A Hardware Unit for Fast SAH-optimised BVH Construction

Michael J. Doyle, Colin Fowler and Michael Manzke

Trinity College Dublin*

Abstract

Ray-tracing algorithms are known for producing highly realistic images, but at a significant computational cost. For this reason, a large body of research exists on various techniques for accelerating these costly algorithms. One approach to achieving superior performance which has received comparatively little attention is the design of specialised ray-tracing hardware. The research that does exist on this topic has consistently demonstrated that significant performance and efficiency gains can be achieved with dedicated microarchitectures. However, previous work on hardware ray-tracing has focused almost entirely on the traversal and intersection aspects of the pipeline. As a result, the critical aspect of the management and construction of acceleration data-structures remains largely absent from the hardware literature.

We propose that a specialised microarchitecture for this purpose could achieve considerable performance and efficiency improvements over programmable platforms. To this end, we have developed the first dedicated microarchitecture for the construction of binned SAH BVHs. Cycle-accurate simulations show that our design achieves significant improvements in raw performance and in the bandwidth required for construction, as well as large efficiency gains in terms of performance per clock and die area compared to manycore implementations. We conclude that such a design would be useful in the context of a heterogeneous graphics processor, and may help future graphics processor designs to reduce predicted technology-imposed utilisation limits.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing;

Keywords: ray-tracing, ray-tracing hardware, bounding volume hierarchy, BVH, SAH

Links: DL PDF

1 Introduction

Ray-tracing algorithms are known for producing highly realistic images, but also for their high computational demands. This has motivated a large body of research on techniques for accelerating such algorithms, on both CPU and GPU platforms. Perhaps the most effective acceleration method known for ray-tracing is the use of *acceleration data-structures*. Among the most widely used acceleration data-structures are *bounding volume hierarchies* (BVHs) and *kd-trees*. These structures provide a spatial map of the scene that can be used for quickly culling away superfluous intersection tests. The efficacy of such structures in improving performance has made them an essential ingredient of any interactive ray-tracing system. When rendering dynamic scenes, these structures must be rebuilt or updated over time, as the spatial map provided by the structure is invalidated by scene motion.

For dynamic scenes, the proportion of time spent building these data-structures represents a considerable portion of the total *time to image*. A great deal of research has therefore been directed to the goal of faster construction of these essential structures. Much of the recent research has looked to parallel construction on both multicore and manycore platforms [Wald 2007; Pantaleoni and Luebke 2010; Wald 2012]. Such implementations have demonstrated that a great deal of parallelism is available in this task, and have achieved large performance improvements over serial algorithms.

Another proposed approach to achieving high ray-tracing performance is with the use of specialised hardware devices. Little work to date has been performed in this area, despite a number of researchers demonstrating considerable raw performance and efficiency gains with a variety of programmable [Spjut et al. 2009], fixed-function [Schmittler et al. 2004] and hybrid architectures [Woop et al. 2005]. So far, these devices have relied on CPU support for acceleration data-structure construction, or have resorted to refitting operations, placing restrictions on the extent to which motion is supported and/or degrading rendering performance. Therefore, the construction of acceleration data-structures in hardware remains an open problem.

Previous research has noted that high-quality acceleration data-structure construction is quite compute intensive and scales very well on parallel architectures [Lauterbach et al. 2009; Wald 2012]. We thus hypothesize that a custom hardware solution to acceleration data-structure construction would represent a highly efficient alternative to execution of the algorithm on a multicore CPU or manycore GPU if used in the context of a heterogeneous graphics processor.

Recent research argues that multicore scaling is power limited due to the failure of Dennard scaling [Esmaeilzadeh et al. 2011]. Esmaeilzadeh et al. show that at 22nm, 21% of a fixed-size chip must be powered off, and at 8nm, it could be more than 50%. This had led some to coin the term *dark silicon*, for logic which must remain idle due to power limitations. In response to this, some researchers have proposed that efficient custom microarchitectures could help heterogeneous single-chip processors to reduce future technology-imposed utilisation limits [Venkatesh et al. 2010; Chung et al. 2010]. It is now a matter of identifying the most suitable algorithms for custom logic implementation for the ages of dark silicon.

