Caching Architecture for Flexible FPGA Ray Tracing Platform

S. Collinson[a], O. Sinnen[a]

[a] *Department of Electrical and Computer Engineering, University of Auckland, New Zealand*

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, FPGAs with their flexibility have the potential to be more power efficient accelerators.

This paper investigates and proposes an FPGA acceleration platform to easily and efficiently explore parallelism in ray tracing on FPGAs. It is integrated with LuxRender, a modern and open rendering engine. A major focus is on the proposal of a cache architecture, given that memory bandwidth is identified as a bottleneck. For the design of the domain-specific cache, we study the typical memory access patterns to ray tracing data structures, i.e. acceleration hierarchy and scene primitives. The results lead to the proposal of a data structure specific cache (separating nodes from primitives) and a novel node cache replacement policy.

We demonstrate that the proposed cache architecture successfully alleviates the memory bandwidth bottleneck and that the novel 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. The evaluation further shows that scaling the number of traversal units with a cache provides

Email addresses: scol084@aucklanduni.ac.nz (S. Collinson), o.sinnen@auckland.ac.nz (O. Sinnen)

an increase in performance over all scenes. We also demonstrate the energy and bandwidth efficiency of the proposed platform in comparison to a CPU and a GPU platform.

---

1. Introduction

Digital rendering of special effects is now a routine requirement for moviemakers, 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 [1]. Ray tracing methods can create the desired photo-realistic digitally rendered images, but requires massive computational power, as the scene has to be rendered frame-by-frame at very high resolutions. It can keep a several thousand server cluster occupied for months rendering a full-length animated movie [2]. GPUs are typically used to accelerate ray tracing, providing performance of up to tens of 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 [3].

FPGAs are massively parallel and show greater power efficiency than GPUs [4] so may be an alternative to GPUs when power is a concern to render farm operators or for embedded environments. The goal of our research work is to investigate architectural research avenues for FPGA implementations of ray tracing. To do so, we have created a flexible tool to explore different ray-tracing acceleration techniques on FPGAs, allowing quick discovery and elimination of performance bottlenecks. To test this platform in a useful and practical environment, we have integrated it in LuxRender. LuxRender is an up-to-date open-source rendering engine that provides the image quality required by movie makers [5]. Our proposed FPGA acceleration platform gives fine-grained control over the area of investigation, only replicating what is necessary to improve performance and leaving more resources for other potential acceleration areas. As with MIMD implementations on GPUs for incoherent rays, our platform

performance is limited by memory bandwidth. In our early evaluations we explored scaling the number of traversal units, which increases the number of rays processed in parallel. As we scaled the number of traversal units, they eventually generated more intersection requests than the memory controller could service, which limited performance. FPGAs offer a more flexible memory architecture than GPUs which may enable a ray tracing specific cache to more efficiently side-step memory bandwidth limitations. While this problem may be solved with a generic direct-mapped or set-associative cache, with knowledge of the data structure we are caching, there may be information available to us at run-time that can be used to implement a more efficient cache through with a domain-specific cache line and a smarter replacement policy. We expect that with an effective cache in place the performance for the system to scale proportional to the number of traversal units while memory is not the bottleneck.

To create an domain-specific cache we first analyse memory access types and patterns, taken from renders of four test scenes created from large models commonly used in previous ray-tracing research, to observe what types of data are the most efficient for caching. This analysis shows that caching the acceleration hierarchy of the scene is more important than the scene primitives. Further, elements higher in the acceleration hierarchy are accessed, on average, more frequently than lower elements. The structure of the acceleration hierarchy during construction can also affect memory bandwidth requirements. Hence, we explore the affect on memory bandwidth of changing the maximum number of primitives that are allowed to be stored per leaf node and how, when a cache is introduced, this affects cache efficiency. We then present the design and evaluation of a domain-specific cache with a level replacement policy to alleviate the memory bottleneck. Our novel level replacement policy is evaluated using the workload obtained from our test scene renders. Finally we implement a direct-mapped cache and evaluate the change in performance and memory utilization on our platform. Lastly, to put the performance of our proposed FPGA approach into relation with non-FPGA platforms, we perform an experimental evaluation against an optimized CPU and GPU implementation, all under

LuxRender. While our platform cannot compete with with them (or other modern GPUs) on raw performance, we will demonstrate that it clearly achieves the higher performance in relation to energy consumption and available bandwidth.

This paper is laid out as follows: we first give a brief explanation of the ray tracing process in Section 2 and previously investigated acceleration techniques in Section 3. We then describe our FPGA ray tracing platform in Section 4. Using the platform, we analyse the memory access patterns during ray tracing of four common scenes in Section 5. We use this analysis to design a cache for scene nodes in Section 6 and a novel level replacement policy in Section 7 and experimentally evaluate their performance. The impact of the cache on the performance of the ray tracing platform is evaluated in Section 8. Before conclude with Section 10, we put the FPGA platform performance into relation with a CPU and a GPU platform in Section 9.

2. Ray Tracing

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. 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. 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 [7] for RTI tests is commonly used in digital rendering programs, including LuxRender used in the presented work.

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 [8].

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

Bounding Volume Hierarchies (BVHs) recursively split the set of scene primitives until the leaf size (number of triangles) 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.

Inputs to the BVH traversal algorithm are the scene primitives, the tree nodes 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*). The order of nodes, *i.e. near* to *far*, is determined by ordering by ascending intersection distances along the given ray. 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.

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.

3. Related Work

The inherent parallelism in ray tracing calculations has lead to much research in the area and several hardware acceleration techniques. The following sections outline previous work into ray tracing hardware acceleration on parallel architectures and ray tracing caches.

3.1. Ray Tracing for SIMD Architectures

SIMD architectures are commonly used to accelerate ray tracing with wide SIMD units available on many modern hardware architectures such as 16-wide AVX on Intel CPUs [9], 16-wide SIMD on Intel Many Integrated Core (Intel MIC) [10], and 16-or-more-wide SIMD on GPUs [11]. 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.

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 [12, 13]. 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 (e.g. reflected rays) due to their incoherence.

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) [14, 15, 16]. 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. 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 [15].

Because of inefficiencies above a branching factor of four and as the typical SIMD width of modern processors is four, MBVH is typically implemented with a width of four and called QBVH (Quad-BVH), however the concept is not limited to this width.

3.2. Hybrid Ray Tracing for MIMD Architectures

As rays are independent of each other, the single ray multiple box or triangle method can be parallelized 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 papers [17, 18, 19, 20] and is the approach we have implemented in our platform.

