Flexible Hierarchy Ray Tracing on FPGAs

Abstract—**Rendering programs use ray tracing to artificially create photo-realistic scenes that would normally be too dangerous, too costly or physically impossible to fabricate. Acceleration of the rendering process can be achieved through spatial or object hierarchy structures, which aim to restrict the number of expensive ray-object intersection calculations along a ray path by trading them for traversal of the structure. With extensive inherent parallelism, ray tracing benefits from GPU acceleration but may also benefit from the more flexible control flow and memory architecture available with FPGAs. We present a flexible FPGA based ray tracing platform capable of traversing varying widths and types of acceleration hierarchies to evaluate their efficiency. The platform consists of four main controllers for communication, traversal, intersection and memory. The platform interfaces with LuxRays, an open-source C++ renderer, over PCI*express* to transfer data for computation to onboard memory. The traversal controller uses a programmable number of priority queues which are implemented efficiently on FPGAs using a series of comparators and a shift register, allowing for single cycle insert and extract operations. We implement a four priority queue configuration of the platform at 250MHz that uses 26% of the registers and 25% of the LUTs available on the targeted device. The configuration showed promising results compared to CPU and GPU renders.**

I. Introduction

Digital rendering of special effects is now a routine requirement for moviemakers, allowing them to create scenes that would normally be too dangerous, too costly or physically impossible to fabricate but still giving a realistic look and feel to the film [1].

Digital rendering generates a digital image from a scene model using a number of different techniques. The scene is represented by a model (or collection of models) which is a digital description of a 3-dimensional object - often stored as a mesh of triangles or vectors. Along with the model, the full representation of a scene also describes the viewpoint (or camera location), lighting, texture and shading [2].

To create scenes for special effects, the model of the environment and the objects within it, is created in a digital rendering program. LuxRender, a physically correct and unbiased rendering engine, is one such program.

LuxRender implements ray tracing methods to create the photo-realistic digitally rendered images required by movie makers, but requires massive computational power, as a scene has to be rendered frame-by-frame at very high resolutions. Ray tracing can keep a several hundred server cluster occupied for months rendering a full-length animated movie.

Ray tracing follows a path of light from the camera through each pixel in the image plane and simulates the light interaction with objects in the scene. It 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 triangle meshes 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 calculates the total light contribution for each pixel, which is used to determine the colour [2].

While it may seem counterintuitive 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.

The inherent parallelism in ray tracing calculations has lead to much research in the area and several hardware acceleration techniques. Ray tracing using CUDA (Nvidia's support architecture for GPU programming) capable of $> 7.7 \times 10^6$ primary rays per second has been implemented [3]. More recently Nvidia developed the Optix ray tracing engine, a programmable system for GPUs and other parallel architectures. Optix builds on the observation that most ray tracing algorithms can be implemented using a small set of programmable operations [4]. Woop *et al.* implemented a Ray Processing Unit on a Xilinx Virtex-2 FPGA, which performed similarly to a CPU [5]. Cameron presented a method for using FPGAs to supplement ray-tracing computations on the Cray XD-1. Preliminary simulations estimated the systems performance to be 1.2×10^7 ray-object intersection tests per second [6]. Nery *et al.* present 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 [7]. The architecture was implemented on both FPGA and General Purpose GPU systems. The FPGA implementation achieved 1.6×10^5 ray-object intersection tests per second while the GPU system achieved 4.0×10^6 ray-object intersection tests per second.

This paper outlines the design and implementation of a generic FPGA ray tracing acceleration platform. The implementation aims to explore the use of the massively parallel and flexible capabilities of FPGAs as an alternative to the traditional use of GPUs. Although GPUs have excellent raw floating-point performance, they do not support flexible control flow and have an inflexible memory hierarchy limiting their efficiency for ray tracing. The FPGA design is aimed to be highly flexible and enable investigation of several different acceleration methods and configurations. A single configuration is implemented and tested for initial performance results and further optimizations are proposed.

