wakes up the function that initiated the transfer using the semaphore the calling function is waiting on.

### 4.5.2 Integration with LuxRays

The core C++ classes that implement LuxRays functionality are accelerators and intersection devices. Accelerators, such as QBVH and *kD*-tree, all inherit from a base **Accelerator** class and are initialized (::Init()) by passing them the set of scene primitives, out of which they create their respective acceleration hierarchies and store them in memory. Intersection devices, such as native CPU threads and OpenCL GPUs, all inherit from a base **IntersectionDevice** class and are initialized (::Init()) by passing a data set from an **Accelerator** class. Once initialized, the **IntersectionDevice** class accepts ray bundles through calls to its ::PushRayBuffer() function, traces each ray on the given device with the given acceleration method, and returns the resulting hits through calls to its ::PopRayBuffer() function. These classes give a standard, modular interface between different acceleration techniques and devices, with each class only doing the minimum that it needs to.

Integration of the FPGA platform was completed with LuxRays by creating an **FPGAIntersectionDevice** class that inherits from the **IntersectionDevice** class. For this created class, when the ::Init() function is called, passing in the accelerator data set, the Linux driver **TransferScene** function is called, transferring the scene data
to the FPGA. Then, for each successive call to the ::PushRayBuffer() function, passing a ray bundle to be traced, the Linux driver IntersectRays function is called. The IntersectRays function transfers the ray bundle to the FPGA to be ray traced, and makes the resulting hits available to a call to ::PopRayBuffer() only once the device signals completion through the HandleInterrupt function.
In computing, a pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one. CPU instruction pipelines, such as the classic RISC pipeline, are a common computing pipeline that allow overlapping execution of multiple instructions with the same circuitry. The circuitry is usually divided up into stages, including instruction decoding, arithmetic, and register fetching stages, with each stage partially processing one instruction at a time. “Pipeline registers” are inserted in-between these pipeline stages, and are clocked synchronously. The time between each clock signal is set to be greater than the longest delay between pipeline stages, so that when the registers are clocked, the data that is written to them is the final result of the previous stage. This means the maximum frequency the pipeline can operate at is determined by the slowest pipeline stage. The maximum frequency also equates to the throughput of such a system, meaning a pipeline with a higher maximum frequency will have a higher throughput.

The throughput of a system is the rate at which it can process data. On a general purpose processor, where a single instruction is processed every cycle, and an application is made up of a set of instructions, the throughput of the given application is the frequency of the processor divided by the average number of instructions required to complete the application. An application specific hardware pipeline can be thought of like a general purpose processor, that only implements a single instruction, which calculates the output of the entire application with the single instruction.

Two basic requirements for ray-tracing implementation are methods to determine if a ray intersects with a bounding box, for acceleration hierarchy traversal, and to determine if a ray intersects with one of the scene primitives. Implementations of these functions for general purpose processors, such as C++ implementations for CPUs or OpenCL implementations for GPUs, execute sequentially and exit early from the function as soon as it is known that an intersection will not occur to reduce the average
number of cycles required, increasing throughput. A pipelined implementation of these intersection algorithms allows concurrent execution of sequential intersection tests, increasing the throughput of the implementation.

Though other FPGA implementations appear to use pipelined designs \[29, 78, 99\], very few implementation details are given, making them hard to replicate or compare. This chapter outlines in detail the steps taken to implement floating-point intersection pipelines for ray-triangle and ray-box intersection, providing a reproducible and comparable reference implementation for FPGAs.

First, to create the ray-triangle intersection pipeline, a common C++ implementation of the Möller and Trumbore algorithm for ray-triangle intersection is presented in Section 5.1.1, for familiarity and easy comparison. In Section 5.1.2 the data-flow of this implementation is analysed and the implementation of required vector operations and the overall pipeline using Xilinx floating-point modules is presented. The utilization and relative performance of the completed pipeline is then evaluated in Section 5.1.3. Second, to create the ray-box intersection pipeline, the same process is presented as was for ray-triangle intersection. A common C++ implementation is presented in Section 5.2.1, its data-flow is analysed and used to create a pipelined implementation in Section 5.2.2, and its utilization and performance is evaluated in Section 5.2.3.

5.1 RAY-TRIANGLE INTERSECTION

This section outlines the design and evaluation of the ray-triangle intersection pipeline. First, the software implementation in Section 5.1.1 is translated into a data-flow diagram, showing the data dependency of the inputs and outputs of each calculation. Then, in Section 5.1.2, the data-flow diagram and required vector operations are analysed to produce a deep pipelined design, using Xilinx floating-point modules, to optimise throughput. Finally, in Section 5.1.3, the overall pipeline performance and logic utilization is evaluated.
5.1.1 Software Implementation

Listing 5.1 shows a common C++ implementation of the Möller and Trumbore algorithm for ray-triangle intersection [55], described previously in Section 2.1.1, solving Equation (2.7) and performing the tests in Equations (2.9a) – (2.9c) where needed to determine intersection. The algorithm is presented in C++ for familiarity and to provide easy comparison during evaluation.
Listing 5.1: Common ray-triangle intersection implementation in C++.

```cpp
bool TriangleIntersect(Vertex v0, Vertex v1, Vertex v2,
const Ray &ray, float *tHit) {
    Vector e1 = v1 - v0; // eq. 2.8a
    Vector e2 = v2 - v0; // eq. 2.8b
    Vector P = Cross(ray.d, e2); // eq. 2.8d

    // compute the divisor
    float divisor = Dot(P, e1);
    if (divisor == 0.0) return false;
    float invDivisor = 1 / divisor;

    // compute first barycentric coordinate (u)
    Vector T = ray.o - v0; // eq. 2.8c
    float u = Dot(T, P) * invDivisor;
    if (u < 0.0 || u > 1.0) // eq. 2.9a
        return false;

    // compute second barycentric coordinate (v)
    Vector Q = Cross(T, e1); // eq. 2.8e
    float v = Dot(ray.d, Q) * invDivisor;
    if (v < 0.0 || u + v > 1.0) // eq. 2.9b and 2.9c
        return false;

    // compute distance to intersection point (t)
    float t = Dot(e2, Q) * invDivisor;
    if (t < ray.tmin || t > ray.tmax)
        return false;

    *tHit = t;
    return true;
}
```

The implementation, after first computing the edge vectors in Equations (2.8a) – (2.8b) and dot-product in Equation (2.8d), computes the divisor ($P \cdot E_1$). If the divisor is zero (line 9), the triangle is degenerate and therefore cannot intersect the ray.

Next, the implementation computes the first barycentric coordinate, $u$, by completing Equation (2.8c) and multiplying the dot-product
\( \mathbf{P} \cdot \mathbf{T} \) by the inverted divisor. The test in Equation (2.9a) is then completed along with a test to determine if \( u \) is greater than one, allowing for an early exit, if either test fails, without completing the remaining calculations. This second test is an optimisation to determine the result of Equation (2.9c) without the value for \( v \), where, assuming \( v \) is greater than zero to pass Equation (2.9b), if \( u \) is greater than one there is no way that the test in Equation (2.9c) will pass.

The second barycentric coordinate, \( v \), is then computed by completing Equation (2.8e) and multiplying the dot-product \( \mathbf{Q} \cdot \mathbf{D} \) by the inverted divisor. The tests in Equations (2.9b) – (2.9c) can then be completed in their full form, exiting if either fail.

Finally, the distance to the intersection, \( t \), is calculated by multiplying the dot-product \( \mathbf{Q} \cdot \mathbf{E}_2 \) by the inverted divisor. This value is then compared against the minimum and maximum bounds of the ray. The minimum bound for the ray is set to the source of the ray (the position of the camera), so if the intersection occurs before this distance the triangle is behind the camera and will not contribute to the final image so is discarded. The maximum bound for the ray is initially set to infinity and is updated upon the first triangle intersection to the distance of the intersection and further updated when any closer intersections occur, ensuring only the closest triangle intersection is returned for the ray.

### 5.1.2 Pipeline Design

The data-flow of Listing 5.1 and Equations (2.7) – (2.9c) was analysed and the diagram in Figure 5.1 was produced, showing the data dependency of the input and output of each calculation. From this data-flow diagram it is evident that there is easily exploitable parallelism in the RTI implementation, and even more parallelism exists within the required vector operations.

The following sections outline the methods and tools used to design and implement the RTI pipeline. The following sections cover: the pipelined floating-point modules used, design and implementation of vector subtraction, cross-product and dot-product operations, an optimisation for early intersection determination and an overview of the complete pipeline.
Floating-point modules

As the target platform is a Xilinx Virtex-5 LX330T-2, all floating-point operators are implemented using the Xilinx floating-point modules [102]. The open-source, parametrisable floating-point library presented by Belanović and Leeser [9] was considered to provide a platform implementation without vendor dependencies. However, as other modules such as the PCIexpress core and DDR2 memory interface rely on vendor specific modules, and as the Xilinx floating-point modules are highly optimised for the target device, the
Xilinx floating-point modules proved to be a better fit. Use of these modules provides:

- multiplication, addition, subtraction, division and comparison operators;
- compliance with the IEEE-754 floating-point standard [7];
- parametrised fraction and exponent word-lengths (if required);
- use of DSP48E primitives to speed up multiplication, addition and subtraction; and,
- a synchronous pipelined design for all operations using a single clock.

The implementation of each required operation is generated as a single file with the maximum possible latency for the operation. Table 5.1 shows the logic utilization and estimated performance for the maximum possible latency of the given operation. Using the maximum possible latency (meaning the longest possible pipeline) uses more resources but ensures the pipeline can operate at a high frequency. The latency of the pipeline can be optimised later if it is found to be a limiting factor to performance.

If an alternate FPGA platform is used, i.e. an Altera development board, the Altera equivalent floating-point modules can be placed in the platform specific folder in the platform generator directory structure, shown previously in Figure 4.10 and described in Section 4.4.

**Vector operations**

This section details vector subtraction, cross product and dot product equations and their resulting pipeline designs implemented with the Xilinx floating-point cores.
Equation (5.1) shows the equation for vector subtraction and the resulting pipelined implementation is shown in Figure 5.2. The figure shows that all three subtractions can occur in parallel with a total latency of 11 cycles.

\[
\begin{bmatrix}
x_0 \\ x_1 \\ x_2
\end{bmatrix} - \begin{bmatrix}
y_0 \\ y_1 \\ y_2
\end{bmatrix} = \begin{bmatrix}
x_0 - y_0 \\ x_1 - y_1 \\ x_2 - y_2
\end{bmatrix} = \begin{bmatrix}
z_0 \\ z_1 \\ z_2
\end{bmatrix}
\]  

(5.1)

**Cross Product** Equation (5.2) shows the equation for vector cross-product calculation and the resulting pipelined implementation is shown in Figure 5.3. The figure shows that the six multiplications
Figure 5.4: Data-flow for vector dot product calculation.

can occur in parallel followed by the three subtractions in parallel, resulting in a total latency of 17 cycles \((6 + 11)\).

\[
\begin{bmatrix}
  x_0 \\
  x_1 \\
  x_2 
\end{bmatrix} \times \begin{bmatrix}
  y_0 \\
  y_1 \\
  y_2 
\end{bmatrix} = \begin{bmatrix}
  x_1 y_2 - x_2 y_1 \\
  x_2 y_0 - x_0 y_2 \\
  x_0 y_1 - x_1 y_0 
\end{bmatrix} = \begin{bmatrix}
  z_0 \\
  z_1 \\
  z_2 
\end{bmatrix}
\] (5.2)

**Dot Product** Equation (5.3) shows the equation for vector dot-product calculation and the resulting pipelined implementation is shown in Figure 5.4. The figure shows that the three multiplications can occur in parallel but the two additions must occur sequentially, requiring the output of one of the multiplications to be delayed while the first addition occurs. The total latency for the parallel multiplications followed by the two sequential additions is 28 cycles \((6 + 11 + 11)\). A three-input floating-point adder, such as that presented by Tenca [84], may be explored if reducing total pipeline latency is later required.

\[
\begin{bmatrix}
  x_0 \\
  x_1 \\
  x_2 
\end{bmatrix} \times \begin{bmatrix}
  y_0 \\
  y_1 \\
  y_2 
\end{bmatrix} = x_0 y_0 + x_1 y_1 + x_2 y_2 = z
\] (5.3)

**Determining intersection**

Determining a valid intersection requires satisfying the tests on the barycentric \(u\) and \(v\) coordinates given in Equations (2.9a) – (2.9c).
Table 5.2: Signs for operands in a division and the sign of the result. Values in brackets represent the binary value of the sign for the floating-point representation.

Looking at the data-flow diagram in Figure 5.1, the values of \( u \) and \( v \) are the very last to be calculated, meaning that the tests must wait until the end and add extra latency to the pipeline.

In this section an optimisation is presented that allows determining if a valid intersection has occurred without waiting for the complete calculation of \( u \) and \( v \). Using Equation (2.7) to solve for \( u \) and \( v \), the tests in Equations (2.9a) – (2.9c) can be rewritten as shown in Equations (5.4a) – (5.4c).

\[
\begin{align*}
\frac{P \cdot T}{P \cdot E_1} & \geq 0 \\
\frac{Q \cdot D}{P \cdot E_1} & \geq 0 \\
\frac{P \cdot T}{P \cdot E_1} + \frac{Q \cdot D}{P \cdot E_1} & \leq 1
\end{align*}
\]

(5.4a)  
(5.4b)  
(5.4c)

Using the observation that all of the dot-product values are calculated at the same time, are then all followed by a 27 cycle division to determine the reciprocal and then a multiplication for the final values for \( u \) and \( v \), the tests on \( u \) and \( v \) can be rearranged to be completed while this expensive division operation is taking place. The tests in Equations (5.4a) – (5.4b) can be simplified by comparing the sign of the numerator and denominator as is shown in Table 5.2, reducing the greater-than-zero test to a simple binary \text{XNOR} of bit 31 of the numerator and denominator. To complete the tests in Equations (5.4a) – (5.4b), this reduction is combined (using a binary \text{OR} of the two values) with an equal-to-zero test of the numerator to complete the required greater-than-or-equal-to-zero test.