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 [11]. Benthin *et al.* [21] 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.

3.3. Ray Tracing on FPGAs

FPGAs have also been considered for ray tracing. Schmittler and Woop presented two FPGA based ray tracing prototypes. The first prototype was a fixed function ray tracer called SaarCOR [22, 23]. The second was a programmable ray processing unit, which allowed for higher quality in the rendered image at the cost of area or performance [24]. It was implemented on a Xilinx Virtex-2 FPGA and performed similarly to a CPU. Cameron [25] 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 1.2×10^7 ray-object intersection tests per second. Fender and Rose [26] 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 6×10^8 ray-triangle intersection tests per second. Nery *et al.* [27] 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 1.6×10^5 ray-object intersection tests per second while the GPU system achieved 4.0×10^6 ray-object intersection tests per second. Lee [28] presented a mobile ray tracing system using the Samsung Reconfigurable Processor. The architecture was implemented on eight Xilinx Virtex-6 LX760 FPGAs each with dual-channel DDR2 RAM. The implementation traced 3.2×10^6 primary rays per second on a 1.7×10^5 triangle scene.

3.4. 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.* [29] 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.

4. Platform

The target acceleration platform of this work is a PCI*express* attached FPGA with onboard memory - the same interface used by GPU accelerators. The FPGA platform is integrated with LuxRays - a subsection of the LuxRender

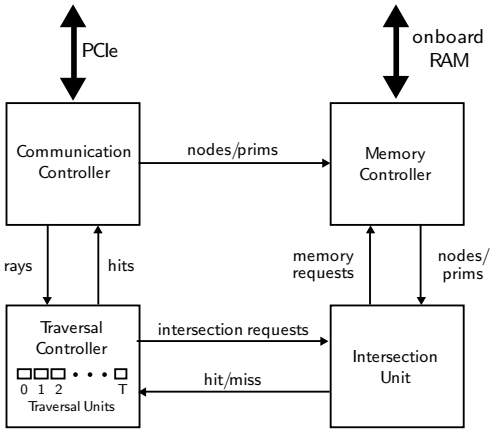

Figure 1: Complete ray tracing acceleration platform.

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 development board's onboard 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 intersection pipelines.

To trace a single ray through an acceleration hierarchy (Section 2), the design must contain three components: the control flow to determine when to continue or end traversal of the acceleration hierarchy, a priority-queue to store intermediary results when we need to traverse more than one child node, and a method to determine ray-box and ray-triangle intersection. On our platform, shown in Figure 1, we split these functions into separate modular units to allow easy replicability. 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) 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), 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 PCI*express* to co-ordinate the flow of nodes, primitives, rays and hits to and from the host computer, in the communication controller (CC), and to interface with the onboard memory and store nodes and primitives to be accessed for intersection tests, in the memory controller (MC).

This ray-tracing platform design has the following features that allow to smartly exploit parallelism.

- **Agnostic to different acceleration hierarchies:** Both kD-tree and QBVHs can be traversed without the need for reimplementation - the only difference between the two is their layout in memory.
- **Programmable widths of acceleration hierarchies:** It supports changing M for MBVH or the maximum number of primitives per leaf to evaluate memory requirements and performance.
- **Multiple rays and intersecting with multiple nodes:** The platform has an MIMD architecture with no assumptions about ray coherency to be efficient for both coherent and incoherent rays.
- **Modular and replicable:** The TU can be easily interchanged to explore the performance of different techniques (*i.e. kd-restart et al.*) or replicated to explore the efficiency of parallelisation. The IU and priority-queues within the traversal unit may also be interchanged to explore their utilisation and replicability versus their performance.

4.1. Implementation

The hardware system used in our work is a Xilinx Virtex-5 PCI*express* Development Board in an Intel i5 host running Ubuntu Linux. The board contains a Xilinx Virtex-5 LX330T FPGA with 192 DSP48E slices, 331,776 logic cells and

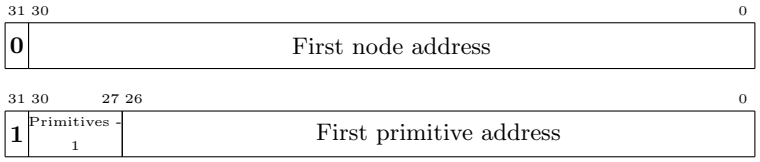

Figure 2: Node (top) and Primitive (bottom) data representation.

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 PCI*express* 1.1 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 and a modified Bus Master DMA reference design [30]. The MC interfaces with the DDR2 using a Xilinx MIG controller. The MC, IU and TC run with a 150MHz clock while the CC runs with a 250MHz clock. FIFOs and block RAMs are generated using the Xilinx Core Generator where required to buffer data or cross clock domains.

All node and primitive references are encoded using 32 bit unsigned integers (Figure 2). 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 id. 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. 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 location of the first primitive. 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 and a reference. This makes a single node in a MBVH require $M \times (7 \times 32)$ bits of memory, which means 896 bits for a QBVH node. Each

(a) Kitchen (b) Classroom

(c) Hairball (d) Sponza

Figure 3: High quality renders of the test scenes used.

primitive in a leaf is represented by three single-precision floating-point points in three-dimensions, meaning 288 bits of memory per primitive and 1,152 bits for four primitives.

5. Memory Access Analysis

An early evaluation of our platform showed that memory is the bottleneck to performance increases. While this problem may be solved with a generic direct-mapped or set-associative cache, with knowledge of the data structure we are caching and the flexibility of FPGAs, there may be information available to us at run-time that can be used to implement a more efficient application specific cache. Therefore, this section analyses memory access patterns and bandwidth requirements 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

Scene	Primitives	Nodes	Max. Tree Depth	Resolution
Kitchen	86,032	26,381	15	640×480
Classroom	$2.47 \cdot 10^5$	80,356	15	640×480
Hairball	$2.88 \cdot 10^6$	$9.96 \cdot 10^5$	17	960×540
Sponza	66,450	22,585	12	960×540

Table 1: Scene statistics using QBVH and maximum four prims per leaf.

caching. Three memory bandwidth areas are explored: node bandwidth relative to primitive bandwidth per ray, the affect of leaf size on bandwidth and node bandwidth per level of the acceleration hierarchy.

5.1. Scene Setup for Analysis

Four test scenes with different sizes and characteristics were selected from freely available models: Kitchen, Classroom, Hairball and Sponza [31]. 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 [32] and 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 3 and scene statistics using QBVH and maximum four primitives per leaf are given in Table 1. Node and leaf memory accesses are recorded for each test parameter (see next 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 surface area heuristic (SAH) to determine the split position during tree construction [33].