The paper is laid out as follows. Ray-triangle intersection - a key ray tracing calculation - and its implementation in software are outlined. Acceleration hierarchies that quickly discard whole groups of objects during intersection are detailed. The platform, design and implementation of the FPGA acceleration

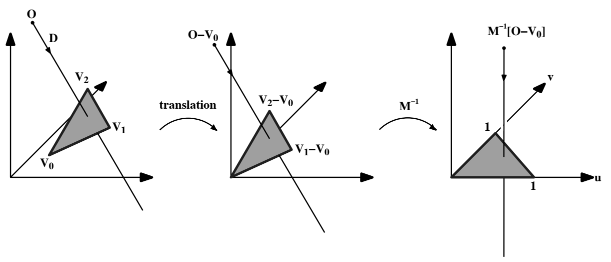

Fig. 1. Translation and change of base of the ray origin [8].

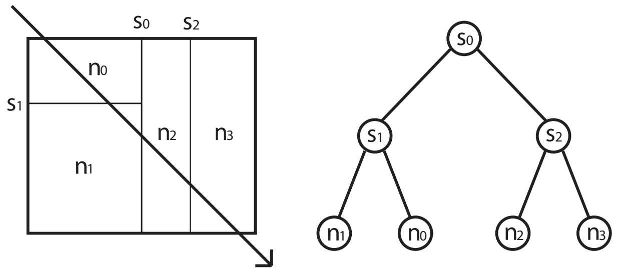

Fig. 2. A two-dimensional *k*D-tree and corresponding graph representation.

system are then detailed and followed by utilization and performance evaluation.

II. 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 [8] for RTI tests is commonly used in digital rendering programs, including LuxRender. The algorithm defines a ray, $R(t)$, by its point of origin, O, and normalised direction vector, $\vec{D}$, and a triangle by its three vertices, V_0, V_1 and V_2. O is the location of the camera lens, $\vec{D}$ is the direction from the lens through a pixel in the image and the triangle is part of an object's triangle mesh.

To calculate the intersection point between the ray and triangle, the triangle is translated so that V_0 is at the origin of the axes. Using the determinant of the combined direction vector and edge vector matrices from Equations (2a) - (2b), the triangle is then transformed into a unit triangle along the y and z planes, with the ray aligned along the x plane. The complete process is shown in Figure 1.

The result is a vector containing the distance, t, to the intersection and the coordinates, u and v, of the intersection, shown in Equation (1).

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot \vec{D} \end{bmatrix} \tag{1}$$

where

$$E_1 = V_1 - V_0 \tag{2a}$$
$$E_2 = V_2 - V_0 \tag{2b}$$
$$T = O - V_0 \tag{2c}$$
$$P = \vec{D} \times E_2 \tag{2d}$$
$$Q = T \times E_1 \tag{2e}$$

The coordinates of the intersection, u and v, are barycentric, *i.e.* an intersection has occured only when Equations (3a) - (3c) are satisfied.

$$u \geq 0 \tag{3a}$$
$$v \geq 0 \tag{3b}$$
$$u + v \leq 1 \tag{3c}$$

For a brute force calculation of a complex scene, checking every ray for intersection with every triangle, RTI tests take up to 95% of computation time [9]. The acceleration structures described in the following section are implemented to quickly discard whole groups of objects during the ray intersection process.

III. Acceleration Hierarchies

Acceleration of the rendering process can be achieved through spatial or object hierarchy structures. These structures aim to restrict the number of 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. This paper focuses on traversal, however construction is co-essential as a factor of traversal performance.

A. kD-Tree

A *k*D-tree is a binary space partitioning tree where every internal node of the tree represents a split in one of k dimensions. Each split creates two *half-spaces* for its given dimension and objects that lie within each *half-space* belong in the respective sub-tree.

A two-dimensional *k*D-tree and the corresponding graph representation, presented by Foley *et al.* [10], is shown in Figure 2. Internal nodes are labeled next to their split planes and leaf nodes are labeled inside their volume.

The following sections outline typical *k*D-tree construction and different traversal techniques with optimizations focused towards hardware acceleration. An efficient *k*D-tree traversal algorithm is important as it can represent 92% of total rendering time [11].