Equation (5.4c) can also be rearranged to Equation (5.5a) when the denominator is positive or Equation (5.5b) when the denominator is negative. The first stage of this test uses an adder to calculate the sum of \( P \cdot T \) and \( Q \cdot D \). The resulting sum and \( P \cdot E_1 \) then pass through
two multiplexers, selecting the appropriate value based on the sign bit of \( \mathbf{P} \cdot \mathbf{E}_1 \), and into a single less-than-or-equal comparator, which completes the test from Equation (5.4c). A binary AND of the results from the rearranged tests in Equations (5.4a) – (5.4c) gives a single binary value indicating if ray-triangle intersection has occurred. In order to use this test in the same logical pipeline stage as the division, this binary value must be delayed for a further 15 stages (27 – 11 – 1) so that it is presented on the same cycle that the calculation of \( t \), \( u \) and \( v \) completes.

\[
\begin{align*}
\mathbf{P} \cdot \mathbf{T} + \mathbf{Q} \cdot \mathbf{D} & \leq \mathbf{P} \cdot \mathbf{E}_1 \quad (5.5a) \\
\mathbf{P} \cdot \mathbf{E}_1 & \leq \mathbf{P} \cdot \mathbf{T} + \mathbf{Q} \cdot \mathbf{D} \quad (5.5b)
\end{align*}
\]

With the latency of an addition (11) plus a floating-point comparison (1) being far less than that of the division (27), this optimisation allows determining if a valid intersection has occurred, without waiting for the calculation of \( u \) and \( v \), and hence does not add any extra latency to the pipeline.

**Pipeline**

After the optimisation for determining intersection in the previous section, the data-flow diagram in Figure 5.1 can be broken down into five logical stages:

- calculation of Equations (2.8a) – (2.8c) using the vector subtraction module;
- calculation of Equations (2.8d) – (2.8e) using the vector cross-product module;
- calculation of \( t \), \( u \), \( v \) and the determinant using the vector dot-product module;
- calculation of the reciprocal of the determinant using floating-point division and the conditional tests in Equations (5.4a) – (5.4c); and,
- floating-point multiplication of \( t \), \( u \) and \( v \) by the inverted determinant.

Where a data item is not required in the current stage, but is required in a later stage, a delay module is inserted to buffer the item,
using a shift-register, for as many cycles as the operation undertaken in the current stage. An example of this is the $E_1$ vector, which is not required on the cross-product stage but is required in the dot-product stage, so is delayed 17 cycles during the dot-product stage. Buffering data items like this means that new data can be inserted into the pipeline on every cycle while preserving the data dependency of calculations that require the previous value of the data.

The five logical stages and their latencies can be seen on the right-hand side of Figure 5.5. The sum of these stages is the total latency for the pipeline, which is 89 cycles. This means that it takes 89 cycles from when data is input until the result is presented at the output. This also means a total of 89 RTI calculations can take place at any one time. If the maximum number of primitives per leaf for the given render is four, that means at least 23 traversal units ($89 \div 4$) will need to have outstanding triangle intersection requests at any given time to fill the pipeline.
The percentage utilization, once the pipeline is synthesized and implemented, of look-up tables, flip-flops and DSP blocks for the target Virtex-5 LX330T-2 is shown in Figure 5.6. The figure shows that DSP blocks are by far the most consumed resource by the pipeline, with a single RTI pipeline requiring over 50% of the DSP blocks on the Virtex-5 LX330T-2, meaning only a single pipeline could be implemented in this configuration. The high DSP block utilization is expected, due to the large number of multiplication, addition and subtraction operations in the pipeline. The Xilinx floating-point cores provide configurations of these operations with lower DSP counts as a trade-off to performance, however this only needs to be investigated if the RTI pipeline latency becomes a bottleneck to performance or duplication is required to increase throughput.

When the pipeline is implemented with a target frequency of 250MHz it easily meets timing requirements and matches the operating frequency of the Xilinx PCle xpress core inside the CC. This frequency is lowered to 150MHz when integrated with the remainder of the platform, to ease timing requirements for the traversal units when the FPGA nears capacity and to not introduce clock-domain-crossing, however 250MHz is a representation of true achievable performance. When implemented at 250MHz, a single pipeline can process $2.5 \times 10^8$ RTI tests per second. This RTI test rate represents a speed over 20 times that of the Cray XD-1 implementation presented by Cameron [15], 5 times that of the Virtex-2 based implementation presented by Fender and Rose [29], over 62 times that of the GPU implementation presented by Nery et al. [63] and over 30 times that...
of the software implementation presented in the author’s previous work [18]. With newer FPGA devices containing more DSP primitives than the target FPGA and allowing for higher operating frequencies, there is plenty of potential to scale the performance of the RTI unit, if it is a bottleneck to performance, by increasing overall throughput through duplicating the pipeline and/or increasing the frequency of operation.

5.2 RAY-BOX INTERSECTION

This section outlines the design and evaluation of the ray-box intersection pipeline. First, the data-flow of the software implementation in Section 5.2.1 is analysed, revealing the data dependency of the inputs and outputs of each calculation. Then, in Section 5.2.2, the data-flow diagram and required vector operations are analysed to produce a deep pipelined design, using Xilinx floating-point modules, to optimise throughput. Finally, in Section 5.2.3, the overall pipeline performance and logic utilization is evaluated.

5.2.1 Software Implementation

Listing 5.2 shows a common C++ implementation of ray-box intersection algorithm, described previously in Section 2.1.2, solving Equations (2.10a) – (2.10b) and performing the required tests between \( t_{\text{min}} \) and \( t_{\text{max}} \) to determine intersection. The algorithm is presented in C++ for familiarity and to provide easy comparison during evaluation.

Listing 5.3 shows the C++ implementation of a function that swaps the values of the two inputs, used on line 14 of Listing 5.2.

The implementation keeps track of temporary minimum and maximum intersection limits, \( t_0 \) and \( t_1 \), respectively. These limits are first initialized to the limits of the ray, ensuring the intersection occurs in front of the camera and before any previously intersected objects.

Then, for each dimension, the near and far values from Equations (2.10a) – (2.10b) are calculated. The values are then compared to each other and swapped, if required, to ensure the near value is smaller than the far value. The intersection limits, \( t_0 \) and \( t_1 \), are
Listing 5.2: Common ray-box intersection implementation in C++.

```c++
bool BBoxIntersect(BBox b, const Ray &ray, float *tHit)
{
    float t0 = ray.mint;
    float t1 = ray.maxt;

    // for each of the three dimensions..
    for (int i = 0; i < 3; i++)
    {
        float invRayDir = 1.0 / ray.d[i];
        float tNear = (b.Min[i] - ray.o[i]) * invRayDir;
        float tFar = (b.Max[i] - ray.o[i]) * invRayDir;

        if (tNear > tFar)
            swap(&tNear, &tFar);

        t0 = tNear > t0 ? tNear : t0;
        t1 = tFar > t1 ? tFar : t1;

        if (t0 > t1)
            return false;
    }

    *tHit = t0;
    return true;
}
```

Listing 5.3: C++ implementation of the swap function.

```c++
void swap(float *f1, float *f2)
{
    float tmp = *f1;
    *f1 = *f2;
    *f2 = tmp;
}
```
then updated if closer limits are found in the calculated near and far values, i.e. a near value larger than $t_0$ or a far value smaller than $t_1$. If the new range is empty (i.e. $t_0 > t_1$) a failure can be returned immediately.

When the for loop has completed for each dimension without the range collapsing, the value in $t_0$ is used as the distance to the ray intersection with the bounding box and the function returns success.

5.2.2 Pipeline Design

Figure 5.7 shows the pipeline design for a single dimension for Equations (2.10a) – (2.10b), shown in the for loop on lines 9–17 of Listing 5.2. The subtraction of the ray origin from the bounding-box minimum and maximum values is calculated first in parallel (while the ray direction value passes through a delay block), followed by
division by the ray direction to give a initial near and far values. The near and far values are then compared and swapped, if required, so the near value is smaller than the far value. This pipeline implementation, labelled from here on as calc_near_far, has a latency of 40 cycles.

Following the assignment of values to the t0 and t1 variables (the temporary near and far intersection variables, respectively), they are initialized to the ray.mint and ray.maxt bounds on lines 3–4. On line 16, t0 is assigned a new “near” value on each iteration of the for loop only if a near value greater than the current t0 value is calculated. These assignments to t0 mean that at the end of calculation, t0 will contain largest value out of ray.mint and the three near values from the calc_near_far calculations. On line 17, t1 is assigned a new “far” value on each iteration of the for loop only if a far value less than the current t1 value is calculated. These assignments to t1 mean that at the end of calculation, t1 will contain smallest value out of ray.maxt and the three far values from the calc_near_far calculations. The functionality of these assignments is implemented in two modules using a floating-point comparator and a multiplexer. The biggest_near module compares two floating-point input values and returns the biggest value while the smallest_far module compares two floating-point input values and returns the smallest value.

Using the calc_near_far, biggest_near and smallest_far modules, the dataflow of Listing 5.2 can be broken down into four logical stages:

- calculation of near and far values for the three-dimensions using the calc_near_far module;
- finding the biggest near and smallest far values between x and y dimensions, and the z dimension and the ray bounds;
- finding the biggest near and smallest far values from the results of the previous stage; and,
- testing if the near value is smaller than the far value to determine if intersection has occurred.

The four logical stages and their latencies can be seen on the right-hand side of Figure 5.8. The sum of these stages is the total latency for the pipeline, which is 45 cycles. This means that it takes 45 cycles from when data is input until the result is presented at the output.
This also means a total of 45 RBI calculations can take place at any one time. If the width of the acceleration hierarchy for the given render is four (i.e. QBVH), that means at least 12 traversal units (45 ÷ 4) will need to have outstanding bounding-box intersection requests at any given time to fill the pipeline.

5.2.3 Evaluation

The percentage utilization, once the pipeline is synthesized and implemented, of look-up tables, flip-flops and DSP blocks for the target Virtex-5 LX330T-2 is shown in Figure 5.9. The figure shows that all resources are consumed approximately evenly, with DSP blocks being the most consumed resource at 6%. This resource consumption per pipeline means that the RBI pipeline can be replicated 16 times (100 ÷ 6) on the empty target device or 7 times (44 ÷ 6) in the leftover resources after the RPI pipeline is implemented as shown in Section 5.1.3. The Xilinx floating-point cores provide a configuration of the subtraction operation with a lower DSP count as a trade-off
to performance, however this only needs to be investigated if the RBI pipeline latency becomes a bottleneck to performance or easier duplication is required.

When the pipeline is implemented with a target frequency of 250MHz it easily meets timing requirements and matches the operating frequency of the Xilinx PCIeExpress core inside the CC and the RTI pipeline implemented in Section 5.1.3. This frequency is lowered to 150MHz when integrated with the remainder of the platform, to ease timing requirements for the traversal units when the FPGA nears capacity and to not introduce clock-domain-crossing, however 250MHz is a representation of true achievable performance. When implemented at 250MHz, a single pipeline can process $2.5 \times 10^8$ RBI tests per second. This RBI test rate represents a speed 2.5 times that of the Virtex-2 based implementation presented by Fender and Rose [29].

With the resource consumption of the RBI pipeline so low, there is plenty of potential to scale the performance of the RBI pipeline on the current target platform, if it is a bottleneck to performance, by increasing overall throughput through duplicating the pipeline. For example, once the primitive cache is introduced in Section 8.2, the RBI pipeline is duplicated to handle the increased throughput.
Priority queues (PQs) are an abstract data structure similar to a stack or queue, but with an associated priority value for each data element. This priority value is used to determine the order in which elements are extracted from the queue, where higher priority elements are extracted first and elements with the same priority are extracted based on their order in the queue. A PQ typically implements two primary instructions, insertion of an element into the queue with a set priority, called push, and extraction of the highest priority element, called pop, both of which are typically followed by a sorting algorithm to maintain the order of the queue. Priority queues are most commonly found within the context of communication and computer networking hardware appliances, for instance prioritised IP packet queues for network routing, where there has already been research into hardware priority queues for this purpose [57, 83].

Graph search algorithms, such as those that solve the shortest path problem, often require priority queues and are a popular area of research for hardware acceleration. The Floyd-Warshall, Sankoff and Kruskal and Dijkstra algorithms all require a large number of arithmetic operations which can be performed concurrently and can therefore be accelerated on parallel architectures [12, 34, 81]. Tree structures can be very efficiently implemented in hardware due to their ability to process multiple branches in parallel and can greatly reduce run times for Dijkstra’s Algorithm [87] as well as other graph problems such as minimum dominating set [89] and minimum spanning tree [73]. Hardware and software integration has also been explored as a viable method for speeding up computation times in graph algorithms [11]. Whilst some of these algorithms are traditionally implemented in software using a priority queue, it is often more efficient to use hardware to perform direct comparisons with the entire data set at once, rather than obtaining the data from the priority queue, which suits the parallel nature of FPGAs. This method of simultaneously comparing a data value to an entire data set is also used to implement highly efficient priority queues, described in Section 6.1 below. However, these implementations
only consider increasing performance and do not address the issue of scalability. A scalable priority queue has been implemented by Moon et al. [56]. Their approach uses a pipelined heap, allowing it to perform insertion and extraction quickly and at constant times, well suited to their target application of signal processing.

Priority queues are used in ray tracing when traversing acceleration hierarchy, implemented in the Traversal Unit of the proposed platform, described previously in Section 4.2.4. As a ray descends down the acceleration hierarchy it tests for intersection with the bounding boxes of the current nodes children. All child nodes where the ray intersects their bounding box are stored on a priority queue, ordered by the distance along the ray path that their intersection occurred. Once all nodes have been sorted on the priority queue, traversal continues by popping the head of the queue as the next node to be processed. This ensures that; all nodes are processed in front to back order relative to the current ray direction, each node is traversed at most once and, each necessary node exactly once. Processing nodes in front to back order guarantees that the first ray-triangle intersection along the ray path is the closest, allowing for early termination of traversal.