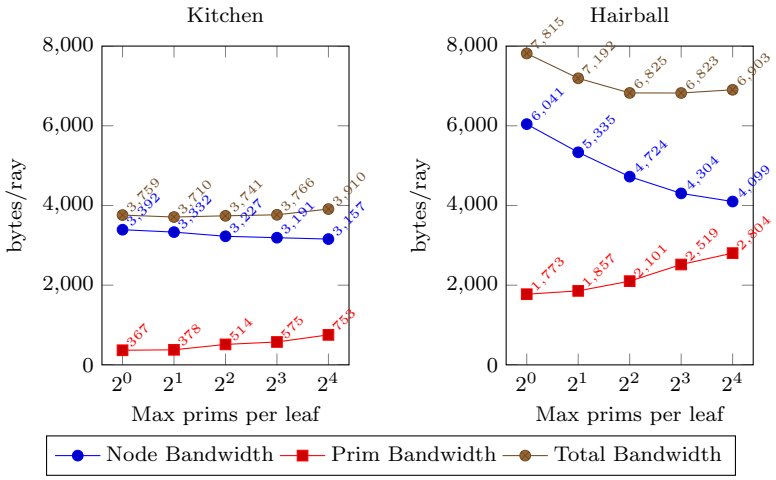

Figure 4: Bandwidth per ray vs Max prims per leaf

5.2. Node vs Leaf Bandwidth

The memory 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 for Kitchen and Hairball are shown in Figure 4 (results for Sponza and Classroom are very similar to those of Kitchen).

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.

5.3. Level affect on Node Bandwidth

With significantly more bandwidth required per ray for nodes, we now analyse memory accesses by their level in the acceleration hierarchy. With an MPPL of four being a good choice over all scenes, we use a dump of the acceleration hierarchy and the same set of node memory accesses for MPPL = 4 to present exemplary data in Figures 5 - 6.

All scenes show the number of nodes increases initially when moving down the tree, peaks just after half-way down the tree then decreases (top-left chart). All scenes show that their hierarchy loses density as we move 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 inserting all primitives.

Node accesses by level show a quick increase to a peak and then decrease moving down the hierarchy (top-right chart). For all scenes the peak in node 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 (bottom-left chart). 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

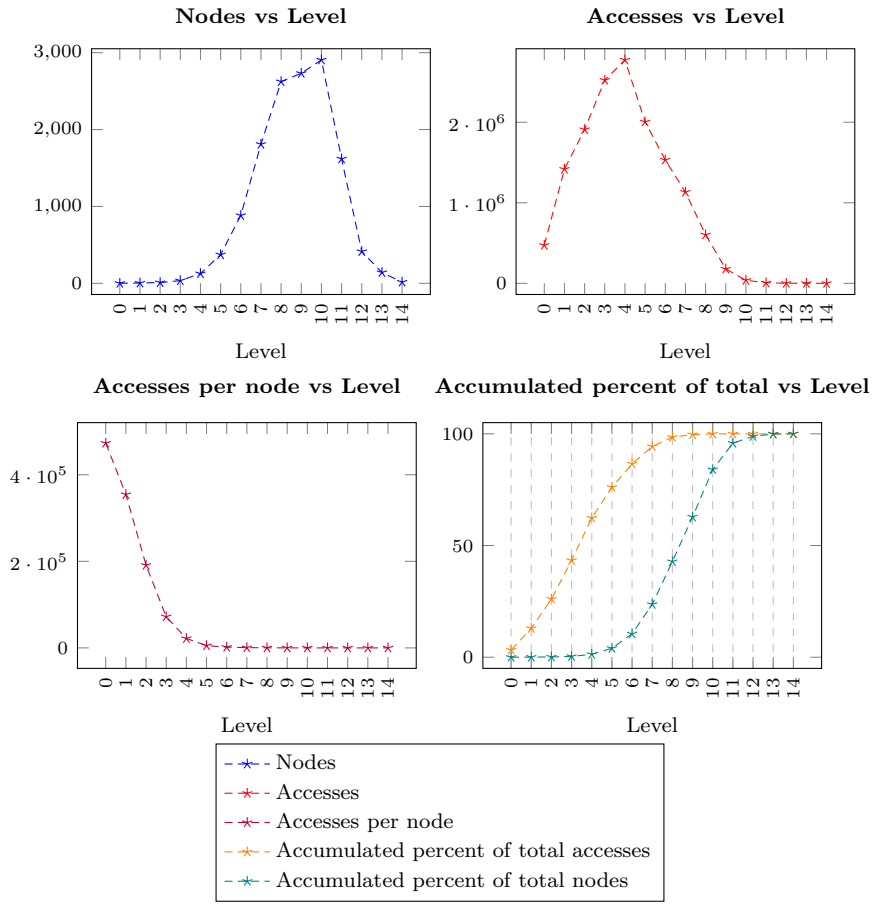

Figure 5: Kitchen scene analysis.

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 we explore a cache replacement policy that exploits this temporal locality by only replacing nodes in the cache when they have the same or higher level.

The accumulated total of nodes and accesses by level (bottom-right chart) shows the affect 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, Classroom and