1) Construction: A *k*D-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, the highest quality *k*D-tree can be constructed using greedy cost optimization based on a surface area heuristic (SAH). The SAH estimates the cost of splitting the current volume on a particular plane according to the probability of intersecting the children created. Greedy decisions are made at each step to avoid this search becoming non-polynomial [11].

```
1:  function TRAVERSE($tree$, $ray$)
2:      ($tmin$, $tmax$) $\leftarrow$ $tree$.root.intersect($ray$)
3:      search_node($tree$.root, $ray$, $tmin$, $tmax$)
4:  function SEARCH_NODE($node$, $ray$, $tmin$, $tmax$)
5:      if $node$.is_leaf then
6:          search_leaf($node$, $ray$, $tmin$, $tmax$)
7:      else
8:          search_internal($node$, $ray$, $tmin$, $tmax$)
9:  function SEARCH_INTERNAL($node$, $ray$, $tmin$, $tmax$)
10:     $thit$ $\leftarrow$ $node$.intersect($ray$)
11:     ($near$, $far$) $\leftarrow$ order($node$.left, $node$.right)
12:     if $thit \geq tmax \vee thit < 0$ then
13:         search_node($near$, $ray$, $tmin$, $tmax$)
14:     else if $thit \leq tmin$ then
15:         search_node($far$, $ray$, $tmin$, $tmax$)
16:     else
17:         $stack$.push($far$, $thit$, $tmax$)
18:         search_node($near$, $ray$, $tmin$, $thit$)
19: function SEARCH_LEAF($node$, $ray$, $tmin$, $tmax$)
20:     for all $tri \in node$ do
21:         $thit$ $\leftarrow$ $tri$.intersect($ray$)
22:         if $thit < tmax$ then
23:             succeed($thit$)
24:     continue_search($node$, $ray$, $tmin$, $tmax$)
25: function CONTINUE_SEARCH($node$, $ray$, $tmin$, $tmax$)
26:     if $stack$.empty() then
27:         fail()
28:     else
29:         ($node$, $tmin$, $tmax$) $\leftarrow$ $stack$.pop()
30:         search_node($node$, $ray$, $tmin$, $tmax$)
```

Fig. 3. Recursive *k*D-tree traversal pseudo code.

2) Traversal: Pseudo code of a typical *k*D-tree traversal algorithm for ray tracing is shown in Figure 3. 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 stack (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 Figure 3, respectively.

Equations (4a) - (4b) show the calculations required to determine ray-box intersection on the x axis for an axis-aligned bounding-box (AABB) represented by $bmin$ and $bmax$. The process is repeated on the y and z axis and the largest $tmin$ and smallest $tmax$ values are determined. If $tmin$ is smaller than $tmax$ then intersection has occurred and $tmin$ is the distance to intersection. If a ray does not intersect with the AABB of a child node, it does not traverse it.

$$tmin.x = \frac{bmin.x - O.x}{\vec{D}.x} \tag{4a}$$

$$tmax.x = \frac{bmax.x - O.x}{\vec{D}.x} \tag{4b}$$

There are three methods to classify *near* and *far* nodes. The first uses the ray origin, the second uses ray direction and the third uses the coordinates of the ray intersection with the splitting plane. After 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 stack stores the *far* node when both need to be visited.

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 stack and traversal continues. If the stack is empty then traversal terminates without intersection.

The worst case performance of the algorithm for n leaf nodes is $\mathcal{O}(n)$, where the ray may visit a number of nodes linear to the size of the tree, shown in Figure 2. 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 $\mathcal{O}(\log n)$[12].

B. Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs), instead of splitting the scene space like *k*D-trees, recursively split the set of scene primitives until the leaf size (number of triangles) meets a given criteria. 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.

1) Construction: BVHs are constructed by recursively 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 *k*D-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.* [13] presented a technique that uses spatial splits, similar to those in *k*D-tree construction, to construct significantly more efficient BVHs than previous techniques.

2) Traversal: A BVH is traversed similarly to a *k*D-tree. Given a ray, a typical stack-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 stack. The routine is repeated by popping the next element, with the same criteria for completion as stack-based *k*D-tree traversal.

3) n-ary Bounding Volume Hierarchy: Dammertz *et al.* [14] presented n-ary Bounding Volume Hierarchy (nBVH) for construction and traversal efficiency on SIMD processors. SIMD width is exploited by increasing the arity of the acceleration structure, which also decreases the hierarchy memory requirements. Their approach flattens the hierarchy and favours larger leaves (more triangles per leaf) for more efficient streaming intersection of primitives. As the typical SIMD width

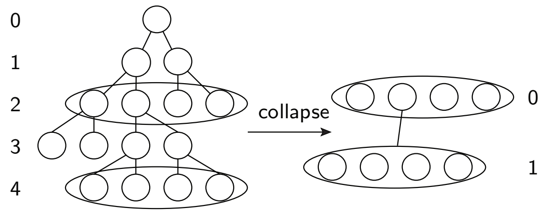

Fig. 4. Collapsing a binary tree to a QBVH. Classical build methods are used to create a binary tree, which is then collapsed by leaving out intermediate levels [14].