Implementing priority queues on GPUs posed a problem to initial research of their use for ray tracing acceleration due to the small and inflexible amount of memory available to each computation thread. This lead to the following techniques that side stepped this limitation at the cost of extra intersection computations, outlined previously in Section 3.4.1. Foley et al. [32] presented \textit{kd-restart}, modifying the traditional \textit{kd-tree} traversal algorithm to eliminate the use of a priority queue altogether. \textit{kd-restart} modifies the ray \textit{tmin} and \textit{tmax} parameters and restarts traversal from the top of the tree when an intersection is not found in a leaf. When traversal is restarted from the top of the hierarchy, intersection tests with the new \textit{tmin} and \textit{tmax} parameters result in traversal taking a different path down the tree, eventually reaching the next closest leaf. Horn et al. [40] presented \textit{push-down}, a technique that moves the restart position down the tree to the lowest sub-tree enclosing the remaining nodes and \textit{short-stack}, which uses a small, fixed-size stack and falls back to stack-less traversal if the stack should underflow. Novák [64] presented another method for alleviating the impact of restarting called \textit{stack-spilling}. Instead of having only a \textit{short-stack}, another long-stack is added in
local memory and a spilling technique is used to deal with overflows and underflows of the short-stack.

The short-stack with spilling concept is also explored by Chandra and Sinnen [17] where they use a hardware queue on an FPGA and spill into a software queue. A smaller hardware queue is embedded on the FPGA and a larger software queue resides in memory to preserve the speed up of a hardware implementation while maintaining the scalability of a software queue. In this chapter, a hardware-only development of this concept is presented that combines a fast single-cycle queue, detailed in Section 6.1, and a scalable heap queue, detailed in Section 6.2. The implementation details and evaluation of this hybrid queue are presented in Section 6.3, which shows the hybrid queue maintains a good balance between performance and scalability for applications where the size of the working set of data can have large variance. Finally, in Section 6.4, a variation of the heap implementation is presented which is optimised for FPGA dual-port RAM implementation and for workloads that repeat several consecutive push operations followed by a single pop operation, which is the workload required when ray tracing.

### 6.1 Single-Cycle Priority Queue

Priority queues can be efficiently implemented in hardware using a series of comparators and a shift register, as shown by Chandra and Sinnen [17]. A high-level view of this architecture is shown in Figure 6.1. Each storage element, labelled from highest to lowest priority $s_0$ to $s_7$ in the figure, contains a register that stores a priority and data pairing. The priority in each storage element is then able to be compared in parallel with the input data. Figure 6.2 shows an example of push and pop operations for a single-cycle queue. Input data is stored in the first register of lower priority and all registers...
(a) Initial priority values of items in an eight element single-cycle queue.

(b) An item with priority 10 (in yellow) is “pushed” onto the queue, moving the lower priorities 5 and 1 (in green) down the queue while higher priorities 26, 19 and 13 (in blue) remain in the same position.

(c) The highest priority in the queue, 26, is “popped”, moving the remaining lower priorities (in red) up the queue and making 19 the new head of the queue.

Figure 6.2: push and pop operations for a single-cycle queue.

<table>
<thead>
<tr>
<th>CMP IN</th>
<th>CMP OUT</th>
<th>MUX SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>OUT (blue)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>IN (yellow)</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>ABOVE (green)</td>
</tr>
</tbody>
</table>

Table 6.1: Decision matrix to determine multiplexer select lines for a push operation.

of lower priority are shifted across, effectively sorting the data in a single-cycle. This means that the storage element must decide to retain the same value (if it is of higher priority), accept the input value (if it is the first element of lower priority) or accept the value of the element above it (if it is of lower priority). Figure 6.2b shows these three options in blue, yellow and green respectively. Data can be extracted by outputting the highest priority register and shifting all other registers along, also in a single-cycle, meaning the storage element must also accept the value of the option below it. Figure 6.2c shows these storage elements in red.

A detailed view of the storage element is shown in Figure 6.3. The four coloured options from Figure 6.2b and Figure 6.2c make up the four inputs into the pre register multiplexer at the bottom of the
Figure 6.3: Storage element used in the construction of the shift-register priority queue.

<table>
<thead>
<tr>
<th>CMP IN</th>
<th>CMP OUT</th>
<th>MUX SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>BELOW (red)</td>
</tr>
</tbody>
</table>

Table 6.2: Decision matrix to determine multiplexer select lines for a pop operation.

The comparator in the top left compares the priority of a new data element with the priority of the data element currently stored, with a value of 1 indicating the new element is of higher priority and 0 indicating lower priority. This value is passed to the CMP OUT port which is connected to the CMP IN port of the storage element below the current storage element. Likewise, connected to the CMP IN port of the current storage element is the CMP OUT port of the above storage element. Using these two values, the control logic uses the decision matrices in Table 6.1 and Table 6.2 to implement the options described in Figure 6.2b and Figure 6.2c by selecting the corresponding input to the pre register multiplexer. The type of operation is selected by asserting either of the PUSH or POP signals, while presenting a new data element on the IN port for a push operation.

This architecture creates a highly efficient priority queue capable of inserting and extracting data in a single-cycle. The primary drawback of such an implementation is the heavy usage of the available logic elements, with each storage element requiring a separate comparator unit and 64 4:1 multiplexers (for 32 bits of data with a 32 bit priority). Further, for large queues of this type, the fanout of the input data net can also reduce the maximum operational frequency. If the application deals with a data set of unknown size or a size with
high variance, very large priority queues are required to process the data. A scenario could arise where much of the synthesized priority queue is rarely used whilst still occupying a large proportion of logic elements.

Implementation details for the single-cycle priority queue are detailed in Section 6.3.1 and scalability is evaluated in Section 8.1.

6.2 HEAP

A heap is a tree-based data structure that satisfies the heap property: Given node A, which is the parent of node B, then the priority of node A is higher than the priority of node B. This relative ordering of
(a) The heap before the pop operation.

(b) The head (26) is “popped” and replaced by the value at the tail (4). The target location is set to the head (grey) and compared to the two children (19 and 13), swapping with the largest (19) if smaller than either.

(c) The target location (4) is smaller than either of the two children (9 and 14) so the process continues, swapping with the largest (14).

(d) The target location (4) is smaller than either of the two children (10, but not 1) so the process continues, swapping with the largest (10).

(e) When the target location is greater than both of the children or there are no children (true in this case), the pop operation has completed.

(f) The heap after the pop operation.

Figure 6.5: A heap pop operation.
parents and children applies across the entire heap and ensures the root node will always have the highest priority [25].

Heaps are typically stored using an array of data with the first index in the array containing the root node. For a heap where the first array index is 1 and given a node at location \( n \), the parent node can be found at \( n/2 \) and the left and right child nodes can be found at \( 2n \) and \( 2n + 1 \) respectively.

**Data:** heap is the array storing the heap, with length heap.length.

**Input:** index is the index of the heap to start the heapify operation from.

**Function** HeapifyUp(index):

1. parent ← index/2;
2. if index > 1 and heap[parent] < heap[index] then /* swap values */
   1. (heap[index], heap[parent]) ← (heap[parent], heap[index]);
3. HeapifyUp(parent);
4. end

**Algorithm 6.1:** The HeapifyUp function, which restores the heap property to a heap when a given index and its parent violate the heap property.

As with the priority queue in Section 6.1 previously, a heap must also implement push and pop operations, described below. Figure 6.4 shows an example push operation, adding the value 15 to the heap. The new value to be “pushed” is written to the bottom of the heap, where the target location is also set, shown in grey. The value in the target location is compared with its parent value. If the target location value is greater than the parent value, the parent value is written into the target location and the target location is moved to the parent location. This process continues up the heap until the target location value is less than the parent value or the target location reaches the top of the heap, where the push operation has completed. This functionality is achieved by the HeapifyUp function, defined in pseudo-code in Algorithm 6.1, for an array-backed heap with an index starting at 1.

Figure 6.5 shows an example pop operation, removing the value 26 from the head of the heap. The lowest value on the heap, 4, is then moved to the head of the heap where the target location is also set and shown in grey. The value at the target location is then compared to the two values below it. If the value at the target location is smaller than either of the children, the largest of the two children is swapped
**Data:** heap is the array storing the heap, with length heap.length.

**Input:** index is the index of the heap to start the heapify operation from.

1. **Function** HeapifyDown(index):
   2. left ← 2 × index;
   3. right ← 2 × index + 1;
   4. largest ← index;
   5. if left ≤ heap.length and heap[left] > heap[largest] then
      6. largest ← left;
   7. end
   8. if right ≤ heap.length and heap[right] > heap[largest] then
      9. largest ← right;
   10. end
   11. if largest ≠ index then
       12. /* swap values */
       13. (heap[index], heap[largest]) ← (heap[largest], heap[index]);
       14. HeapifyDown(largest);
   15. end

**Algorithm 6.2:** The HeapifyDown algorithm, which restores the heap property to a heap when a given index and its two direct children violate the heap property.

This section presents the novel hybrid queue, a combination of the single-cycle queue from Section 6.1 and the heap from Section 6.2. If an application has to deal with a working data set of unknown size or a size with high variance, a very large priority queue is needed to store the required data. The resulting large priority queue occupies

6.3 **HYBRID QUEUE**

This section presents the novel hybrid queue, a combination of the single-cycle queue from Section 6.1 and the heap from Section 6.2. If an application has to deal with a working data set of unknown size or a size with high variance, a very large priority queue is needed to store the required data. The resulting large priority queue occupies
a large proportion of logic but during normal operation the majority of it is rarely used. While a heap can be used to reduce the required logic size, this introduces a performance penalty to maintain the heap after each operation.

Provided the size of the working data set is distributed such that it remains at a certain mean for the majority of execution time, the hybrid queue has been designed in such a way that most of the operations are performed with the speed of the single-cycle queue. When the data size moves to larger extremes and the single-cycle queue becomes full, extra values are accommodated by “spilling” from the bottom of the single-cycle queue into the heap queue. For these larger extremes, performance degrades to that of the heap queue, but allows the hybrid queue to be both fast during normal operation as well as being able to cope with larger data sets in extremes. Figure 6.6 shows an example of push and pop operations for a small hybrid queue. The single-cycle queue and heap store four data items each and are shown in orange and purple respectively. This example queue would be used when, during the majority of execution, four or less items are required to be stored, but eight items also need to be stored on rare occasions. Figure 6.6a shows
an example push operation, where 13 is to be pushed onto the queue. Because the single-cycle queue is full, the bottom value, 7, spills onto the heap queue, shown in Figure 6.6b. Figure 6.6c shows an example pop operation, where 26 is to be popped from the head of the queue. Because the single-cycle queue has now one empty slot, the head of the heap queue, 7, is popped and placed on the tail of the single-cycle queue, shown in Figure 6.6d.

The combination of a single-cycle queue and a heap allows us to maintain a good balance between performance and scalability for applications where the size of the working set of data can have large variance. Implementation of the hybrid queue is described in Section 6.3.1 below and performance is evaluated in Section 6.3.2.

6.3.1 Implementation

The following subsections outline the implementation of the single-cycle queue, heap and hybrid queue.

Single-Cycle Queue

The single-cycle queue is implemented in VHDL using the same method as Chandra and Sinnen [17]. It is constructed from a number of modular storage elements, implemented in VHDL from the logic shown previously in Figure 6.3, which each contain a data item. Each block is linked with the master push/pop control signals and the new element input. Each block also communicates the result of comparison with a new element to the block below it, and accepts the comparison result of the block above it. Using the above signals, the decision matrices in Table 6.1 and Table 6.2 are used to construct a case statement which is synthesized into the multiplexer and control logic that will select the data to be stored in the storage register on the next clock edge. The architecture utilises generic parameters and generate statements at synthesis time to facilitate trivial scalability of data width, priority width, and queue size. The critical path of the implemented single-cycle queue is the comparator in each storage unit, which has a maximum operating frequency of 450MHz [102]. However, the fanout of the input data can become the critical path when the queue size is greater than approximately 100 elements or when FPGA utilization approaches capacity.
Heap

The arithmetic required to calculate parent and child locations, described previously in Section 6.2, is particularly convenient for implementation in hardware as the parent node can be found simply by a logical right shift and the left and right child nodes can be found by a logical shift left, shifting in either 0 or 1 respectively. This allows the parent and child nodes to be found without the need of complex logic.

The heap push operation was implemented in VHDL by adapting the HeapifyUp function in Algorithm 6.1 to work efficiently with FPGA dual-port memory. FPGA dual-port memory, as the name suggests, has two ports which memory can be read from independently, but only one fixed port which is able to write to memory. As can be seen on line 4 of Algorithm 6.1, swapping of the value in heap[index] with the value in heap[parent] is needed, which requires two write operations and therefore two cycles on the dual-port memory, as only one value can be written at a time. Looking at the example from Figure 6.4 earlier, the writes required for the entire process are shown in the listing below:

Listing 6.1: Series of writes required during example push operation.¹

```
1 // Push data received:
2 heap[13] = push_data
3 // Swap 1:
6 // Swap 2:
```

From the listing above, if the movement of the original push_data value is followed, it can be seen that it is first written to location 13, then swapped to location 6 and finally swapped to location 3. Instead of writing this value multiple times only to have to swapped, this value is instead stored in a working register (working_reg) and only written to memory just before the push operation terminates. Eliminating continuously writing this value means each value swap

¹Note that the input values to the swap lines are cached in a temporary register, so the write on the second swap line does not use the new value that was written on the previous swap line.
(a) The heap before the modified push operation. The working register is shown in the top right-hand corner.

(b) The new value (15) is placed in the working register and target location set to the bottom of the heap (grey). The working register is greater than the target location parent (7) so the parent is written to the target location and the target location moved up.

(c) The working register (15) is greater than the target location (grey) parent (13) so the process continues.

(d) When the working register (15) is less than the parent value (26, so true in this case) or the target location is at the head of the heap, the working register value is written to the target location and the push operation has completed.

(e) The heap after the modified push operation.

Figure 6.7: A modified heap push operation that uses a working register to reduce writes to memory.
Figure 6.8: A modified heap pop operation that uses a working register to reduce writes to memory.
on FPGA dual-port memory can be completed in a single cycle, plus one cycle at the end to write the working register value.