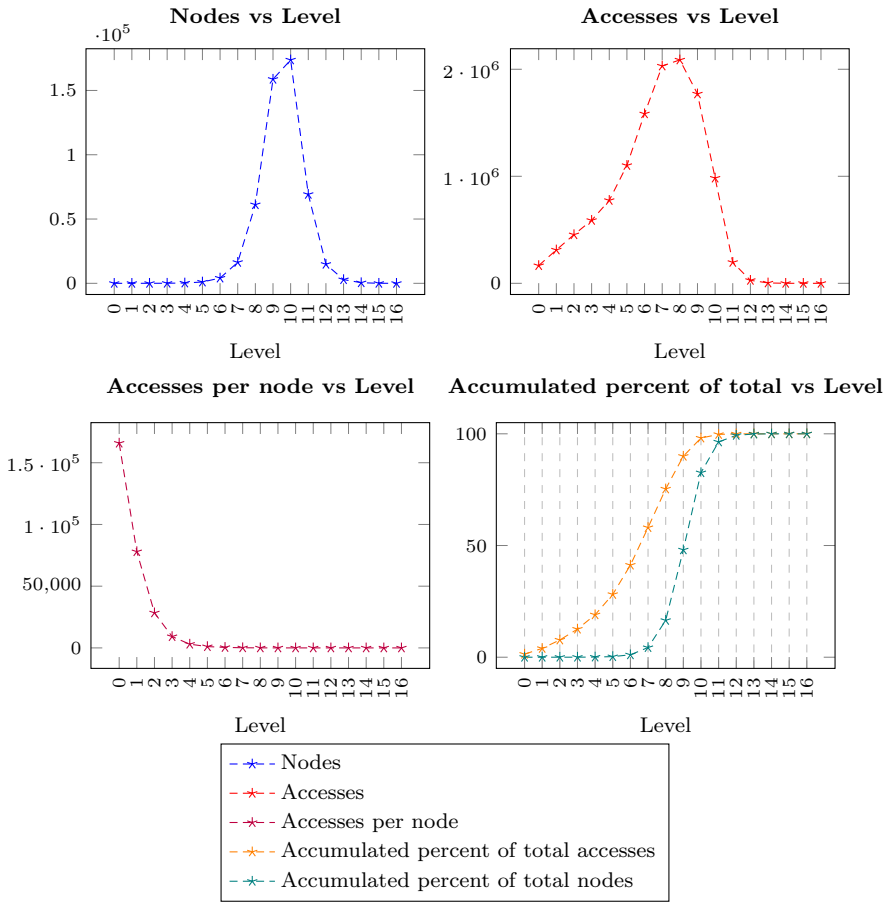

Figure 6: Hairball scene analysis.

Sponza this represents 1.3%, 0.8%, 3.0%, respectively, of the total nodes but 62%,73%,68%, respectively, 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.

6. Caches for Ray Tracing

Based on the previous memory access study, we now design, implement and evaluate of our proposed node and primitive caches.

6.1. Design

The node and primitive caches will naturally sit in the Memory Controller in Figure 1, 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 2, 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 analyze differences in performance. The cache will be direct mapped, simplifying implementation and logic requirements by requiring only one row be compared with incoming requests [34].

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 we use multiple threads of computation 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.

6.2. Implementation

Index	**Tag**	**Node Data**
0	Tag 0	Node Data 0
1	Tag 1	Node Data 1
	⋮	⋮
$N-1$	Tag $N-1$	Node Data $N-1$

Figure 7: Node cache implementation.

The node cache is shown in Figure 7 and the primitive cache is the same but with a different data width. They are both implemented in VHDL, instantiating block RAM as storage for both tags and data and using `generate`

and `parameter` statements to enable the cache to be of configurable size during synthesis. Inputs to the cache are requests from the intersection controller and data returned from the onboard memory controller. 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 onboard memory.

The target settings for the node cache are QBVH, meaning a node row data width of 896 bits. 288 bits are required per primitive and the total row data width for the primitive cache is determined by this value and the MPPL setting. The flexibility of FPGA block RAM primitives means that if the total space provisioned for the primitive cache can accommodate 4,096 primitives, then this space can be configured - using $W \times D$ format, where W is width (or MPPL) and D is depth - as a 1×4096 cache, a $2 \times 2,048$ cache or a 4×1024 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.

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 block RAM, 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.

6.3. Evaluation

The memory access analysis showed that, per ray, significantly more bandwidth is required for nodes than for primitives. For this reason we exclusively implement the largest node cache that can fit on our platform initially and will only look to include the primitive cache 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 5.1 over different MPPL settings. A cache with the maximum size that

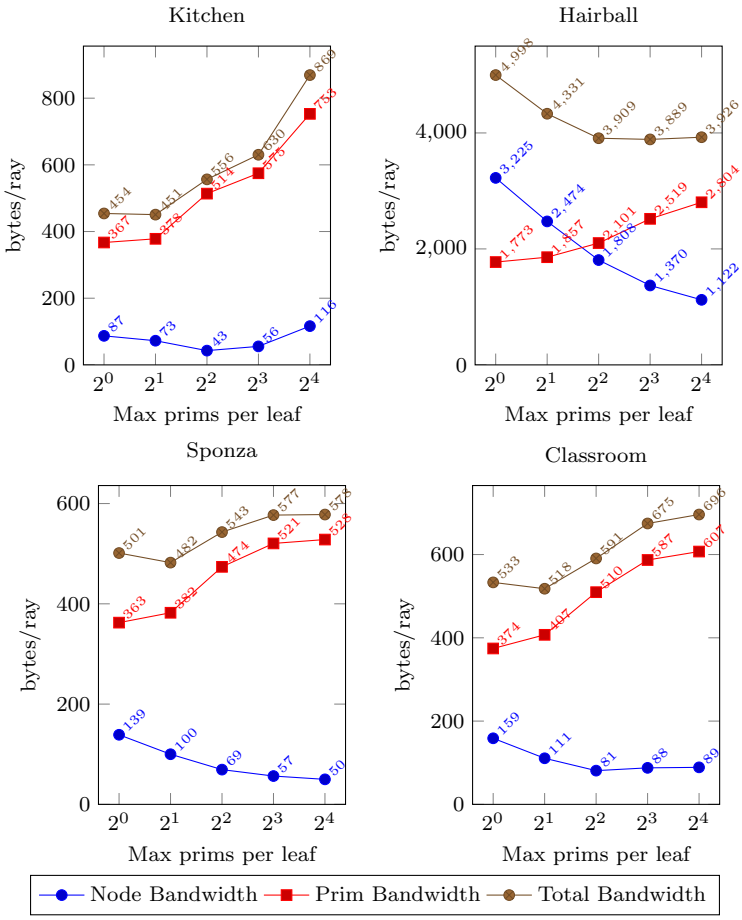

Figure 8: Bandwidth per ray vs Max prims per leaf with Node Cache

can fit on the target platform is simulated 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 we are not yet simulating the primitive cache, the primitive bandwidth requirement is the same as the results from Figure 4.