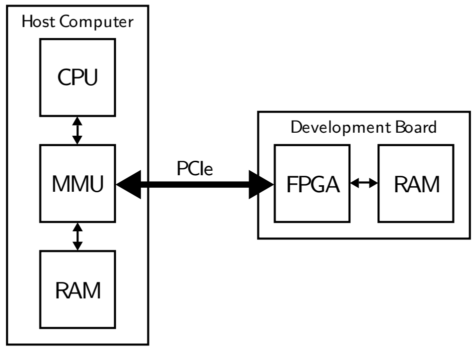

Fig. 5. The FPGA acceleration platform, attached to the host computer by PCI*express*.

of modern processors is four, their implementation is called QBVH (Quad-BVH), however the concept is not limited to this width.

A QBVH is formed by creating a classical binary BVH and collapsing two levels into one, as shown in Figure 4. Every second level of the tree is kept and the rest is merged, resulting in four bounding volumes per node, approximately halving the memory requirements. This method enables use of the same construction principles used for binary trees to construct n-ary trees. The resulting hierarchies build faster and smaller than previous structures while also outperforming them in traversal. Their QBVH implementation traverses 1.3 to 1.6 times faster than an efficient *k*D-tree.

IV. Hardware Acceleration Platform

The inherent parallelism in ray tracing makes it a promising target for hardware acceleration through use of a coprocessor. Production rendering acceleration systems use GPUs and offload processor intensive computations to the large number of floating-point pipelines that exist on these devices. Rendering software transfers ray tracing data to the device and waits for it to signal completion. The device takes the data, processes it and writes the results to the CPU attached main memory.

The target acceleration platform of this paper is a PCI*express* attached FPGA with onboard memory - the same interface used by GPU accelerators - shown in Figure 5. The FPGA platform will be integrated with LuxRays - a subsection of the LuxRender suite dedicated to hardware acceleration. LuxRays is an open-source C++ library that implements ray tracing with both QBVH and *k*D-tree acceleration hierarchies available. The FPGAs onboard memory is used to store scene primitives and the acceleration hierarchy. An OpenCL implementation is available to provide a comparison to GPU performance.

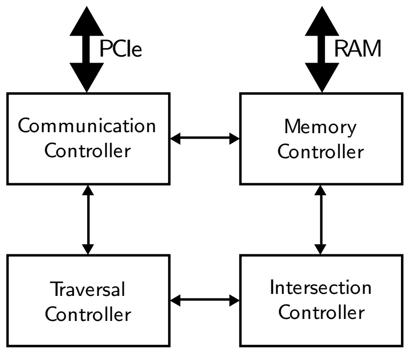

Fig. 6. Complete ray tracing acceleration platform.

V. FPGA Ray Tracing Platform

This section outlines the design goals and overview of the FPGA ray tracing platform. The communication controller, traversal controller, intersection controller and memory controller that make up the platform are detailed.

A. Design Goals and Overview

The ray-tracing platform was carefully designed to be agnostic to different acceleration hierarchies *i.e.* to traverse both *k*D-tree and QBVHs without the need for reimplementation - the only difference between the two is their layout in memory. The platform can also traverse programmable widths of acceleration hierarchies - *i.e.* varying n for nBVH - to evaluate their memory requirements and performance. Storing the acceleration structure and primitives on the memory local to the FPGA makes it available at lower latency and allows the data to stream directly into the intersection pipelines. The complete design, shown in Figure 6, consists of the four components described in the following sections.