To this end, we propose a design for what we believe to be the first dedicated microarchitecture for the construction of an acceleration data-structure. Such structures are fundamental to much of computer graphics and simulation technology. The fast construction of such data-structures is of particular interest to the ray-tracing community, and for this reason, we focus on construction of high-quality, binned SAH bounding volume hierarchies for ray-tracing. Such techniques

*[mjdoyle, cfowler, michael.manzke]@scss.tcd.ie

ACM Reference Format
Doyle, M., Fowler, C., Manzke, M. 2013. A Hardware Unit for Fast SAH-Optimised BVH Construction. ACM Trans. Graph. 32, 4, Article 139 (July 2013), 10 pages. DOI = 10.1145/2461912.2462025 http://doi.acm.org/10.1145/2461912.2462025.

DOI: http://doi.acm.org/10.1145/2461912.2462025

can often produce hierarchies of quality quite close to SAH sweep build methods, while being suitable in interactive contexts. Additionally, many other BVH builders depend on binned SAH methods in some way, including the spatial split oriented SBVH [Stich et al. 2009], and also hybrid SAH/HLBVH builders [Garanzha et al. 2011]. As such, given that the binned SAH is commonly used in a variety of construction algorithms, and given that it is a competitive builder in its own right, we consider it to be the primary challenge of any new implementation strategy, and choose to focus on it in this paper.

We propose that a purpose-built microarchitecture for acceleration data-structure construction will exhibit greater performance and efficiency than software implementations executed on current multicore and manycore processors. We can summarise the motivation for including hardware support for binned SAH BVH construction in a dedicated ray-tracing processor or other heterogeneous graphics processor by noting the advantages of our microarchitecture:

- **Accelerated Performance** Raw construction performance improvements of up to 10× relative to current binned SAH BVH software implementations, and significant performance improvements over some less accurate SAH builders. This is achieved despite the fact that our results are measured with large clock frequency, bandwidth, and die area disadvantages compared to current multicore and manycore processors.
- **Greater Efficiency** Since our design achieves a performance improvement with much fewer hardware resources, it represents a large efficiency improvement over existing software approaches. Existing software methods scale quite well, and require engaging a large amount of programmable resources to achieve optimal performance. Utilising our design in a heterogeneous single-chip processor would minimise the hardware resources needed to achieve fast builds. Since BVH construction is a core algorithm in ray-traced rendering, our design could have performance implications not only for the BVH build, but also for the rest of the application pipeline.
- **Low Memory Bandwidth and Footprint** Our design requires much less bandwidth to main memory and requires a small memory footprint for hierarchy construction compared to software approaches. These bandwidth savings could be used to support the additional parallelism already stated.
- **Binned SAH Rebuild Speed Competitive with Updating** Our architecture is quite scalable and can achieve full binned SAH rebuilds with performance similar to many software updating strategies, while remaining within modest area and bandwidth costs. This ensures higher quality trees, much fewer edge cases and suitability for applications where updating may not be appropriate (e.g. photon mapping). Full rebuilds also do not limit scene motion in any way, in contrast to updating schemes.

By this reasoning, we believe there is significant motivation for including hardware support for acceleration data-structure construction in a heterogeneous graphics processor. We envision that such logic could coexist with and complement programmable components to form a hybrid rendering system. This is in fact quite similar to how current rasterization-based GPUs operate.

2 Background

The bounding volume hierarchy (BVH) is one of the most widely used acceleration data-structures in ray-tracing. This can be attributed to the fact that it has proven to represent a good compromise between traversal performance and construction time. In addition, fast refitting techniques are available for BVHs [Lauterbach et al. 2006; Kopta et al. 2012], making them highly suitable for deformable geometry.

The classical BVH is typically a binary tree in which each node of the tree represents a bounding volume (typically an *axis-aligned bounding box (AABB)*) which bounds some subset of the scene geometry. The AABB corresponding to the root node of the tree bounds the entire scene. The two child nodes of the root node bound disjoint subsets of the scene, and each scene primitive will be present in exactly one of the children. The two child nodes can be recursively subdivided in a similar fashion until a termination criterion is met. Typical strategies include terminating at a certain number of primitives, or at a maximum tree depth.