Figure 6.7 shows the same example as Figure 6.4 but with the above changes to the algorithm. The new value to be “pushed” is placed in the working register and the target location, shown in grey, is set to the bottom of the heap. The working register is compared with the parent of the target location. If the working register value is greater than the parent value, the parent value is written into the target location and the target location is moved to the parent location. This process continues up the heap until the head is reached or the working register value is less than the parent value, where the working register value is written to the target location and the push operation has completed. These changes to the algorithm result in the series of writes in the listing below:

Listing 6.2: Modified series of writes during example push operation.

```
// Push data received:
working_reg = push_data
// Swap 1:
// Swap 2:
// Operation terminates:
heap[3] = working_reg
```

While this technique has only saved us a single cycle in the example above, it will save an extra cycle for any additional required value swap, meaning it will have a greater impact as the heap gets larger or for values of higher priority that move further up the heap. Additionally, because the parent value only needs to be read from memory for each level of the heap, the read of the next parent can be completed in parallel with the write of the swap value to allow this functionality to be implemented in a single state. The resulting FSM to implement the complete operation is shown in Figure 6.9.

The heap pop operation was implemented in a similar manner to the push operation, by adapting the HeapifyDown function in Algorithm 6.2. We use the same working register method as previously to save unnecessary writes for the swap on line 14. Figure 6.8 shows the same example as Figure 6.5 but with this modification to the algorithm. The top value is “popped” from the heap. The lowest value on the heap is then moved into a working
Compare the current index value with its parent value, swapping if necessary.

\[
\text{if } \text{heap[parent]} \geq \text{working_reg} \text{ or index = 0}
\]

\[
\text{heap[index]} \leftarrow \text{heap[parent]}
\]

\[
\text{parent} \leftarrow \text{parent}/2
\]

\[
\text{end}
\]

\[
\text{heap[parent]} < \text{working_reg}
\]

\[
\text{and index} > 0
\]

Figure 6.9: FSM for implementation of push operation.

Swap values if required, return to IDLE if not.

\[
\text{if largest} \neq \text{index then}
\]

\[
\text{heap[index]} \leftarrow \text{heap[largest]}
\]

\[
\text{index} \leftarrow \text{largest}
\]

\[
\text{else}
\]

\[
\text{heap[index]} \leftarrow \text{working_reg}
\]

\[
\text{end}
\]

\[
\text{largest} \neq \text{index}
\]

Figure 6.10: FSM for implementation of pop operation.
register and the target location is set to the head of the heap, shown in grey. The value in the working register is compared to the two values below the target location. If the value in the working register is smaller than either of the children, the largest of the two children is written to the target location and the target location is then moved down the heap. This process continues down the heap until the value in the working register is greater than both of the target location children or the target location has no children, where the value of the working register is written to the target location and the pop operation has completed.

The comparison of the left and right children on lines 5 - 10 of Algorithm 6.2 is done in parallel, which requires additionally comparing the left and right child to each other to ensure the highest priority of the two is selected. As these steps require two reads for the left and right children and a write for the swap, which cannot be done in a single cycle using dual-port memory, the function must be spread over two states. The resulting FSM to implement the complete operation is shown in Figure 6.10.

To complete the control logic implementation the FSMs in Figure 6.9 and Figure 6.10 are combined into a single FSM by merging their IDLE states together and requiring the same conditions to transition to the next state for each operation. From both of these FSMs it can be seen that there are three comparators required. Two are used to compare the output from each read port to the working register, and one to compare the two outputs to each other. The array that stores the heap is implemented using FPGA dual-port memory primitives. Note that the array in Algorithm 6.1 and Algorithm 6.2 is indexed starting at 1, not 0 as is done with FPGA memory. To account for this, dedicated logic is added in front of each port input address to decrement the address when it is presented. This allows us to keep using the simple and FPGA friendly arithmetic for finding parent and child nodes in the control logic. The combination of control logic, comparators and dual-port memory make up the complete heap implementation. The critical path of the implemented heap are the three comparators, which have a maximum operating frequency of 450MHz [102].

Hybrid Queue

The hybrid queue implementation, shown in Figure 6.11, is created in VHDL as a wrapper around the existing single-cycle queue and
Figure 6.11: HybridQ wrapper around the single-cycle queue (labelled SC) and the heap.

the heap modules, described previously. The single-cycle queue acts as the primary queue and is directly connected to the external control signals and data input. Once this queue is filled, overflowing data is directed to the heap. Signals are emitted by the single-cycle queue to control the operation of the heap when the single-cycle queue is considered to be full. The heap emits a busy signal while maintaining the heap, which blocks all input activity until complete. The busy signal is not emitted when the single-cycle queue is not full, allowing the hybrid queue to process data at single-cycle speeds, only degrading in performance when spilling into or out of the heap.

6.3.2 Evaluation

In this section the performance and scalability of the hybrid queue is evaluated against the fast single-cycle queue and scalable heap implementations. The performance of each queue is compared, timing the execution of multiple usage profiles - combinations of push and pop operations designed to simulate different possible application workloads. The mean and standard deviation of the population of the data in the queue and the order the data is processed is varied to see the affect different usage profiles have on performance. Finally, the performance and scalability of different size hybrid queues is evaluated for a selected usage profile.

Method

Testing is performed by creating VHDL test benches that push and pop data onto the given queue to fit a selected usage profile, executing the test bench in Modelsim, and timing the execution for each type of queue. The priority queues evaluated are: a single-cycle
queue, a heap, and hybrid queues with single-cycle queue sizes of 100, 300, 500 and 780. The heap is implemented with the Xilinx BlockRAM primitive which has a size of 1,024 elements when using the minimum configuration. We use a single-cycle queue size of 1,024 to match this. When the length of the hybrid queue is mentioned in the following sections, the length of the single-cycle queue within it is what is being discussed. The same heap of size of 1,024 is used within all the hybrid queues, due to the minimum size of the BlockRAM primitive, i.e. the hybrid queue of length 780 can potentially store 1,804 elements. We artificially limit this queue size during comparison by ensuring we never exceed 1,024 elements on the queue at any one time. Logic utilisation is recorded for each queue after synthesis in Xilinx ISE.

To create usage profiles, data sets are first generated in ordered, reverse ordered and random ordered sequence. For example, for a sequence of numbers from one to five, the ordered sequence is [1, 2, 3, 4, 5], the reverse ordered sequence is [5, 4, 3, 2, 1] and a random ordered sequence is one such as [3, 4, 2, 1, 5]. The length of these sequences is referred to as the number of data points in the figures in the following sections. A set of normally distributed integers with a given standard deviation is then generated representing the desired queue population. push or pop operations are recorded accordingly until the queue population is equal to the generated integer, whereupon the next generated integer is used. Once all of the elements in the data set have been pushed onto the queue, the queue will pop until empty before finishing. For example, for a generated desired queue population of [3, 2, 4], the following sequence of push and pop operations is generated: [push, push, push, pop, push, push, pop, pop, pop, pop]. The first required queue population is 3, so three values are pushed onto the queue. The next requested value is 2, lower than the current population of 3, so values are popped until the required population is met (only one in this case). The next required value is 4, higher than the current population of 2, so values are pushed until the required population is met (two in this case). As there are no remaining values in the data sequence to push onto the queue, the queue is emptied by popping all the remaining populated values (four in this case) and the process is finished. This sequence of operations is then used to create three test benches for the three ordered sets of numbers, i.e. the following operations are implemented in the test bench for the random ordered sequence:
Figure 6.12: Graphs showing execution time of different size hybrid queues (in brackets) for different mean queue populations.

[push (3), push (4), push (2), pop, push (1), push (5), pop, pop, pop, pop]. The number of clock cycles is then recorded for each queue type and for each usage profile by running the test benches under Modelsim until completion.

This method allows the usage profiles to somewhat emulate the randomness of generic real world data and, whilst not a complete statistical analysis, will provide some useful information to illustrate the performance of the hybrid queue. The results for each queue are presented and analysed in the following subsections.

**Effect of Mean Queue Population**

Figure 6.12 demonstrates the effects of the mean queue population in relation to hybrid queues of different size by comparing the execution times for usage profiles with a mean queue population of 500, standard deviation of 25% (125) and mean queue population of 780, standard deviation of 25% (195).

As expected, hybrid queue sizes of 100 and 300 remain largely unaffected by this change as the majority of operations are performed in the heap for both cases. However, for hybrid queue sizes of 500 it can be seen that a mean queue population of 780 will result in a slower performance than a mean queue population of 500. This is expected, due to more of the operations being performed in the slower heap queue for the former case whilst the operations are spread equally between the single-cycle queue and heap queue in the latter case. Likewise for a hybrid queue size of 780, a usage profile
with an average data population of 500 will be performed entirely in the single-cycle queue and thus is significantly faster when compared to the usage profile of mean data population 780.

It can also be seen that a hybrid queue size of 300 performs better than a hybrid queue size of 100, when the average queue population is either 500 or 780. This is because, for the same queue population, when more elements are stored in the single-cycle queue, less elements spill onto the heap, meaning operations complete faster as they require maintenance of a smaller heap. From this it can be seen that even if the expected queue population far exceeds the single-cycle queue size, there are still benefits in using the hybrid queue.

**Effect of Standard Deviation of Mean Queue Population**

Figure 6.13 shows the effect of the standard deviation of the mean queue population for a hybrid queue size of 500 and mean queue population of 500. Unlike the mean queue population, the standard deviation has very little effect on execution time. It can be seen that the difference between the execution times of the 100 and 300 length hybrid queue sizes are less differentiated when the standard deviation is lower. This is likely due to less variance in the population of the hybrid queue, and thus the population of the heap is more consistent, resulting in relatively more consistent data.
Figure 6.14: Graphs showing execution time of different data orders for hybrid queues of different sizes.

<table>
<thead>
<tr>
<th>Queue</th>
<th>LUTs</th>
<th>Registers</th>
<th>FMax (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle</td>
<td>$1.27 \times 10^5$</td>
<td>65,536</td>
<td>108.7</td>
</tr>
<tr>
<td>Heap</td>
<td>11,536</td>
<td>2,352</td>
<td>87.8</td>
</tr>
<tr>
<td>HybridQ (500)</td>
<td>62,194</td>
<td>32,000</td>
<td>91.7</td>
</tr>
</tbody>
</table>

Table 6.3: Logic utilization for single-cycle queue, heap and hybrid queue with length of 1,024 elements when synthesized in Xilinx ISE 14.4 targeting a Virtex-5 LX330T-2 FPGA.

Effect of Data Order

Figure 6.14 shows the effect of the order of input data on execution time. For a heap, ordered data is the best case scenario and reverse ordered data is the worst case scenario. This is due to the HeapifyUp and HeapifyDown functions having to continuously traverse the entire heap for reverse ordered data whilst only traversing the initial node for ordered data. As the single-cycle queue executes at a single-cycle regardless of the data type, a larger single-cycle queue size will also reduce the execution time difference as less operations are performed in the heap. This is illustrated by the relative execution speeds in the left and the right graphs of Figure 6.14, which were measured for different hybrid queue sizes. Naturally, random ordered data sits in between ordered and reverse ordered data, preforming no better than the best case of ordered data and no worse than the worst case of reverse ordered data.
Effect of Queue Type on Logic Utilization and Performance

Figure 6.15 shows relative execution times between single-cycle queue, heap and hybrid queue for random data with a mean queue population of 500 and 25% (125) standard deviation. Table 6.3 shows the required logic utilization and achievable clock frequency for each queue type when synthesized in Xilinx ISE 14.4 targeting a Virtex-5 LX330T-2 FPGA (as is used on the target platform). The execution time in Figure 6.15 is calculated using the number of cycles to complete the simulated workload, as done previously, multiplied by the synthesized clock frequency of the given queue, giving relative performance which includes the effect of the speed that the given queue can operate at.

The performance from Figure 6.15 and logic utilization from Table 6.3 shows the hybrid queue offers better performance than the heap but uses more logic, and less logic utilization than the single-cycle queue but with slower performance. This means the hybrid queue offers better performance than a heap queue and better scalability than a single-cycle queue, allowing a user to pick a compromise between performance and scalability for their application instead of only the extremes. The critical path of the implemented hybrid queue is comparators within the included single-cycle and heap implementations.
6.4 Heap Optimisations

The scalability evaluation of the single-cycle, heap and hybrid queues, presented later in Section 8.1, shows that the push and pop operation performance provided by the single-cycle queue is heavily outweighed by its inability to scale. Only six single-cycle queues were able to fit on the target device where 32 heap queues were able to fit. With each traversal unit requiring a priority queue, this means over five times more traversal units can fit on the target device by using a heap queue. The performance increase gained from more traversal threads outweighs the performance increase from single-cycle queue operations. This section outlines optimisations made to the heap to increase performance and suitability for ray-tracing type applications, including: optimisations for workloads that require bursts of push operations, less-than floating-point comparators for priorities, and small heap sizes.

6.4.1 Optimisation for Burst-Heavy Workloads

A “burst-heavy” queue workload is one where there are large time periods without operations and, when they occur, operations occur close together over a short period of time (in “bursts”), often on consecutive cycles. This workload is managed well by the single-cycle queue, where each operation can be handled in a single cycle. However, for a heap, it must be maintained immediately after each operation is received, meaning each incoming operation that arrives while heap maintenance is under way for a previous operation must be queued in a storage mechanism such as a FIFO and processed later, one-by-one.

For a heap where bursts of push operations are followed by a single pop operation, such as when traversing acceleration hierarchy, this means that each incoming push operation must be queued, and the pop operation must wait until the preceding push operations are processed. Also, if a push operation is requested during maintenance of the heap for a pop operation, the push will need to be queued until maintenance for the pop is complete, because the order of the operations can not be modified.

The need to queue requests for both of these situations can be removed by accepting and storing push operations at the tail of the heap during any stage of heap maintenance, for either push or pop operations.
6.4 Heap Optimisations | 111

(a) A heap without any push operations pending at the tail.

(b) While the heap is being maintained, the value 4 arrives and is written to the tail of the heap, incrementing push_location once the write is complete.

(c) While the heap is being maintained, the value 21 arrives and is written to the tail of the heap, incrementing push_location once the write is complete.

(d) The heap returns to IDLE and maintains the heap for the value 4, incrementing heap_size once maintenance is complete.

(e) The heap again returns to IDLE and maintains the heap for the value 21, incrementing the heap_size once maintenance completes, which now has the same value as push_location again, meaning heap maintenance has completed.