1) Communication Controller: The communication controller (CC) is responsible for maintaining memory mapped configuration registers, completing the transfer of nodes, primitives and rays to the FPGA and the transfer of hits back to the host computer. To initiate traversal, the host computer application must write the memory location of the nodes and primitives to the configuration registers and signal the CC to start. The CC then transmits the appropriate read request packets to the host computer MMU which replies with the requested data. The CC forwards the node and primitive data to the memory controller to be written to memory. Once complete, the host computer writes the address of the rays to be intersected and an address to write the results to the configuration registers and signals the CC to start. The CC then transmits the appropriate read request packets to the host computer MMU which replies with the requested data. The CC forwards the ray data to the traversal controller and waits for it to return results. When results arrive the CC writes them to the host computers memory. The step of reading rays and writing results can be repeated several times for different sets of rays in the same scene. Detail of the CC connections are shown in Figure 7.

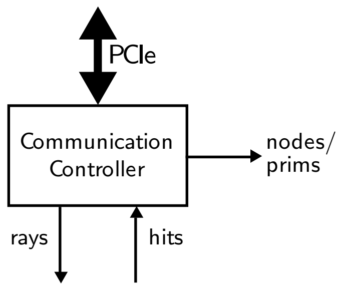

Fig. 7. The communication controller and connections to PCI*express*, the traversal controller and the memory controller.

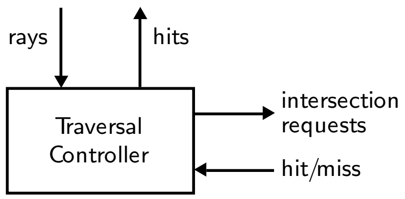

Fig. 8. The traversal controller and connections to the communication controller and the intersection controller.

2) *Traversal Controller:* The traversal controller (TC) buffers rays received from the communication controller in a FIFO and transfers them to block RAM when a traversal unit is available. A traversal unit (TU) is a progammably replicatable module that implements acceleration hierarchy traversal as shown in Figure 3. When the 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 controller to determine if the ray intersects with any of the nodes' children. The intersection controller 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. 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 controller which responds with a hit or miss flag and the distance to intersection 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. If no intersection is found, a node is popped from the queue and traversal continues. If the stack is empty then traversal terminates without intersection. Detail of the TC connections are shown in Figure 8.

Efficient implementation of the priority queue in the TU is of great importance to its replicatability and is discussed in more detail in the following section.

3) *Intersection Controller:* The intersection controller (IC) buffers intersection requests from the traversal units and passes them onto the memory controller when requested. As multiple intersection requests can be received in a single cycle, the IC uses fixed-priority scheduling to determine which to receive and acknowledge. The traversal unit must wait and hold its' request signal high until it is acknowledged by the IC.

In response to an intersection request, the memory controller streams the objects for the request to the IC. The IC implements deep ray-box and ray-triangle intersection pipelines into which the responses from the memory controller are fed. Detail of the IC connections is shown in Figure 10.

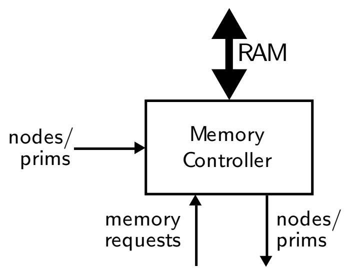

Fig. 9. The memory controller and connections to RAM, the communication controller and the traversal controller.

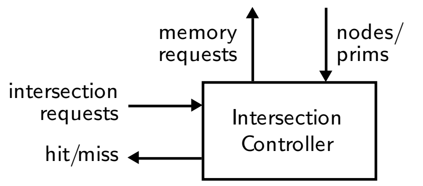

Fig. 10. The intersection controller and connections to the traversal controller and the memory controller.

4) *Memory Controller:* The memory controller (MC) takes nodes and primitives from the CC and writes them to consecutive addresses in local memory. The use of SSE in LuxRays means the acceleration hierarchy and primitives are stored in memory as a structure-of-arrays. The MC swizzles the incoming data into an array-of-structures before it is written to memory.

Intersection requests are taken from the IC and translated into the appropriate local memory read requests. When the requested data is returned from local memory it is streamed to the appropriate IC intersection pipeline. Detail of the MC connections is shown in Figure 9.