The results showing the node, primitive and total bandwidth requirements when only the node cache is implemented are shown in Figure 8. The results show, when compared to Figure 4, that the node cache is extremely effective for Kitchen, Sponza and Classroom for all MPPL settings, decreasing the required

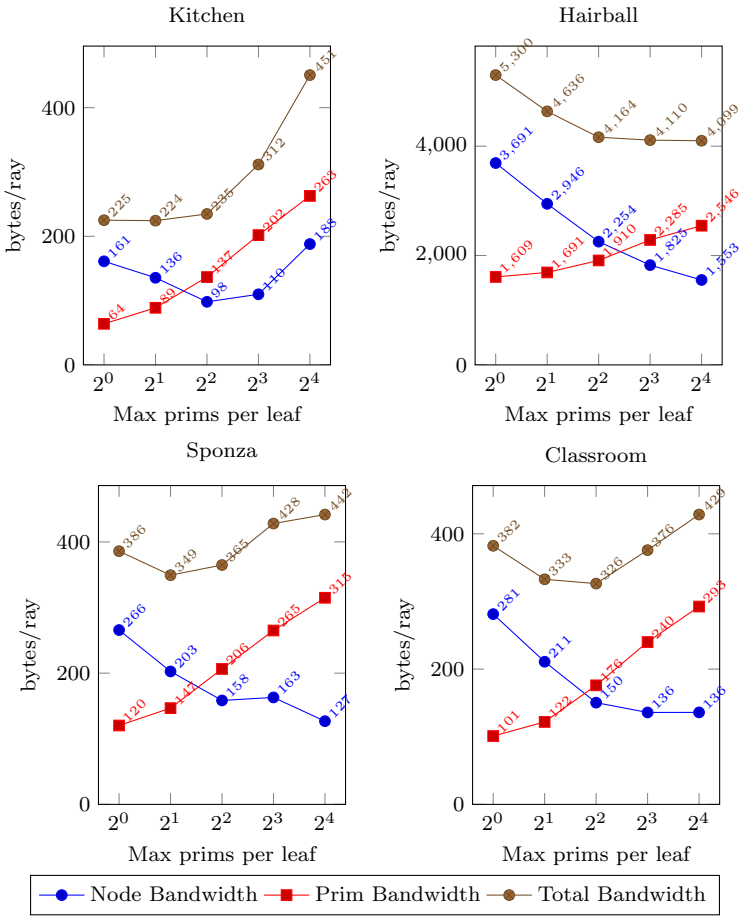

Figure 9: Bandwidth per ray vs Max prims per leaf with Node and Prim Cache

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 8 shows the same tendencies as Figure 4 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. We now look at sacrificing some of performance the node cache by making it smaller and using the freed block RAM to implement a primitive cache. We do this by halving the size of the node cache to 4,096 and then implementing the largest primitive cache possible in the remaining block RAM, 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 affect 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 affect the miss rate.

The results depicting node, primitive and total bandwidth requirements with the node and primitive cache implemented are in Figure 9. The results show, when compared to Figure 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.

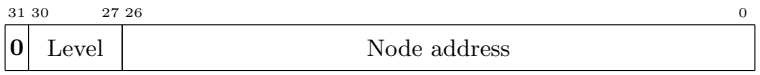

Figure 10: New node data representation.

7. Level Replacement Policy

Looking at the accesses per node per level from Section 5, 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 we will investigate a cache replacement policy that makes use of this observation, not replacing based on recent use but instead based on the node's level in the acceleration hierarchy, called the Level Replacement Policy (LRP).

7.1. 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. This implicitly corresponds to the least recently used (LRU) "replacement policy", where the data replaced is the one that has been unused for the longest time relative to the other data items. In a direct-mapped cache, in contrast to associative cache, there is no need to keep track of usage history, given that there is no choice. We keep this lean and simple design of a direct-mapped cache, but change the replacement policy. Nodes on higher levels of the acceleration hierarchy are given 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 direct-mapped cache line if it is on the same or higher level. This allows the simplicity of a direct-mapped cache to be maintained, while achieving a Level Replacement Policy (LRP).

7.2. Implementation

In order for us to compare the level of each node we must make it available during each request. We do this by encoding the level of the node in the same way we encode the number of primitives in each leaf, using four upper bits of the address, shown in Figure 2. The new node encoding, shown in Figure 10, allows us to encode up to 16 levels of each scene. This decreases the number of nodes we can represent 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. During tree construction, we encode the levels 0 to 14 as is from `0x0` to `0xE` respectively, and nodes on level 15 or below have an encoding of `0xF`.

We also include in our design a 4-bit configuration register, `MAX_LEVEL`, to limit the depth to which our 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 (level) priority and are replaced using LRU.

7.3. Evaluation

To evaluate the LRP we use Modelsim 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. We only include `MAX_LEVEL` values up to six, as after this value performance degrades and no improvement was shown with any greater value over all scenes. For reference, we also include the miss rate of a fully-associative-by-level static buffer, where the buffer 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.* [29], where tree construction is changed from depth first to breadth first and the buffer stores the first N nodes based on their index. The results for Classroom and Hairball are shown in Figure 11; results for Kitchen and Sponza are qualitatively the same.

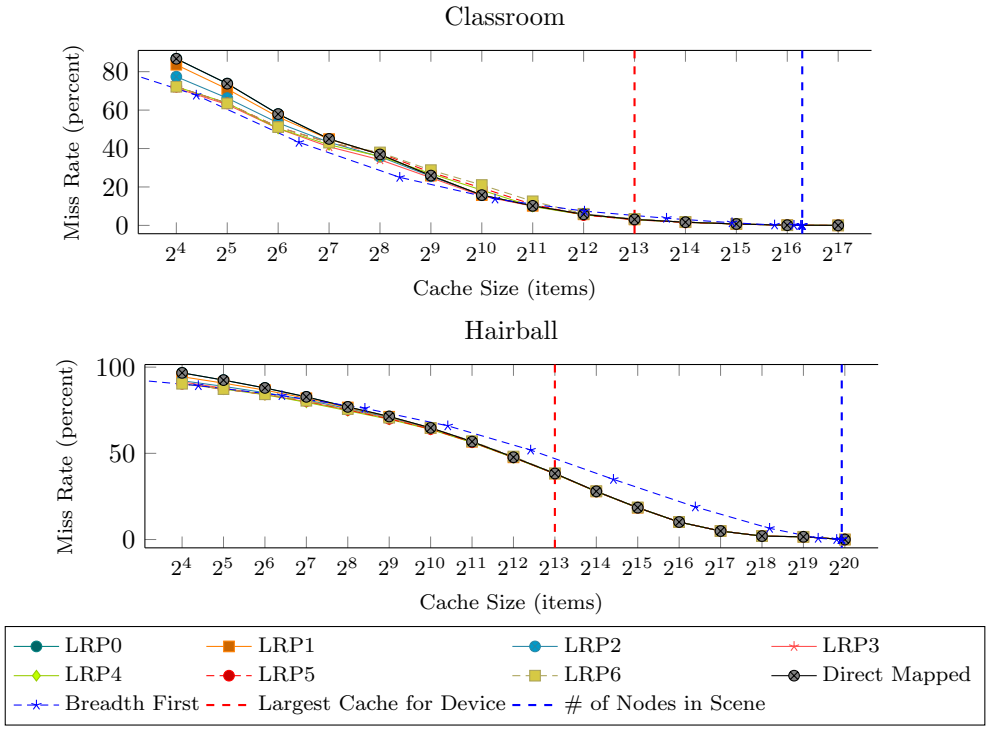

Figure 11: Cache Size versus Miss Rate

The curve for Directed Mapped corresponds to the original direct mapped cache with implicit LRU policy.