Figure 6.16: Diagram showing the modified heap where the values 4 and 21 arrive during heap maintenance and are later processed in order once the heap returns to IDLE.
operations, and processing them after the current maintenance is completed. This is achieved by keeping an additional push_location register, writing any incoming values to the location stored in this register and incrementing it afterwards. When the heap reaches its IDLE state, it compares the value in the push_location register with the value of the heap_size register. If the value in the push_location register is larger, the heap proceeds to maintain the heap for the stored push values, incrementing heap_size until it matches the value of the push_location register. These modifications allow push operations to be accepted at any stage of operation, while pop operations must wait for heap maintenance to complete before being processed, desirable for a burst-heavy workload while maintaining the scalability of the heap.

An example of these modifications is shown in Figure 6.16 for a heap that initially has 10 values on it. The values 4 and 21 arrive during other heap maintenance and are stored at the tail of the heap, increasing the push_location to 12. When the heap next returns to its IDLE state the heap_size (10) and push_location (12) values are compared. As they are not the same, the heap is maintained first for the value 4, increasing the heap_size to 11, then for the value 21, increasing the heap_size to 12. As the heap_size (12) and push_location (12) values are now the same, heap maintenance has completed for the two pushed values.

The typical workload for the priority queue in the TU during ray-tracing is burst-heavy, requiring a series of 0 to M sequential push operations as intersection results are returned (where M is the width of the acceleration hierarchy). If intersection is to continue after the results are processed, this is followed by a single pop once the top of the queue is sent to the IC for the next required intersection tests. Also, when the time taken to maintain the heap for the pop operation is longer than the time taken for the intersection results to return, the results must be stored while maintenance for the pop operation is completed. The TU state machine must still wait before continuing traversal while the heap is maintained for these results, but the small logic resource cost of the push_location register and minor algorithm changes are traded for the larger amount logic resources required to queue and process incoming requests. The burst-heavy optimisations to the heap make it perfect for use for ray-tracing on the proposed platform.
6.4.2 Floating-Point Priorities

As the intersections queued during ray tracing have a single-precision floating-point priority, representing the distance along the ray the intersection occurred, the integer comparator in the single-cycle queue and heap implementations must be swapped for a floating-point comparator, which is essentially an integer comparator but accounts for special numbers defined in the IEEE 754 standard \[7\]. As smaller distances along the ray are of higher priority, this means that the comparator must be a less-than comparator and a priority of \texttt{FP\_INFINITY} represents an empty storage block. The floating-point comparators used in the platform evaluation are implemented using the Xilinx floating-point comparator IP, each requiring 45 LUTs and 19 registers and are capable of operating at a maximum frequency of 450 MHz \[102\].

6.4.3 Smaller Heap Sizes

The BlockRAM primitive used until now has a minimum configurable size of 1,024 elements for a 64 bit width. However, the stack used in LuxRays software implementation has a maximum usable size of 64 elements, raising an error if more elements need to be stored. Over all the test scenes used, including large high quality scenes, this error was not once seen, showing that a limit of 64 is sufficient for the majority of scenes. With such a large difference in the size of the BlockRAM based heap and the stack used in LuxRays, it is clear that the majority of the BlockRAM storage would go unused when used for ray-tracing, and would perhaps be better used to implement a cache as is explored later in Chapter 7.

On current generation Xilinx FPGAs there are two types of slices which implement the core FPGA functionality: SLICEL (L for logic) and SLICEM (M for memory). Approximately two thirds of slices are SLICEL and the remainder SLICEM. The LUTs in SLICEMs can be implemented as a synchronous RAM resource called a distributed RAM element. Multiple LUTs in a SLICEM can be combined in various ways to store larger amount of data. One such configuration is a dual-port 64 x 1-bit RAM, requiring two of the four LUTs in a SLICEM, and labelled the RAM64X1D primitive. This primitive has the same input and output port configurations as the BlockRAM used previously, only with a size matching the stack used in LuxRays,
Table 6.4: Logic utilization for single-cycle queue and heap with 64 storage elements, each storing 32 bits of data and 32 bit float-point priorities, when synthesized in Xilinx ISE 14.4 targeting a Virtex-5 LX330T-2 FPGA.

<table>
<thead>
<tr>
<th>Queue</th>
<th>LUTs</th>
<th>Registers</th>
<th>FMax (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle</td>
<td>19,070</td>
<td>4,096</td>
<td>217.2</td>
</tr>
<tr>
<td>Heap</td>
<td>721</td>
<td>247</td>
<td>168.5</td>
</tr>
</tbody>
</table>

allowing it to slot easily into the previous heap implementation with minor modifications. As the heap used for ray-tracing on the target platform requires a width of 64 bits (32 bits of priority, 32 bits of data), 64 of these primitives are required per heap.

With all SLICEMs of the Virtex-5 LX330T-2 on the target platform configured as distributed RAM, a maximum of 3,420 Kbit of memory can be implemented. As each 64 x 64-bit heap memory requires only 4 Kbit of distributed RAM, this means that a maximum of 855 heap memories can fit within the SLICEMs on the target device. This is far more than can realistically be implemented as the logic requirements of 855 traversal units to operate the heaps would first fill the remaining logic of the target device, meaning it will not be a limiting factor to the number of heap implementations. Table 6.4 shows the difference in resource utilization between the distributed RAM heap implementation and the single-cycle queue when used for their evaluation in Section 8.1. Of the 721 LUTs required for the heap, 128 are for element storage and the remainder are the floating-point comparators and control logic.
Efficient Object Caching

A cache is a component that transparently stores data so that future requests for the data can be served faster. The data stored in a cache might be previously computed values or values that are stored in another, slower location. If the requested data is contained in the cache (a cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (a cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. The hit rate of a cache is the average percentage of requests able to be served from the cache and, inversely, the miss rate is the average percentage of requests unable to be served from the cache. Therefore, the greater the number of requests that can be served from the cache (i.e. a high hit rate), the faster the overall system performance becomes [70].

To achieve a high hit rate, the items stored in a cache should be items accessed frequently or be likely to be accessed in the future. Data that exhibits these traits is said to have a temporal or spatial “locality of reference”, respectively. Temporal locality (i.e. locality in time) refers to when data that is accessed is likely to be accessed again soon. Spatial locality (i.e. locality in space) refers to when data is accessed and those data items in memory locations close by are likely to be accessed again soon. These forms of locality exist naturally in algorithms from constructs like loops and arrays and also from patterns in memory access specific to the behaviour of the algorithm [70].

A direct mapped cache is a cache where each data element can be stored in exactly one place. At the other extreme of this is a fully associative cache, where each data element can be stored in any location. When there is only one possible cache location for a data element in a direct mapped cache, then, to check if a data element is stored in the cache, only one location must be checked. Conversely, to check if a data item is present in a fully associative cache, every location must be checked. A common compromise between the two extremes is an N-way set associative cache, in which each data entry can be placed in one of N locations and therefore only N locations
need be checked. Checking more locations requires more power, chip area and potentially time, but caches with more associativity suffer fewer misses so provide better performance [70].

In order to store a new entry in the cache after a miss, the cache may have to evict an existing item. The heuristic used to determine what item to evict is called a replacement policy. Designing effective replacement policies can be hard as which existing cache entry is least likely to be used in the future must be predicted to optimise locality of reference of the remaining entries. Predicting future use of data is difficult and can vary largely between applications using the cache, so a wide variety of replacement policies have been developed [108]. The most commonly used scheme is “least recently used” (LRU), which evicts the data item that has been unused for the longest time. LRU is implemented by keeping track of when each data element in a set was used relative to the other elements. For a direct mapped cache there is only one potential entry to evict, but for a set associative cache there are N potential entries to evict and for a fully associative cache any item in the cache can be evicted. Because of this, with increasing associativity, there is an increased hardware requirement and complexity to implement the given replacement policy [70].

In Xilinx FPGAs, BlockRAM is a dedicated two-port memory structure containing several kilobits of RAM that include other features such as output registers to increase pipeline performance. BlockRAMs are used for efficient data storage or buffering, for high-performance state machines or FIFO buffer, for large shift registers, large look-up tables, or ROMs. BlockRAMs are placed in columns through the FPGA fabric and their total number depends on the size of the device [104]. With the majority of the BlockRAM on the target platform unused, they are an ideal primitive in which to implement a cache.

Initial evaluation of the platform, detailed later in Section 8.1, shows that when the number of traversal units is scaled, memory is the bottleneck that limits increasing performance. While this problem may be solved with a generic direct-mapped or set-associative cache, with knowledge of the data structure being cached and the flexibility of FPGAs, there may be information available at run-time that can be used to implement a more efficient application specific cache. In this chapter, memory analysis during ray tracing of four test scenes is presented in Section 7.1. The analysis compares the difference in memory requirements between nodes and primitives and the
memory requirements of different levels of the acceleration hierarchy. Using this analysis, the design, implementation and evaluation of node and primitive caches for FPGAs is then presented in Section 7.2. The design, implementation and evaluation of a novel replacement policy is also presented in Section 7.3. The effect of the node and primitive caches on overall platform performance is presented later in Section 8.2.

### 7.1 MEMORY ACCESS ANALYSIS

In this section memory access patterns and bandwidth requirements are analysed during ray tracing for different types of data and different scene configurations to determine what types of data or areas are the most efficient for caching. Three memory bandwidth areas are explored: node bandwidth relative to primitive bandwidth per ray, the effect of leaf size on bandwidth and node bandwidth per level of the acceleration hierarchy.
Table 7.1: Scene statistics.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Primitives</th>
<th>Nodes</th>
<th>Max. Tree Depth</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kitchen</td>
<td>86,032</td>
<td>26,381</td>
<td>15</td>
<td>640 × 480</td>
</tr>
<tr>
<td>Classroom</td>
<td>2.47 · 10^5</td>
<td>80,356</td>
<td>15</td>
<td>640 × 480</td>
</tr>
<tr>
<td>Hairball</td>
<td>2.88 · 10^6</td>
<td>9.96 · 10^5</td>
<td>17</td>
<td>960 × 540</td>
</tr>
<tr>
<td>Sponza</td>
<td>66,450</td>
<td>22,585</td>
<td>12</td>
<td>960 × 540</td>
</tr>
</tbody>
</table>

### 7.1.1 Scene Setup for Analysis

Four test scenes with different sizes and characteristics were selected from freely available models: Kitchen, Classroom, Hairball and Sponza [54]. The Kitchen and Classroom scenes form a closed environment, where the camera is placed within the scene, while Hairball and Sponza scenes form an open environment. All scenes use the Metropolis-Hastings algorithm for sampling [46] with a maximum ray path depth of five, i.e. for each primary (coherent) ray that intersects with a primitive, the incoherent ray paths generated have a maximum depth of five and the average coherency between all rays will be low. Renders of these scenes are shown in Figure 7.1 and scene statistics using QBVH and a maximum of four primitives per leaf are given in Table 7.1. Node and leaf memory accesses are recorded for each test parameter (which are explained in the following sections) over ten seconds for each scene for as many rays as could be processed in this time. The memory requirement of a node and triangle for the given test settings are used as the node and triangle intersection costs, respectively, for the SAH during tree construction.

### 7.1.2 Node versus Leaf Bandwidth

The bandwidth required for a node access is 128 bytes while a leaf is 64 bytes times the number of primitives in the leaf. Together with the number of node and leaf accesses these values produce the average bandwidth for nodes and leaves per ray and the total average bandwidth required per ray. The maximum size of the leaves of the tree were varied by changing the maximum primitives per leaf (MPPL) setting during tree construction. This means, for example, that for an MPPL setting of four, even if a split is more expensive (using the SAH) than testing for intersection with each primitive, the
primitives must continue to be split until there are no more than four primitives in each leaf. Once the MPPL threshold is met then the SAH is used to determine when to stop splitting and declare a leaf. Data was collected for each scene for different MPPL settings and the results are shown in Figure 7.7.

For scenes Kitchen, Sponza and Classroom the results show that an order of magnitude less bandwidth is required for primitives than for nodes. For Hairball the bandwidth required is also less, but only 1.5 - 3 times less. The results show that decreasing the MPPL increases the total node bandwidth required. This is because a taller acceleration hierarchy is created, requiring more node accesses to reach leaves. However, it also decreases the total bandwidth required for leaves. This is because, once a leaf is reached, the probability of intersection with a primitive within the leaf is higher, meaning fewer total leaves need to be accessed. The total bandwidth required per ray is qualitatively the same for all MPPL settings over Kitchen, Sponza and Classroom, with an MPPL of two requiring slightly less bandwidth than the other settings. The Hairball scene favours an MPPL setting of four or more, with a setting of one or two requiring more bandwidth than a setting of four, eight or sixteen.

7.1.3 Level Effect on Node Bandwidth

With significantly more bandwidth required per ray for nodes, memory accesses by their level in the acceleration hierarchy are now analysed. With a MPPL of four being a good choice over all scenes, a dump of the acceleration hierarchy and the same set of node memory accesses for MPPL = 4 are used to present the data in Figures 7.2 - 7.5. Going left to right then top to bottom for each figure, the first sub-figure shows the distribution of nodes by level. The second shows the distribution of node accesses by level and the third shows the accesses on each level divided by the number of nodes on that level. The fourth and last sub-figure shows an accumulated total of nodes and accesses, for the given level and those above it, as a percentage of total nodes and accesses respectively.

All scenes show the number of nodes increase initially when moving down the tree, peak just after half-way down the tree then decrease. Any value less than \(4^L\) indicates the level is not full. All scenes show that their hierarchy loses density when moving down, with Hairball retaining density the longest, fully populated in the
upper half of its hierarchy. This trend is because the hierarchy is built in a top down fashion, splitting the scene primitives as construction continues down, and with more primitives still unplaced while in the higher levels there are more opportunities to find a split where the SAH cost is less than interesting all primitives.

Node accesses by level show a quick increase to a peak and then decrease moving down the hierarchy. For all scenes the peak in node
Figure 7.3: Classroom scene analysis.

accesses by level occurs before the peak in the number of nodes by level.