VI. FPGA Ray Tracing Implementation

All object references are encoded using 32 bit unsigned integers. The most significant bit is used to differentiate between nodes and primitives, with 1 representing a primitive. For a node, the remaining 31 bits represent the node id. The address and length of the node memory requests are shown in Equations (5a) - (5b). The root node has the id 0 and is the first node placed on the priority queue in a traversal unit when traversal starts for a ray.

$$ADDR = NODE[31:1] \times NODE\_WIDTH \tag{5a}$$
$$LEN = NODE\_WIDTH \tag{5b}$$

For primitives, bits 31 to 28 represent the number of primitives in the leaf and the remaining 27 bits represent the starting location of the first primitive. The address and length of the primitive memory requests are shown in Equations (6a) - (6b).

$$ADDR = PRIM[27:1] \times PRIM\_WIDTH \tag{6a}$$
$$LEN = (PRIM[31:28]+1) \times PRIM\_WIDTH \tag{6b}$$

TABLE I. PLATFORM PERFORMANCE FOR 16,384 RAYS IN 1000 RAYS/S

		Scene triangles			
		500	2000	10000	20000
Device	CPU	264.3	65.5	14.4	7.3
	GPU*	65.5	15.2	3.0	1.5
	FPGA	68.8	47.2	21.2	21.1

TABLE II. PLATFORM PERFORMANCE FOR 65,536 RAYS IN 1000 RAYS/S

		Scene triangles			
		500	2000	10000	20000
Device	CPU	233.3	64.5	14.5	7.4
	GPU*	62.6	15.2	–	–
	FPGA	67.3	49.3	21.3	21.1

Priority queues are efficiently implemented in hardware by a series of comparators and a shift register [15]. Each element is stored in a register and is compared in parallel to the input data. The data is stored in the first register of lower priority and all registers of lower priority are shifted across, effectively sorting the data in a single cycle. Data is extracted by outputting the highest priority register and shifting all other registers along. This creates a highly efficient priority queue capable of inserting and extracting data in a single cycle.

The platform used is a Xilinx Virtex-5 PCI*express* Development Board in an Intel Q6600 host running Windows. The board contains a Xilinx Virtex-5 LX330T FPGA with 192 DSP48E slices, 331776 logic cells and 11664 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 PCI*express* 1.0 bus capable of 16Gb/s end-to-end bandwidth.

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 PCI*express* core [16] and a modified Bus Master DMA reference design. The MC interfaces with the DDR2 using a Xilinx MIG controller [17]. FIFOs and block RAMs are generated using the Xilinx Core Generator where required.

Ray-triangle intersection (Equations (1) - (3c)) and ray-box intersection (Equations (4a) - (4b)) modules were implemented in VHDL using Xilinx floating-point cores. They have pipeline lengths of 84 and 40 respectively and both operate at 250MHz. This makes a full pipeline of RTI calculations on the FPGA 30 times faster than in software [18].

VII. Platform Performance

In this section we evaluate the potential viability of our platform. The three implementations being tested are a CPU, GPU and our FPGA platform. The CPU is an Intel Q6600 at 2.4 GHz. The GPU is a Nvidia Quadro FX 570. The FPGA platform was implemented with 4 traversal units - each containing a priority queue capable of storing 32 nodes. The target frequency was 250MHz and 26% of the registers and 25% of the LUTs available on the device were used.

Four test scenes were randomly generated containing differing numbers of primitives. Increasing the number of scene primitives increases the height of the acceleration hierarchy and increases the minimum size of the priority queue required to traverse it. Each implementation traced 16,384 and 65,536 randomly placed and orientated rays and the time taken was measured. The rates obtained by each implementation are shown in Table I and Table II.

*The GPU is included in these results for completeness. However it is not indicative of typical GPU performance as the GPU used is low end. Typical GPU performance exceeds 10^6 rays/s [3].

VIII. Discussion

The results in Table I and Table II show that all implementations decrease in performance as scene density increases. This is due to larger scenes having taller acceleration hierarchies, requiring more steps to traverse to leaves, more leaves to traverse and larger priority queues to store intermediary nodes.

The CPU shows the best performance for scenes with lower density, however it has the sharpest performance drop off as density increases. This is likely due to the cache of the acceleration hierarchy being less effective as it increases in size. While the low end GPU is not indicative of typical GPU performance, it showed the same tendencies as the CPU.

The FPGA shows more stable performance with performance decreasing least as density increases and better performance than the CPU for the most dense scene. The stability is likely due to the performance of the FPGA single-cycle priority queue compared to the naive queue used in both the CPU and GPU implementations. With promising results from the FPGA platform, it can now be used to investigate the best design.