The first observation that can be made is that the LRP has an improved miss rate compared to Direct Mapped for small cache sizes. This is more evident for Classroom (and also for Kitchen and Sponza), but still noticeable for Hairball for very small caches. It seems that the LRP is able to somewhat compensate for the lack of storage resources. The maximum depth of 6 (LRP6) performs best among the tested. The advantage of LRP disappears with larger caches as can be expected. The results also show that just buffering the highest nodes is not a general good solution as can be seen by the worse miss rate of the Breadth First approach for Hairball.

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.

8. Cache Impact on Platform Performance

In Section Section 6 we demonstrated the significantly improved bandwidth when using the proposed cache. This section now studies the impact of the proposed cache on the performance of the entire ray tracing platform. The ease of scalability of our system allows us to scale different aspects of the design and quickly determine areas for potential performance improvement. In this evaluation we explore scaling the number of traversal units, which will increase the number of rays processed in parallel. As we scale the number of traversal units, they eventually generate more intersection requests than the memory controller can service, which limits performance. We expect that with an effective cache in place the performance for the system to scale much higher before the memory becomes the bottleneck.

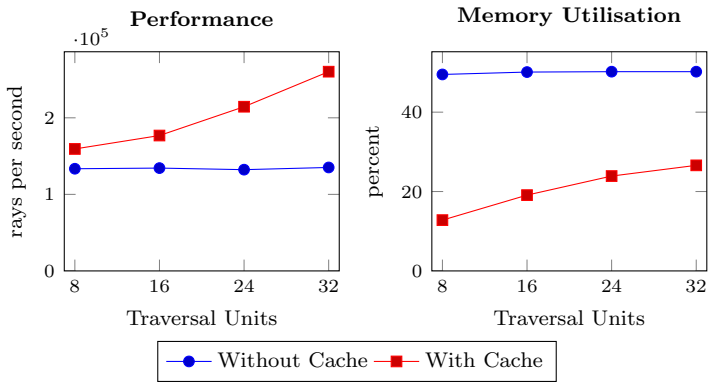

Figure 12: Kitchen Performance and Utilisation vs Traversal Units.

We created several versions of the platform, scaling the number of traversal units until the timing constraints could not easily be met on the FPGA, which resulted in a maximum of 32 traversal units for this device. The implemented cache was of the largest size supported by our device (2^13) with a direct mapped

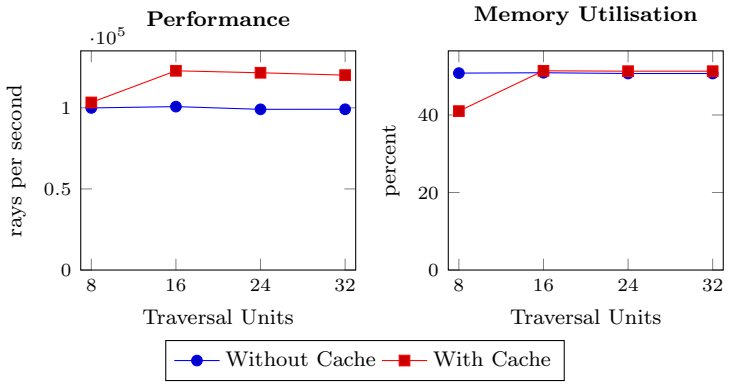

Figure 13: Hairball Performance and Utilisation vs Traversal Units.

cache policy as this was the effective policy for this size (see red line marker in Figure 11). The length of the priority queues in each implementation is 64, meaning 64 intermediary nodes can be stored for further intersection tests. While this queue length may limit the size of the scene able to be processed, it is the same statically defined limit used in LuxRays and therefore a good number in practice. For our comparison we rendered each scene for at least ten seconds and counted the number of rays processed in that time. Graphs of the resulting performance and memory utilisation for each scene are shown in Figures 12 - 13, where Kitchen is exemplary also for Classroom and Sponza.

The results show that on all scenes scaling the number of traversal units without a cache provides no improvement on performance, as memory utilisation is limited to not far above 50%. While the maximum transfer rate for the FPGA development board onboard 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 [35].

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

	Kitchen	Classroom	Hairball	Sponza
CPU	$8.7 \cdot 10^5$	$1.1 \cdot 10^6$	$3.31 \cdot 10^5$	$7.64 \cdot 10^5$
GPU	$2.54 \cdot 10^5$	$2.86 \cdot 10^5$	$65,400$	$2.65 \cdot 10^5$
FPGA	$2.6 \cdot 10^5$	$2.52 \cdot 10^5$	$1.23 \cdot 10^5$	$3.04 \cdot 10^5$

Table 2: Raw ray-tracing performance (rays/second) for different scenes

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.

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 8, and the poor miss rate for the maximum size cache on this scene, shown in Figure 11.

9. Performance compared to a CPU and GPU

Previous sections have shown the scalability of the platform and the effect of a cache on memory utilisation. Before we conclude, we want to put the overall platform performance into relation to CPU and GPU platforms. The CPU used is a quad-core Intel Core i5-3470 @ 3.2GHz with 16GB of dual-channel DDR3-1600 memory [36]. The GPU used is an AMD Radeon HD 5450 with 1GB of 800 MHz DDR3 memory [37]. While this is a rather low-end GPU, it was chosen on purpose to have a GPU supporting OpenCL 1.2 that is comparable in process technology and power consumption to the employed FPGA board (specification in Section 4.1 and also having large amounts of memory bandwidth available. While we expect higher raw performance from the CPU and the GPU in comparison to our FPGA platform, we will see that the FPGA platform is superior in terms of performance per consumed energy and available bandwidth.

For the CPU, we selected the best configuration in LuxRender for the given CPU, namely a fully optimised QBVH implementation with SSE and four threads. For the GPU we used a fully optimised OpenCL QBVH implementation from LuxRender. Our FPGA platform uses 32 traversal units.