The accesses on each level divided by the number of nodes shows the average number of times the nodes on a given level are accessed. Over all scenes, this shows that nodes on higher levels are accessed, on average, more frequently than lower levels and nodes on the top 4 levels are accessed significantly more than lower levels. From a caching perspective, this means that higher level nodes have greater
temporal locality of reference than lower nodes. In the following section a cache replacement policy is explored that exploits this temporal locality by replacing nodes in the cache only when they have the same or higher level.

The accumulated total of nodes and accesses by level shows the effect of the peak in node accesses occurring higher in the tree than the peak in the number of nodes. For example, caching all nodes down to level 4 would require caching a maximum of 341 nodes. On Kitchen, this represents 1.3% of the total nodes but 62% of total
accesses. For Classroom, this represents 0.8% of total nodes but 73% of total accesses and for Sponza, it represents 3.0% of total nodes but 68% of total accesses. As Hairball is a much larger scene, this represents only 0.07% of total nodes but still 19% of total accesses. This shows it is possible to cache a small number of nodes and achieve a high hit rate.
7.2 CACHES FOR RAY TRACING

The following subsections outline the design, implementation and evaluation of the node and primitive caches.

7.2.1 Design

The node and primitive caches will naturally sit in the Memory Controller in Figure 4.2, intercepting requests bound for memory and fulfilling them if they contain the required data. Requests will be sent to their respective caches using the most significant bit of the request representation shown in Figure 4.8 and Figure 4.9, with 0 indicating a node request and 1 indicating a primitive request. The presence and size of each cache will be programmable, meaning different sizes of each cache can be implemented or a cache can be left out completely to analyse differences in performance. In the case that either the node or primitive cache is not present, the respective requests will be directed straight to memory. The cache will be direct mapped, simplifying implementation and logic requirements by requiring only one row be compared with incoming requests [70].

It is also important to note that the purpose of the cache on the platform is to reduce the bandwidth required per ray, not to hide latency as multiple threads of computation are used to do this. Any increase in latency that a cache miss will introduce may increase the total time taken for any single ray, but a decrease in bandwidth utilization will allow other traversal units that would otherwise be waiting on memory requests to be processed, increasing overall throughput. This is in contrast to typical cache designs for general purpose processors, which are strongly concerned about the latency of transactions due to the slow access time of memory relative to the clock speed of the processor.

7.2.2 Implementation

The node and primitive caches are implemented as standard direct-mapped caches [70]. The node cache is shown in Figure 7.6 and the primitive cache is the same but with a different data width. They are both implemented in VHDL, instantiating BlockRAM as storage for both tags and data and using generate and parameter statements to enable the cache to be of configurable size during
Figure 7.6: Node cache implementation.

synthesis. Inputs to the cache are requests from the intersection controller and data returned from the on-board memory controller. Requests are 31 bits instead of 32 bits as the node/primitive identifier at bit position 32 can be removed after routing the request to the relevant cache. Outputs are addresses to the memory controller when a miss occurs and data for the intersection pipelines once a request has been fulfilled from either the cache or on-board memory. The critical path of the implementation is the BlockRAM, meaning the implementation is capable of operating above the 150MHz required by the platform.

The target settings for the node cache are QBVH, meaning a node row data width of 896 bits. For the primitive cache there are 288 bits per primitive, but the total row data width is adjusted based on the MPPL setting. The flexibility of FPGA BlockRAM primitives means that if the total space provisioned for the primitive cache can fit 4,096 primitives, then this space can be configured for any integer combination where $W \times D = 4096$, where $W$ is width (or MPPL) and $D$ is depth i.e. possible configurations are a $1 \times 4096$ cache, a $2 \times 2,048$ cache or a $4 \times 1,024$ cache and so on. For example, when the total space is capable of containing 4,096 primitives and an MPPL setting of four is used, the primitive cache will have 1,024 rows with a data width of 1,152 bits (four primitives of 288 bits each).

For a cache of size $N$, the lower $\log_2 N$ bits of the request are used as the index to the cache and are connected to the address input of the BlockRAM, and the remaining bits ($31 - \log_2 N$) are used as the tag, which is compared to the tag in the current data line to determine if the current data is a hit or miss. For example, the largest power-of-two size node cache that can fit on the target platform is 8,192, meaning the first 13 bits are used as the cache index and the remaining 18 bits are used as the tag.
7.2.3 Evaluation

The memory access analysis showed that, per ray, significantly more bandwidth is required for nodes than for primitives. For this reason the largest node cache that can fit on the platform is exclusively implemented initially and the primitive cache will be included only if the evaluation shows that primitive bandwidth dominates the remaining node bandwidth requirement.

For evaluation, the same scenes and memory access patterns are used from Section 7.1.1 over different MPPL settings. A cache with the maximum size that can fit on the target platform is simulated...
Figure 7.8: Bandwidth per ray vs Max prims per leaf with Node Cache
Figure 7.9: Bandwidth per ray versus Maximum primitives per leaf with Node and Prim Cache
using Modelsim for the given access pattern and hit and miss rates are recorded. As a request miss in the cache must be fulfilled by memory, the miss rate represents the resulting node bandwidth requirement. As the primitive cache is not included in simulation, the primitive bandwidth requirement is the same as the results from Figure 7.7.

The results showing the node, primitive and total bandwidth requirements when only the node cache is implemented are shown in Figure 7.8. The results show, when compared to Figure 7.7, that the node cache is extremely effective for Kitchen, Sponza and Classroom for all MPPL settings, decreasing the required node bandwidth by over an order of magnitude on all settings and up to over 75 times decrease for Kitchen, up to over 50 times decrease for Classroom and up to over 28 times decrease for Sponza. The cache is also effective for Hairball, decreasing the node memory bandwidth requirement over all settings, but not as effective as the other scenes, with the maximum decrease being just over 3.7 times.

The node memory bandwidth requirement in Figure 7.8 shows the same tendencies as Figure 7.7 with the bandwidth requirement decreasing as MPPL increases. This is true for all scenes apart from the MPPL setting of 16 for Kitchen, where there is an increase relative to the setting of 8. This increase is due to the different structure and memory layout of that particular tree creating cache rows with more frequent thrashing, increasing the miss rate relative to the other settings.

With the exclusive node cache, the primitive bandwidth requirement now dwarves the node bandwidth requirement. Sacrificing some of performance of the node cache is now explored by making it smaller and using the freed BlockRAM to implement a primitive cache. This is done by halving the size of the node cache to 4,096 and then implementing the largest primitive cache possible in the remaining BlockRAM, which can fit 4,096 primitives, and setting the width of each cache line to the current MPPL setting. This means that the cache in each simulation can hold the same number of primitives (MPPL times the number of rows), but the effect of the “fullness” of each row for each setting - i.e. the number of primitives in a leaf compared to the MPPL - and the different number of cache lines may effect the miss rate.

The results showing node, primitive and total bandwidth requirements with the node and primitive cache implemented are
shown in Figure 7.9. The results show, when compared to Figure 7.8, that the primitive cache is effective for Kitchen, Sponza and Classroom for all MPPL settings, decreasing the required primitive bandwidth over all settings. However, the primitive cache shows little improvement for Hairball, where there is only a slight decrease in primitive bandwidth. The results also show that, as expected, decreasing the size of the node cache increases node bandwidth requirement. However, for Kitchen, Classroom and Sponza this increase in node bandwidth is smaller than the decrease in primitive bandwidth, meaning lower total bandwidth is required, which is the main objective. For Hairball the increase in node bandwidth is greater than the decrease in primitive bandwidth, meaning using the largest possible node cache has a lower total bandwidth requirement than a smaller node cache and a primitive cache, but only slightly.

7.3 Level Replacement Policy

Looking at the accesses per node per level from Section 7.1, it is clear that nodes on higher levels are accessed, on average, more frequently than lower levels, meaning they have a greater temporal locality of reference. In this section a cache replacement policy that makes use of this observation is investigated, not replacement based on recent use but instead based on the nodes level in the acceleration hierarchy, called the Level Replacement Policy (LRP). The following subsections outline the design, implementation and evaluation of the LRP.

7.3.1 Design

The cache implemented in the previous section is direct-mapped, meaning that for every request value, there is only one possible cache location to locate the data. Whenever there is a cache miss, the new data replaces the existing data before the next request is processed. Increasing the associativity of a cache means increasing the number of locations data can be placed (or “associated”), but also means that to check if data is a match it must be compared with more locations. To increase performance this checking is often done in parallel, adding the logic cost of extra comparators, otherwise it has to be done over multiple cycles, decreasing the throughput of the cache [70].
This also raises the question: When all of the multiple places in which data can be stored are occupied, which location should be replaced? The most commonly used “replacement policy” is least recently used (LRU), where the data replaced is the one that has been unused for the longest time relative to the other data items. Extra logic must be implemented to keep track of such information, and as associativity increases, so does the required logic and complexity for replacement schemes [70].

Looking at the accesses per node per level from the Section 7.1, it is clear that nodes on higher levels are accessed, on average, more frequently than lower levels, meaning they have a greater temporal locality of reference. Using this observation, the LRP was designed to give nodes on higher levels of the acceleration hierarchy higher priority over those on lower levels. To achieve this, when a cache miss occurs, the node data returned from memory only replaces the node data currently in the cache if it is on the same or higher level. This allows the simplicity of a direct-mapped cache to be maintained with only minor logic additions required to compare node levels to implement the LRP.

7.3.2 Implementation

In order for the level of each node to be compared it must be made available during each request. This is done by encoding the level of the node in the same way the number of primitives in each leaf is encoded, using four upper bits of the address, shown in Figure 4.9. The new node encoding, shown in Figure 7.10, allows us to encode up to 16 levels of each scene. Using these bits for node encoding rather than the node address decreases the number of nodes that can be represented by a factor of 16 to the same as the number of primitives that can be represented, but as there are less nodes than primitives for all of our scenes, the number of primitives that can be represented will remain the limitation for scene size. The impact of the LRP on resource utilization and clock frequency is negligible, as only a single 4-bit comparator is introduced and it is not in the critical path of the cache. During tree construction, levels 0 to 14 are encoded as is,
from 0x0 to 0xF respectively, and nodes on level 15 or below have an encoding of 0xF.

Also included in the implementation is a 4-bit configuration register, MAX_LEVEL, to limit the depth to which the level replacement policy is active. For example, a MAX_LEVEL setting of 0x5 means the level replacement policy is only active for levels 0x5 and above, and on all levels below the nodes are considered to have the same priority.

The policy works like so:

- When a cache miss occurs, the node is requested from memory using the node address in the node representation in Figure 7.10.

- When the missed node data is returned from memory, the level fields of the missed node and existing node in the cache line are compared and the existing node is only evicted if the new node is on the same or higher level of the acceleration hierarchy and on the same or higher level as MAX_LEVEL.

### 7.3.3 Evaluation

To evaluate the LRP, Modelsim is used to simulate a range of MAX_LEVEL values over different cache sizes for each scene and compare the miss rate with a direct-mapped cache. Only MAX_LEVEL values up to six are included, as after this value performance degrades and no improvement was shown with any greater value over all scenes. Also included, is the miss rate of a fully-associative-by-level cache, where the cache stores nodes based on their level and is guaranteed to contain the N highest nodes in the hierarchy. This can be achieved with the breadth first technique used by Nah et al. [61], where tree construction is changed from depth first to breadth first and the cache stores the first N nodes based on their index. The results are shown in Figure 7.11.

The LRP shows an improvement in the miss rate for small cache sizes and the miss rate approaches the same performance as the direct-mapped cache as cache size increases. On scenes where the miss rate for the breadth first cache remains worse than that of the direct-mapped cache for more cache sizes, the LRP also remains worse for more cache sizes. Investigating LRP cache lines where performance was inferior compared to their counterpart direct-mapped cache lines revealed seldom accessed nodes on higher levels.
hogging the cache line and blocking a more frequently accessed lower level node with the same cache index.

Overall, the results show that the LRP can provide a good performance increase for small cache sizes and may work well as a complimentary cache along side a direct mapped cache. For larger cache sizes the LRP cache shows little or no improvement over a direct mapped cache so the overhead of implementing LRP would not be worth the effort.
Figure 7.11: Cache Size versus Miss Rate
Ray-tracing performance can be measured in many ways depending on the application or area of research. For example, real-time ray-tracing is where a scene or environment is rendered over and over and presented as frames for applications such as video games. This allows the scene to be moved around in, or interacted with, and the changes to appear in (or near) “real-time”. The performance of real-time ray-tracing is therefore measured in the number of frames produced per second (FPS), with more frames per second equating to more responsiveness.

As ray-triangle intersections are an integral part of ray-tracing and take up a significant amount of computation time, implementations that aim to accelerate this process measure performance in ray-triangle or ray-object intersections completed per second. However, as ray-triangle intersections are not the only part of ray-tracing, and as they are often traded for traversal of an acceleration hierarchy, this metric is poor for measuring overall performance in modern ray-tracing. On mobile devices, for which ray-tracing has recently become a research focus [47, 48, 51], power efficiency is a concern, as lower power usage means a device can run on a battery longer before running out of energy and needing to be recharged. While current research favours ASICs for implementing ray-tracing for mobile platforms, it may be beneficial to instead include an FPGA as a general purpose accelerator, reconfiguring it on demand to accelerate different tasks or applications. As mentioned previously, power costs and air-conditioning costs (which are proportional to the amount of power dissipated) can be over 50% of the cost of running a render farm [74], so power efficiency is also important for attached hardware accelerators. Power efficiency during ray-tracing is measured in rays per Joule, taking into the account the speed rays can be traced but also the average power required to do so.

For high-quality rendering, such as that required by movie makers, it can take hours to render a single frame, meaning measurements in FPS would be several orders of magnitude below 1 and less than ideal to compare. Also, factored into measurements in FPS is the
resolution of the frame and the number of samples (or number of rays traced) per pixel, making it hard to compare performance between high-quality and mobile ray-tracing of the same scene as these values differ significantly. In these cases, and as used in this chapter, a more appropriate measure for performance is the number of rays traced per second, or rays per second (RPS), calculated by timing the rendering of the frame and dividing the number of rays traced in the frame by this time.