Memory performance can be optimised by swapping the current dual-rank module for a single-rank module. When the module is dual-rank the MIG controller operates at a maximum frequency of 150MHz, meaning a maximum bandwidth of 19.2Gb/s. A single-rank module can operate at a maximum of 250MHz for our target device, giving 40% extra bandwidth but at the cost of smaller module capacity. The increase in bandwidth would decrease the time taken to perform each intersection request and increase the number of rays per second processed.

Further optimisation may be achieved by increasing the number of traversal units. With only four traversal units, the intersection requests generated use 20% of the available memory bandwidth. When scaling the number of traversal units, either this memory bottleneck is going to be reached or the device logic will be exhausted. The memory bottleneck can be alleviated by implementing a direct mapped cache of the acceleration hierarchy, while a more logic efficient priority queue implementation will allow more traversal units on the device. The configuration tested uses 15% of the device block RAM, leaving enough room for a reasonably large cache.

The current priority queue implementation requires each storage element to have a separate comparator unit, resulting in heavy usage of the available logic elements on the device. If the renderer deals with scene data of unknown sizes or sizes 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.

The concept of a short-stack with spilling described in [11] demonstrates the benefits of a short, fast stack spilling over into a longer, slower stack. This concept is applied to priority queues, with a faster small priority queue performing the majority of the executions whilst a larger priority queue is used to catch the spillover data. This implementation allows the speed of the short queue to be utilized whilst a secondary large queue would allow larger data sets to be processed with lower resource consumption. We are currently investigating efficient priority queue optimisations to improve their scalability while maintaining their size.

The bandwidth available to the board from the PCI*express* bus is capable of delivering 5.5×10^7 rays/s to the FPGA platform. If the FPGA were able to meet the processing demand of the incoming rays it would represent performance comparable to current GPU implementations. With FPGAs consuming considerably less power than GPUs and power consumption becoming a considerable problem for datacenters, the FPGA platform could offer a potential cost saving by consuming less power per ray processed.

IX. Conclusion

We have presented a flexible platform for ray-tracing on FPGAs. The platform can trace varying widths and heights of acceleration hierarchy. The platform takes advantages of an efficient FPGA priority queue implementation, allowing single-cycle operations. We have integrated the platform with LuxRays and compared performance with a CPU and GPU. While the CPU was faster for smaller test scenes, the FPGA performed more consistently over all scenes and was faster for larger scenes. Promising optimizations are yet to be investigated on the FPGA platform which should lead to increased performance.

References

[1] A. Apodaca, L. Gritz, and R. Barzel, *Advanced RenderMan: creating CGI for motion pictures*. Morgan Kaufmann, 2000.

[2] D. Cortes and S. Raghavachary, *The RenderMan Shading Language Guide*. Thomson Course Technology, 2007.

[3] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," *Proc 2007 Symp on Interactive 3D graphics and games*, p. 167, 2007.

[4] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," in *ACM SIGGRAPH 2010 papers*. New York, USA: ACM, 2010, pp. 66:1–66:13.

[5] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Trans Graphics*, vol. 24, no. 3, pp. 434–444, 2005.

[6] C. Cameron, "Using FPGAs to supplement ray-tracing computations on the Cray XD-1," in *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, june 2007, pp. 359 –363.

[7] A. Nery, N. Nedjah, F. Franca, and L. Jozwiak, "A parallel ray tracing architecture suitable for application-specific hardware and GPGPU implementations," in *14th Euromicro Conf Digital System Design*, 31 2011-sept. 2 2011, pp. 511 –518.

[8] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *Journal of Graphics, GPU, and Game Tools*, vol. 2, no. 1, pp. 21–28, 1997.

[9] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980.

[10] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '05. New York, NY, USA: ACM, 2005, pp. 15–22.

[11] J. Novák, "Global illumination methods on gpu with cuda," Master's thesis, Czech Technical University, June 2009.

[12] V. Havran, "Heuristic ray shooting algorithms," Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[13] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 7–13.

[14] H. Dammertz, J. Hanika, and A. Keller, "Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays," *Computer Graphics Forum*, vol. 27, no. 4, pp. 1225–1233, 2008.

[15]

[16] Xilinx, "LogiCORE IP Endpoint Block Plus v1.15 for PCI Express."

[17] ——, "Memory Interface Generator."

[18]