Table 2 provides the best achieved performance of the three platforms across the four scenes. It is clear that the CPU performance is significantly higher than that of the FPGA. Surprisingly, the FPGA is quite on par with the GPU performance.

FPGA shows performance an order of magnitude slower than that presented by Lee [28] for scenes of similar sizes, however we used an older generation FPGA, which has 18 times less logic and 16 times less memory bandwidth available, making the performance comparable if these factors were scaled.

9.1. Energy Efficiency

To study the energy efficiency, we analyse the power consumption over the rendering period. 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 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 [38]. The FPGA power usage is estimated at 12W using the Xilinx Power Analyzer (XPA) [39].

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 estimates 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

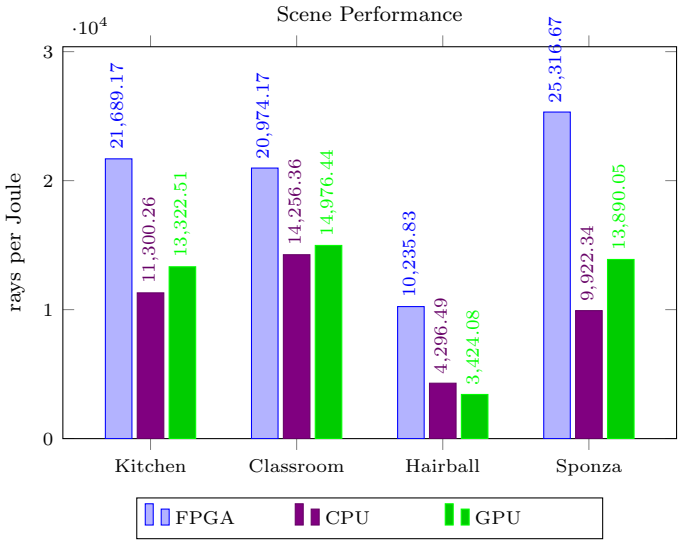

Figure 14: The number of rays traced per unit of energy on the platform and the most efficient CPU implementation over all scenes.

CPU approach does not include the power consumption of the memory subsystem, which is assumed to balance the comparison. 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 results are presented in Figure 14. The results clearly show that FPGA has better energy efficiency than both CPU and GPU over all scenes, with over twice the energy efficiency than CPU on Hairball and Sponza, just under twice the efficiency on Kitchen and just under 1.5 times more efficiency on Classroom. Energy efficiency varies across the scenes as the raw performance did, but the relative behaviour is very similar comparing the three platforms.

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

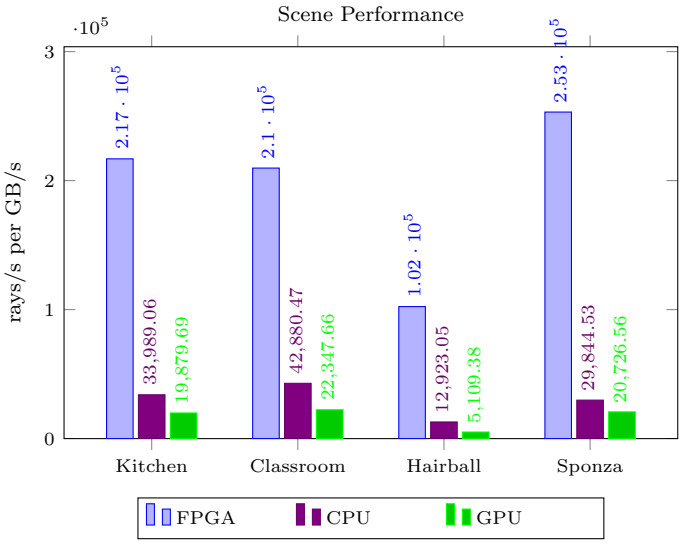

Figure 15: Ray-tracing performance per unit of bandwidth available on the platform and the most efficient CPU implementation over all scenes.

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 [40, 41]. With the older-technology platform energy efficiency already better than a more recent CPU, use of a newer FPGA process technology will provide greater relative energy efficiency.

9.2. Available Bandwidth

With memory earlier confirmed as a bottleneck to platform performance in cache evaluation and also commonly a bottleneck to performance on other SIMD and MIMD architectures [11], bandwidth efficiency gives an indication of how well an architecture makes use of the bandwidth available to it. The FPGA board has 1.2 GB/s memory bandwidth (between FPGA and on board DDR memory), the CPU has 25.6 GB/s memory bandwidth and the GPU has 12.8 GB/s of memory bandwidth, meaning that the dual-channel DDR3 attached to the CPU provides just over 21 times the memory bandwidth than is available to the FPGA.

Bandwidth efficiency for each platform is measure by setting the rays per second performance into relation with the available bandwidth. The resulting unit 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 15. The results show that FPGA has significantly better bandwidth efficiency than CPU and GPU over all scenes, with over six times the bandwidth efficiency of CPU for Kitchen, five times the efficiency for Classroom, nearly eight times the efficiency for Hairball and over eight times the efficiency for Sponza.

In summary, the FPGA platform is not the fastest for the tested scenes, but the most efficient, both in terms performance per energy and bandwidth. 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 for FPGAs, 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.

10. Conclusion

This paper proposed a general and flexible FPGA ray tracing platform to easily and efficiently explore hardware acceleration of ray tracing on FPGAs. It is integrated with a modern, acceleration-focused and currently developed rendering engine, LuxRender.

Early results with this platform indicated that memory bandwidth is the main bottleneck preventing scalability. Hence, memory access patterns to the acceleration hierarchy and scene primitives during ray tracing were explored and analysed. This revealed that nodes are more important to cache than primitives,

which lead to the proposal of a domain-specific cache design for ray tracing distinguishing between nodes and primitives. The study also showed that nodes on higher levels of the acceleration hierarchy are accessed far more frequently than those on lower levels. Consequently, a novel node cache replacement policy, based not on recent use, but on a node's level on the acceleration hierarchy, was proposed. Evaluation showed 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.

An experimental evaluation of the proposed platform by ray tracing four typical scenes on real FPGA hardware was performed, investigating the impact of a cache on platform performance. 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. Lastly we put the performance into relation with a CPU and GPU platform, which demonstrated the strong promise of the FPGA-based approached in terms of energy efficiency and performance per available bandwidth.

11. References

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

[2] NVIDIA, Nvidia collaborates with weta to accelerate visual effects for avatar, `http://www.nvidia.com/object/wetadigital_avatar.html`, accessed: 2014-02-12 (2010).