Comparing performance (or other related metrics) of different ray-tracing implementations by using different scenes is hard to do and often inaccurate, as the different memory access requirements for different models, or even different orientations of the same model, can easily impact performance. It is therefore only truly objective to compare implementations using the same scenes. This poses a problem when comparing the presented platform with other research, as although there is a common set of models available and used in most research [85], the scene set-up (including lighting and camera orientation) is either not available or not available in a compatible format. For this reason, the main focus for comparison in this chapter is with other devices where the exact same scenes can be rendered and only general comparison is made with previous research performance.

The scenes used for performance evaluation in this chapter are those introduced previously in Section 7.1.1: Kitchen, Classroom, Hairball and Sponza. Renders of these scenes are shown in Figure 7.1 and scene statistics using QBVH and a maximum of four primitives per leaf are given in Table 7.1.

To measure platform performance for a given scene, it is rendered on a certain platform configuration for at least ten seconds and the number of rays processed in that time is counted. Ten seconds is a sufficient time to allow throughput to stabilise while allowing the completion of the many required tests in a reasonable time. Longer rendering time improves the quality of the image, but does not have a significant impact on the throughput measured. Rays are bundled up into groups of 1,024 and then sent to the FPGA for intersection. Varying the number of rays gathered per bundle does not have any affect on the platform or software performance, but increasing this number may provide benefits when the PCIe express bandwidth becomes a bottleneck to the platform. If a ray bundle has not completed processing when ten seconds is reached, the render
time measurement is extended until it is completed. The total number of rays processed divided by the total render time gives the number of rays processed per second. This process is repeated for all scenes and over all possible platform implementations that are being compared. Other performance factors that need to be measured, such as memory bandwidth utilization, are explained in the sections in which they are required.

In this chapter, the difference in performance and scalability between the single-cycle queue and heap, presented earlier in Chapter 6, is compared in Section 8.1. The impact on performance of implementing a direct-mapped cache in BlockRAM to decrease memory bandwidth requirements, discussed earlier in Chapter 7 is presented in Section 8.2. Finally, to put the proposed platform performance into relation, performance, power consumption, and performance relative to available bandwidth is presented in Section 8.3.

8.1 **Effect of Queue Type on Performance and Scalability**

The most simple implementation of the proposed platform would contain the minimum logic to trace a single ray at a time. That is, a single traversal unit and single ray-triangle and ray-box intersection units, representing a SIMD system, tracing a single ray but intersecting with multiple bounding boxes or triangles at a time. When tracing a single ray, only one intersection request could be active at a time, meaning at most a single set of intersection tests can be performed at a time. This would dictate a maximum of four stages (for four bounding boxes) of the 40 stage ray-box intersection pipeline or four stages (for four triangles) of the 84 stage ray-triangle intersection pipeline would be active at any one time. The first most obvious way to increase performance of the platform is to increase the efficiency of the intersection pipelines, achieved by creating more intersection requests to try and fill all stages of the pipelines.

Increasing the number of intersection requests created is done by adding more traversal units, which each require a priority-queue to store intermediary intersection results. In this section the number of traversal units is scaled for traversal units containing single-cycle queues and heaps to measure their effect on performance. The
maximum number of each type of queue that can fit on the target device, a Virtex-5 LX330T-2, will also show the relative scalability of the queue types. The two queues investigated are the single-cycle queue and the heap implementations from Section 6.3.1 with the heap including the optimisations made in Section 6.4. It is expected, without any bottlenecks in any other areas, that performance will scale linearly when increasing the number of traversal units, up until the intersection pipelines are full. The single-cycle queue performance advantage over the heap for push and pop operations should also translate into a ray-tracing performance advantage for the same number of queues, as heap maintenance will delay the sending of intersection requests for the heap implementation and increase the time taken per ray.

The number of each type of queue was scaled until the device was either full or it was hard to meet timing requirements and performance was measured. Optimisations were not attempted to meet timing after the target device was 80% full as the returns in performance for the amount of time spent diminish significantly past
this point. The results are shown in Figure 8.1. The results show that only six single-cycle queues could fit on the target device, while 32 heaps could fit, easily explained by the logic utilization difference between the two queues shown previously in Table 6.4.

The results show performance scales very well from two to four traversal units and is still good from four to eight traversal units. For more TUs performance completely plateaus. The Hairball scene had the lowest overall performance, which happens to be the scene with the most triangles. Despite such differences, the scalability for all scenes is qualitatively the same. The single-cycle queue is measurably better than the heap for lower numbers of TUs but this is noticeable for the Hairball scene only. The difference is small, however, and the poor scalability prevented going beyond six traversal units to take further benefit from it. It seems clear that the heap, while slower but more scalable, is the better choice for the proposed platform implementation.

At this point, it is evident from the plateau in heap performance that there is a bottleneck elsewhere in the platform limiting performance, as the intersection pipelines are not yet full. Measuring the utilized memory bandwidth during ray-tracing (shown in the following section on the “without cache” plots in Figures 8.2 – 8.5) indicated that memory bandwidth was limiting performance. This led to the detailed investigation of an efficient object cache to alleviate the memory bottleneck, detailed earlier in Chapter 7 and evaluated in Section 8.2.

8.2 CACHE IMPACT ON PLATFORM PERFORMANCE

With the results in the previous section showing a plateau in platform performance when increasing the number of traversal units, memory bandwidth was investigated as the bottleneck to performance. To measure the memory utilization of the platform during ray tracing, a probe was placed on the valid signal of the memory interface, which indicates whether the data has been successfully read. The probe is connected to two counters; one which increments when the valid signal is low, indicating the memory interface is inactive, and one which increments when the valid signal is high, indicating the memory interface is active. When either of these signals is about to overflow, further incrementing is blocked and the values from both
are read using ChipScope [101]. The ratio of the memory interface being active to inactive gives the memory utilization during ray tracing. It is expected that if memory utilization is the bottleneck to performance then it will show the same tendencies as the platform performance over all scenes. The performance and corresponding memory utilization is measured for each scene and results are shown in Figures 8.2–8.5 as the “without cache” plots.

The results show that on all scenes scaling the number of traversal units without a cache provides no improvement on performance, as memory utilization is limited to just above 50%. While the theoretical maximum transfer rate for the FPGA development board
on-board memory is 1.2 GB/s, in practice, it can not be sustained for real-world workloads. This is because row-address conflicts, data-bus turnaround penalties and write recovery all degrade the peak transfer rate [45]. For a ray tracing workload on this platform the upper bound of memory utilisation appears to be around 50% of the theoretical peak. As the platform performance and memory utilization show identical tendencies over all scenes, it is confirmed as the primary limiting factor in platform performance. This conclusion led to the memory analysis and cache design presented previously in Chapter 7. The impact of a cache on platform performance as discussed in Chapter 7 is now evaluated. It is expected that
introducing an effective cache will decrease the memory utilization and allow performance to scale as the number of traversal units increases.

In the cache evaluation shown in Figure 7.11 from Section 7.3.3 there is a red line marking the “largest cache for device”. This value represents the largest power-of-two sized cache that can fit in the BlockRAM on the target platform. For the Virtex-5 LX330T-2 used for the target platform, this value equates to a cache size of 8,192 lines. Over all scenes, at this point on the graph, the LRP caches show no improvement over the direct mapped cache. For this reason a direct-mapped cache has been implemented to investigate the impact on memory utilisation and overall ray tracing performance on the platform. The RBI pipeline from Section 5.2.2 is also duplicated to handle the increased throughput provided by the cache. The per scene performance and memory utilization measurements were repeated as above but with the cache implemented on the platform. The results for which are shown in Figures 8.2 – 8.5 as the “with cache” plots.

The results show that scaling the number of traversal units with a cache provides an increase in performance on Kitchen, Classroom and Sponza. Memory utilisation also increases with the number of traversal units but is lower than without a cache and does not plateau or reach the apparent 50% limit as it does without a cache. For these scenes, it can be said that memory bandwidth is no longer the bottleneck for absolute performance and a larger FPGA or a more logic efficient traversal unit may be used to further increase performance. Additionally, the increase in performance and memory utilisation is not linear, meaning there are other factors reducing the effect of scaling the traversal units. Factors such as the pipeline latency of the intersection units and the throughput of the unit that presents requests to the cache from the traversal units, which is currently limited to one request every two cycles, are the next areas to explore for increased performance.

On Hairball, there is also an overall increase in performance but scaling after 16 traversal units has no effect on performance. At this point and onwards, the same memory limitation is reached as was for when there was no cache present. This is due to the large bandwidth requirement per ray for this scene, shown in Figure 7.8, and the poor miss rate for the maximum size cache on this scene, shown in Figure 7.11.
### Table 8.1: Process technology and approximate component cost for devices for evaluation during ray-tracing.

<table>
<thead>
<tr>
<th>Device</th>
<th>Process Technology (nm)</th>
<th>Approximate Component Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex-5 LX330T-2</td>
<td>65</td>
<td>3,000</td>
</tr>
<tr>
<td>Artix-7 XC7A200</td>
<td>28</td>
<td>450</td>
</tr>
<tr>
<td>Intel Core i5-3470</td>
<td>22</td>
<td>300</td>
</tr>
<tr>
<td>Radeon HD 5450</td>
<td>40</td>
<td>150</td>
</tr>
</tbody>
</table>

8.3 **Performance Compared to a CPU and GPU**

After evaluating the scalability of the platform and the effect of a cache on memory utilisation, the four test scenes are now rendered with a CPU and a GPU to put the results into relation with other architectures. The CPU used is a quad-core Intel Core i5-3470 @ 3.2GHz with 16GB of dual-channel DDR3-1600 memory [41]. The GPU used is an AMD Radeon HD 5450 with 1GB of 800 MHz DDR3 memory [5]. This GPU was chosen as it has the process technology closest to the FPGA that is able to support OpenCL 1.2 in the AMD Radeon range, while also having large amounts of memory bandwidth available and power consumption comparable to that of the FPGA. The process technology and approximate component cost of each device is shown in Table 8.1.

The approximate component cost shows a reference point for initial outlay of cost per device, but does not take into consideration factors such as “economies of scale”, which would be an influencing factor when setting up a large rendering farm. The component cost of the Virtex5 LX330T-2 is relatively high because it has reached end-of-life and is now no longer produced. The included Artix7 XC7A200 cost shows a more accurate reference for the typical price of an appropriate FPGA. When choosing a solution, this component cost must be balanced against any perceived cost savings over the lifetime of the solution due to increased power efficiency, which is discussed in Section 8.3.2.
Initially, in Section 8.3.1, the raw performance of the platform is compared against an unoptimised QBVH implementation and a fully optimised SSE QBVH implementation on the CPU using one and four threads and against a fully optimised OpenCL QBVH implementation on the GPU. Then, in Section 8.3.2, using the raw performance measurements plus the power consumption of each device their power efficiency is compared. Finally, in Section 8.3.3, using the amount of memory bandwidth available to each device, the bandwidth efficiency is estimated to give an indication of how well each architecture makes use of the bandwidth available to it.

8.3.1 Raw Performance

In this section the raw performance in terms of rays per second is compared for the FPGA platform, the GPU and all CPU implementations. Results for all scenes are shown in Figure 8.6. FPGA32 shows the results for the 32 traversal unit implementation from Section 8.2, CPU1 shows results for unoptimised QBVH on one thread, CPU1SSE shows results for fully optimised SSE QBVH on one thread, CPU4 shows results for unoptimised QBVH on four threads and CPU4SSE shows results for fully optimised SSE QBVH on four threads. GPU shows the results for the fully optimised OpenCL QBVH implementation.

The results show that the CPU implementations have the same tendencies across all scenes; CPU1 is around a quarter of the speed of CPU4 and CPU1SSE, which are both around a quarter of the speed of CPU4SSE. GPU shows similar results to CPU1SSE and CPU4 over Kitchen, Classroom and Sponza but lesser performance on Hairball where it degrades.

FPGA32 shows stable performance over all scenes. It is faster than CPU1 across all scenes, is faster than CPU1SSE and CPU4 over the Kitchen, Hairball and Sponza scenes and slower over the Classroom scene. It is slower than CPU4SSE over all scenes and slower than GPU for Classroom but faster over Kitchen, Hairball and Sponza. FPGA32 shows the least degradation in performance for the Hairball scene compared to the CPU and GPU implementations, with GPU showing the most degradation. This is due to the parallel operation and efficiency of the priority queues on the FPGA platform, where maintenance of the heap in the CPU implementation takes up more time for scenes where more intermediary results need to be stored.
Figure 8.6: Raw ray-tracing performance of the platform and different CPU implementations over all scenes.
<table>
<thead>
<tr>
<th>Device</th>
<th>Energy Requirement (W)</th>
<th>Measurement Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex-5 LX330T-2</td>
<td>12</td>
<td>Xilinx Power Analyzer</td>
</tr>
<tr>
<td>Intel Core i5-3470</td>
<td>77</td>
<td>Thermal Design Power</td>
</tr>
<tr>
<td>Radeon HD 5450</td>
<td>19.1</td>
<td>Thermal Design Power</td>
</tr>
</tbody>
</table>

Table 8.2: Power requirements for devices to evaluate their power efficiency during ray-tracing.

and can not be over-lapped with other tasks like it can on the FPGA. FPGA32 shows performance an order of magnitude slower than that presented by Lee [50] for scenes of similar sizes, however FPGA32 uses an older generation FPGA and has 18 times less logic and 16 times less memory bandwidth available, making the performance comparable if these factors were scaled.

8.3.2 Energy Efficiency

In this section the energy efficiency of the FPGA platform, the most optimised CPU configuration (CPU4SSE) and the optimised OpenCL GPU implementation is compared. The power requirement and its measurement basis is shown for each device in Table 8.2. The CPU power usage is estimated at 77W and the GPU power usage is estimated at 19.1W, both using the Thermal Design Power (TDP). The TDP of a device is the maximum amount of heat energy generated by the device that can be dissipated by the cooling system during typical operation. The TDP is an accurate power consumption value for the CPU and GPU during ray-tracing as the repeated use of SIMD instructions require more power than other instructions (by using more of the instruction pipeline and registers) e.g. a CPU must often decrease operational frequency when executing many SIMD instructions to remain within the TDP [43].