[3] RenderStream, Costs associated with high computation render farms, `http://blog.renderstream.com/2009/09/how-many-computers-do-you-need-in-your-farm/` (09 2009).

[4] J. Carabano, F. Dios, M. Daneshtalab, M. Ebrahimi, An exploration of heterogeneous systems, in: Reconfigurable and Communication-Centric

Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on, 2013, pp. 1–7. doi:10.1109/ReCoSoC.2013.6581542.

[5] LuxRender, LuxRender - GPL physically based renderer (1 2014) [cited 6/1/14].
URL http://www.luxrender.net/en_GB/index

[6] Xilinx®, Xilinx and pico computing announce industry's first 15gb/s hybrid memory cube interface, http://prn.to/1meMc0B (2014).

[7] T. Möller, B. Trumbore, Fast, minimum storage ray-triangle intersection, Journal of Graphics, GPU, and Game Tools 2 (1) (1997) 21–28.

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

[9] Intel®, Intel® instruction set architecture extensions, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (2013).

[10] Intel®, The intel® xeon phi product family, http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf (2013).

[11] T. Aila, S. Laine, Understanding the efficiency of ray traversal on gpus, in: Proc. High-Performance Graphics 2009, 2009.

[12] R. Overbeck, R. Ramamoorthi, W. Mark, Large ray packets for real-time whitted ray tracing, in: Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on, 2008, pp. 41–48. doi:10.1109/RT.2008.4634619.

[13] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, I. Wald, Packet-based whitted and distribution ray tracing, in: Proceedings of Graphics Interface 2007, GI '07, ACM, New York, NY, USA, 2007, pp.

177–184. doi:10.1145/1268517.1268547.
URL http://doi.acm.org/10.1145/1268517.1268547

[14] H. Dammertz, J. Hanika, A. Keller, Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays, Computer Graphics Forum 27 (4) (2008) 1225–1233. doi:10.1111/j.1467-8659.2008.01261.x.

[15] I. Wald, C. Benthin, S. Boulos, Getting Rid of Packets – Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2008, 2008.

[16] M. Ernst, G. Greiner, Multi bounding volume hierarchies, in: Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on, 2008, pp. 35–40. doi:10.1109/RT.2008.4634618.

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

[18] D. Kopta, J. Spjut, E. Brunvand, A. Davis, Efficient mimd architectures for high-performance ray tracing, in: Computer Design (ICCD), 2010 IEEE International Conference on, 2010, pp. 9–16. doi:10.1109/ICCD.2010.5647555.

[19] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, T.-D. Han, Traversal and intersection engine for hardware accelerated ray tracing, ACM Trans. Graph. 30 (6) (2011) 160:1–160:10. doi:10.1145/2070781.2024194.

[20] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, T.-D. Han, Sgrt: A mobile gpu architecture for real-time ray tracing, in: Proceedings of the 5th High-Performance Graphics Conference, HPG '13, ACM, New York, NY, USA, 2013, pp. 109–119. doi:10.1145/2492045.2492057.
URL http://doi.acm.org/10.1145/2492045.2492057

[21] C. Benthin, I. Wald, S. Woop, M. Ernst, W. Mark, Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture, Visualization and Computer Graphics, IEEE Transactions on 18 (9) (2012) 1438–1448. doi:10.1109/TVCG.2011.277.

[22] J. Schmittler, I. Wald, P. Slusallek, Saarcor: A hardware architecture for ray tracing, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '02, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002, pp. 27–36.
URL http://dl.acm.org/citation.cfm?id=569046.569051

[23] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, P. Slusallek, Real-time ray tracing of dynamic scenes on an fpga chip, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '04, ACM, New York, NY, USA, 2004, pp. 95–106. doi:10.1145/1058129.1058143.

[24] S. Woop, J. Schmittler, P. Slusallek, RPU: a programmable ray processing unit for realtime ray tracing, ACM Trans Graphics 24 (3) (2005) 434–444.

[25] 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, 2007, pp. 359 –363. doi:10.1109/HPCMP-UGC.2007.79.

[26] J. Fender, J. Rose, A high-speed ray tracing engine built on a field-programmable system, in: Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on, 2003, pp. 188–195. doi:10.1109/FPT.2003.1275747.

[27] A. Nery, N. Nedjah, F. Franca, L. Jozwiak, A parallel ray tracing architecture suitable for application-specific hardware and GPGPU implementations, in: 14th Euromicro Conf Digital System Design, 2011, pp. 511 –518. doi:10.1109/DSD.2011.71.

[28] J. Lee, Y. Shin, W.-J. Lee, S. Ryu, J. Kim, Real-time ray tracing on coarse-grained reconfigurable processor, in: Field-Programmable Technology (FPT), 2013 International Conference on, 2013, pp. 192–197. doi:10.1109/FPT.2013.6718352.

[29] J. ho Nah, J. suk Heo, W.-C. Park, T.-D. Han, A split node cache scheme for fast ray tracing, in: Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on, 2008, pp. 186–186. doi:10.1109/RT.2008.4634649.

[30] Xilinx®, Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions (September 2011).
URL `http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf`

[31] M. McGuire, Computer graphics archive (August 2011).
URL `http://graphics.cs.williams.edu/data`

[32] C. Kelemen, L. Szirmay-Kalos, G. Antal, F. Csonka, A simple and robust mutation strategy for the metropolis light transport algorithm, in: Computer Graphics Forum, 2002, pp. 531–540.

[33] I. Wald, On fast Construction of SAH based Bounding Volume Hierarchies, in: Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing, 2007.

[34] D. A. Patterson, J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[35] L. Johnson, Improving ddr sdram efficiency with a reordering controller, XCell 69 (2009) 38–41.

[36] Intel®, Intel® core™ i5-3470 processor (6m cache, up to 3.60 ghz), `http://ark.intel.com/products/68316/` (6 2012).

[37] AMD, Ati radeon hd 5450 graphics, http://www.amd.com/en-us/products/graphics/desktop/5000/5450 (10 2014).

[38] Intel®, Optimizing performance with intel advanced vector extensions, http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf (9 2014).

[39] Xilinx®, Xilinx power tools tutorial, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug733.pdf (3 2010).

[40] Xilinx®, Virtex-7 fpga device family, http://www.xilinx.com/publications/prod_mktg/Virtex7-Product-Brief.pdf (2 2012).

[41] S. Trimberger, Keynote: Beyond moore. beyond programmable logic., in: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, 2012.
URL http://www.fpl2012.org/Presentations/Keynote_Steve_Trimberger.pdf