The FPGA power usage is estimated at 12W using the Xilinx Power Analyzer (XPA). XPA does power analysis on design data of an implemented FPGA design, using the implemented netlist, an accurate power model of the chosen device, and a simulation activity file as inputs. The simulation activity file gives specific switching information (toggle rates, signal rates, and frequency information)
which combined with the device utilization and power model give an accurate power estimation for the device [107]. The value given in Table 8.2 is estimated using the FPGA32 implementation files and a simulation of the rendering of Classroom as it represents a typical scene.

As all of the compared components require an entire host system to operate, this factor is assumed to be “constant” and is not included in the power measurements for each approach under the following scenarios: When testing the CPU implementation, the CPU and main memory subsystem are the significant consumers of power, and when testing the GPU or FPGA implementation, the CPU and main memory are mostly idle with the GPU or FPGA boards being the significant consumers of power. The FPGA and GPU approaches do not include the power consumption of the idle CPU on the host, while the CPU approach does not include the power consumption of the memory subsystem, which is assumed in this thesis to balance the comparison. This constant factor of power consumption, which should be equal for all approaches though not included, should be kept in mind during the performance comparison below. The comparison below is only made between the variable factor, in the same way conventional processors and their power consumption are compared. It is also conceivable for the GPU and FPGA that the constant host power consumption would be amortized as additional accelerators were added to the system, tilting the power advantage towards them.

The power efficiency of each architecture is measured by repeating the performance measurements for each scene, as completed previously in Section 8.3.1, and dividing the result by the device power requirement. The resulting units for these values is rays per second per Watt or rays per Joule, showing the number of rays that are processed per unit of energy for each device over each scene, with more rays processed per unit of energy desirable for power efficiency. The results are presented in Figure 8.7. FPGA32 shows the power efficiency for the 32 traversal unit implementation from Section 8.2, CPU4SSE shows the power efficiency for fully optimised SSE QBVH on four threads on the CPU and GPU shows the power efficiency for the fully optimised OpenCL QBVH implementation on the GPU. The results show that FPGA32 has better power efficiency than both CPU4SSE and GPU over all scenes, with over twice the power efficiency than CPU4SSE on Hairball and Sponza, just
Figure 8.7: The number of rays traced per unit of energy on the platform and the most efficient CPU implementation over all scenes.
<table>
<thead>
<tr>
<th>Device</th>
<th>Memory Type</th>
<th>Available Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex-5 LX330T-2</td>
<td>DDR2</td>
<td>1.2</td>
</tr>
<tr>
<td>Intel Core i5-3470</td>
<td>Dual-channel DDR3</td>
<td>25.6</td>
</tr>
<tr>
<td>Radeon HD 5450</td>
<td>DDR3</td>
<td>12.8</td>
</tr>
</tbody>
</table>

Table 8.3: Memory type and available bandwidth for devices to evaluate their memory efficiency during ray-tracing.

under twice the efficiency on Kitchen and just under 1.5 times more efficiency on Classroom. It can therefore be said that, despite having lower raw performance than the CPU, the FPGA platform is a more power efficient solution than the CPU. The CPU4SSE and GPU power efficiencies are more closely matched, with GPU more power efficient over Kitchen, Classroom and Sponza and CPU4SSE more power efficient on Hairball where the raw performance of GPU degrades more than CPU4SSE.

It is worth noting that the process technology of the Virtex-5 FPGA is 65nm while that of the CPU is 22nm and the GPU is 40nm. Three generations of Xilinx devices have been released since the Virtex-5 (Virtex-6 at 40nm, Virtex-7 at 28nm and UltraScale at 20–16nm), each achieving more power efficiency than the previous generation through means such as reduced I/O power consumption and intelligent clock gating [105, 88]. With the older-technology platform power efficiency already better than a current generation CPU, use of a newer FPGA process technology will provide greater relative power efficiency.

8.3.3 Available Bandwidth

In this section the bandwidth efficiency of the FPGA platform, the most optimised CPU configuration (CPU4SSE) and the optimised OpenCL GPU implementation is compared. As CPU and GPU technologies have matured, they have been able to operate at frequencies that allow extremely fast and wide memory interfaces, e.g. enabling up to 320 GB/s on a Radeon R9 290 with GDDR5 memory [4]. The same cannot be said for FPGAs, where even custom built boards with many multiple channels of memory are still over an order of magnitude slower [62]. With memory earlier confirmed as a bottleneck to platform performance in Section 8.2 and also
commonly a bottleneck to performance on other SIMD and MIMD architectures [1], bandwidth efficiency gives an indication of how well an architecture makes use of the bandwidth available to it. A list of the devices tested and their available memory bandwidth is shown in Table 8.3, showing that the dual-channel DDR3 attached to the CPU provides just over 21 times the memory bandwidth than is available to the FPGA. But for an application where memory is often the bottleneck, does this equate to 21 times the performance?

Bandwidth efficiency for each device is measured by repeating the performance measurements for each scene as completed previously in Section 8.3.1 and dividing the result by the bandwidth available to the given device. The resulting units for these values is rays/s per GB/s, or rays per byte of bandwidth, showing the number of rays that are processed per unit of available memory bandwidth, with more rays per second desirable for bandwidth efficiency. The results are presented in Figure 8.8. FPGA32 shows the bandwidth efficiency for the 32 traversal unit implementation from Section 8.2, CPU4SSE shows the bandwidth efficiency for fully optimised SSE QBVH on four threads on the CPU and GPU shows the bandwidth efficiency for the fully optimised OpenCL QBVH implementation on the GPU. The results show that FPGA32 has better bandwidth efficiency than CPU4SSE and GPU over all scenes, with over six times the bandwidth efficiency of CPU4SSE for Kitchen, five times the efficiency for Classroom, nearly eight times the efficiency for Hairball and over eight times the efficiency for Sponza. It can therefore be said that, despite having lower raw performance than the CPU, the FPGA platform is a more bandwidth efficient solution than the CPU, making better use of the bandwidth available to it. The bandwidth efficiency for CPU4SSE and GPU is closer matched, but CPU4SSE is slightly more bandwidth efficient than GPU over all scenes.

With the platform bandwidth efficiency better than a current generation CPU, use of a newer FPGA process technology will provide more and faster logic (and higher clock frequencies), and access to newer memory interfaces with higher bandwidth, showing great potential for memory scalability. The next generation of the Xilinx MIG IP (for Virtex-6 and newer devices) contains a dynamic memory reordering core which shows significant increases in memory bandwidth utilization for certain workloads [45] and should provide an increase to the apparent 50% maximum memory utilization currently shown. The upcoming Xilinx UltraScale FPGAs
Figure 8.8: Ray-tracing performance per unit of bandwidth available on the platform and the most efficient CPU implementation over all scenes.
also promise possible configurations with 240 GB/s peak memory bandwidth when combined with a new Hybrid Memory Cube serial interface and four million logic cells [106]. Given the difference in technology and available bandwidth, and the available and upcoming FPGA memory technology improvements, the performance results for the platform are encouraging. It is not unreasonable to expect the performance of the platform to scale close to linearly with new memory technologies and increased available memory bandwidth, as once other areas become bottlenecks to performance there will be ample logic available on newer devices to replicate or speed up these areas, i.e. duplicating intersection pipelines when intersection becomes a bottleneck.
Ray tracing is a computationally intensive task with many avenues of hardware acceleration. This thesis investigated how FPGAs can be used to efficiently accelerate the time consuming aspects of ray tracing. The key motivations to the research in this thesis were: to provide a platform to easily and efficiently explore parallelism in ray tracing on FPGAs, to decrease the productivity gap between the exponentially increasing amount of logic found on FPGAs and the number of man-hours taken to develop applications for them, to provide building blocks and optimisations that enable ray tracing efficiency on FPGAs and to demonstrate that FPGAs have a great potential for power and bandwidth efficiency when accelerating ray tracing.

The current state of the art of ray tracing hardware acceleration was explored, identifying areas for improvement and new areas to exploit parallelism. These areas lead to the proposal of an innovative FPGA ray tracing platform to easily and efficiently explore hardware acceleration of ray tracing on FPGAs. The platform maintains the same API as GPU platforms to aid software integration, is integrated with a modern, acceleration-focused and currently developed rendering engine, and enables easy generation of complete VHDL or Verilog test benches for the entire system or individual components using real ray-tracing data. The platform is also agnostic to different acceleration hierarchies, allows traversal of programmable widths of acceleration hierarchies, allows traversal of multiple rays at a time and intersection of each ray with multiple objects, and is modular where units need to be replicated or interchanged. These factors, combined with easy generation of the platform, fulfil the first objective of this research: to decrease the productivity gap between the exponentially increasing amount of logic found on FPGAs and the number of man-hours taken to develop applications for them.

The design and development of deep, parallel floating-point pipelines for ray-triangle and ray-box intersection tests were presented. The analysis of the data-flow of the respective intersection
equations was presented and optimisations developed to reduce the latency of the pipelines. The pipelines were implemented using the Xilinx floating-point cores, with the ray-triangle intersection pipeline having a latency of 89 cycles and the ray-box intersection pipeline having a latency of 44 cycles. Both pipelines were evaluated and showed potential for scalability and improved performance over other accelerated implementations.

A novel hybrid hardware combination of a single-cycle queue and a heap was presented that combines the speed of a single-cycle queue with the scalability of a heap. An innovative method to simulate different priority queue workloads was developed to evaluate queue performance under different conditions. The method showed that the hybrid queue provides performance slower than a single-cycle queue but faster than a heap, while logic utilization showed the hybrid queue to consume more resources than a heap but less resources than a single-cycle queue. The hybrid queue therefore is perfect for use where an application has to deal with a working data set of unknown size or a size with high variance, where a very large priority queue is needed to store the required data, i.e. graph search algorithms. Optimisations are made to the heap for workloads that require bursts of push operations, less-than floating-point comparators for priorities, and small heap sizes. These optimisations produce a highly efficient and scalable heap queue, a key component to the performance of the FPGA ray tracing platform.

Memory access patterns to the acceleration hierarchy and scene primitives during ray tracing of four typical scenes on the platform are explored and analysed. The extensive analysis reveals that nodes are more important to cache than primitives, and nodes on higher levels of the acceleration hierarchy are accessed far more frequently than those on lower levels, leading to development of a novel node cache replacement policy, based not on recent use, but on a node’s level on the acceleration hierarchy. Evaluation shows that the replacement policy can provide a good performance increase for small cache sizes and may work well in a complimentary cache along side a direct mapped cache.

The presented floating-point intersection pipelines, heap implementation and optimisations, and ray tracing cache architecture fulfill the second objective of this research: to provide building blocks and optimisations that enable ray tracing efficiency on FPGAs.
Extensive experimental evaluation of the proposed platform by ray tracing four typical scenes on real FPGA hardware was performed, including investigating the effect of priority queue type on performance and scalability, the impact of a cache on platform performance and the performance of the platform in relation to a CPU and GPU. The evaluation showed that scaling the number of traversal units with a cache provides an increase in performance over all scenes, with the cache successfully alleviating the memory bottleneck.

The evaluation shows that the FPGA platform has better power efficiency than both the CPU and the GPU over all scenes, up to two times more efficient on the Hairball and Sponza scenes. With the power efficiency of the older-technology platform already better than a current generation CPU, and three newer generation devices available each achieving more power efficiency than the previous generation, use of a newer FPGA process technology will provide greater relative power efficiency.

The evaluation also shows that the FPGA platform has better bandwidth efficiency than both the CPU and the GPU over all scenes, up to over eight times more efficient on the Sponza scene. Given the difference in technology and available bandwidth, and the available and upcoming FPGA memory technology improvements, the performance results for the platform are encouraging. It is not unreasonable to expect the performance of the platform to scale close to linearly with new memory technologies and increased available memory bandwidth, as once other areas become bottlenecks to performance there will be ample logic available on newer devices to replicate or speed up these areas using the flexibility of the platform, i.e. duplicating intersection pipelines when intersection becomes a bottleneck.

**Future Research Avenues**

It is clear that FPGAs show potential as hardware accelerators for ray tracing, with the presented platform enabling easy evaluation of newer FPGA technologies and ray tracing techniques an important stepping-stone to researchers in the future. Key areas of investigation include:

- Development of a formal model to describe the platform for the objective of optimising throughput. The platform throughput can be described as a function of the number of traversal units, available memory bandwidth (with or without a cache) and
Intersection pipeline speed with resource constraints described as inequalities. Without caching, a model could be developed to determine the number of TUs required to saturate a given memory bandwidth for a given scene. However, a complete model is not currently feasible due to missing information and a number of factors which need to be investigated:

- Performance and the effectiveness of caching are highly scene dependent (i.e. the cache hit rate and the average number and type of intersection requests required per ray). This means a general model could only be developed after first developing a model with a very large set of test scenes to statistically identify different scene characteristics related to different areas of performance (i.e. the number of primitives, the density of the primitives, the camera position, etcetera).

- When caching is introduced, the number of TUs can be increased to provide increasing performance, but this relationship does not appear to be 100% linear, potentially due to the priority encoded method used to service TUs, but this must be further investigated to determine the true cause.

- Scaling the number of TUs will eventually introduce limitations to operational frequency such as high fanout of input signals which, while not experienced for the FPGA evaluated, will make the modelling of FPGA resource processes and technologies difficult.

- What effect platform characteristics such as pipeline latency will have on performance once memory is no longer the primary bottleneck.

- Design and evaluation of custom ray tracing acceleration hardware: a PCIe express printed circuit-board incorporating a Xilinx UltraScale FPGA connected to multiple high-bandwidth memory modules. This would produce a hardware platform technologically on-par with modern ray tracing accelerators, like GPUs, to evaluate the platform performance with competitive memory bandwidth.

- Use of half-precision floating-point numbers for bounding-boxes and bounding-box intersection. This would reduce
memory bandwidth and decrease logic utilisation, but the effect of the loss of precision needs investigation.

• Implementation of a three-input adder in the floating-point vector dot product module to decrease latency.

• Simulation of the platform with an ideal, infinitely fast memory interface to identify areas of the platform that will become a bottleneck when memory bandwidth is no longer a concern.

• Implementation of ray tracing techniques such as packet tracing and the longest common traversal sequence to further decrease memory bandwidth requirements (but for coherent rays only).

• Rendering with a cluster of FPGA development boards and possibly multiple systems to demonstrate the scalability of the approach for a render farm.
BIBLIOGRAPHY