For ray-tracing, many BVH construction algorithms follow a *top-down* procedure. Starting with the root node, nodes are split according to a given splitting strategy and child nodes produced which are further subdivided until a leaf node is reached. The choice of how to split the nodes can have a profound effect on rendering efficiency. Perhaps the most widely used strategy is the *surface area heuristic (SAH)*. The SAH estimates the expected ray traversal cost C for a given split, and can be written as:

$$C(V \rightarrow \{L, R\}) = K_T + K_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right)$$

where V is the original volume, V_L and V_R are the subvolumes of the left and right child nodes, N_L and N_R are the number of primitives in the left and right child nodes, and SA is the surface area. K_I and K_T are implementation-specific constants representing the cost of ray/primitive intersection and traversal respectively. The SAH can be evaluated for a number of split candidates and the best candidate chosen. *Sweep builds* sort all primitives along a given axis and evaluate each possible sorted primitive partitioning, which yields highly efficient trees, but at a construction cost too high for realtime performance. *Binned SAH* algorithms approximate this process by evaluating the SAH at a small number of locations (typically 16 or 32) spread evenly over the candidate range. The binned SAH algorithm achieves much faster build times, while preserving high rendering efficiency, and is therefore more suitable for realtime application.

2.1 Parallel Construction of BVHs

The construction of BVHs for ray-tracing is conceptually a very parallel problem. Parallelisation schemes to date have utilised many forms of parallelism, including assigning subtrees to individual cores, building single nodes using multiple cores, and parallel breadth-first schemes. Both CPU and GPU approaches have utilised such techniques. Our microarchitecture described in Section 3 follows similar principles, and since we wish to compare to these platforms, we provide a brief overview of the key work to date.

Early parallel construction algorithms targeted multicore CPUs [Wald 2007]. Wald's algorithm distinguishes between the upper and lower nodes in the tree, utilising a more data-parallel approach for the upper nodes and a task parallel per-subtree scheduling for lower nodes. In addition to construction, parallel refitting techniques for BVHs have been shown on multicore CPUs [Lauterbach et al. 2006].

More recent work on multicore BVH builds include the Intel Embree set of ray-tracing kernels [Ernst 2012]. The Embree project includes support for SAH BVHs of several branching factors and is highly optimised for current generation CPUs.

A breadth-first parallelisation of binned SAH BVH construction has been shown to be effective on GPUs [Lauterbach et al. 2009]. Each

child node generates a new thread in the build, allowing for a large number of concurrent threads to effectively utilise the GPU. The authors also propose an alternative hybrid LBVH/SAH scheme to extract more parallelism at the top of the tree. This work was extended to the Hierarchical LBVH, to take greater advantage of data coherence [Pantaleoni and Luebke 2010]. Other work on HLBVH includes faster and more efficient implementations [Garanzha et al. 2011; Karras 2012].

A recent implementation of binned SAH BVH construction targets the Intel MIC architecture [Wald 2012]. The tested architecture in this work consists of 32 x86 cores operating at a frequency of 1GHz. Algorithmically, this implementation resembles earlier work [Wald 2007]. A data-parallel approach is used for large nodes, and smaller subtrees are assigned to individual threads. Furthermore, data quantization of primitives is employed to improve cache performance, at reasonable hierarchy quality degradation.

Sopin et al. describe another fast approach to binned SAH BVH construction on the GPU [Sopin et al. 2011]. Like other algorithms, this approach distinguishes between different node sizes for the purposes of more efficiently assigning tasks to the GPU architecture, utilising a larger number of cores for upper nodes, and assigning fewer cores per node as the nodes become smaller. This work is among the fastest published implementations of the binned SAH BVH construction algorithm.

Finally, the OptiX ray-tracing engine [Parker et al. 2010] provides developers with highly-optimised BVH builders for both CPU and GPU platforms, including SBVH and LBVH-type hierarchies.

2.2 Ray-tracing Hardware

Previous work on ray-tracing hardware has included fixed-function, programmable, and hybrid fixed-function/programmable designs. We first examine the major fixed-function work and then move onto programmable and hybrid solutions.

The *SaarCOR* architecture is a fixed-function design for ray tracing of dynamic scenes [Schmittler et al. 2004]. The architecture utilises multiple units in parallel, each traversing wide packets with a kd-tree data-structure. Each unit operates on multiple packets in a multithreaded manner to hide memory latency. An FPGA prototype of this architecture is presented, albeit requiring CPU support for data-structure construction.

More recent work on fixed-function ray-tracing hardware includes the T&I engine [Nah et al. 2011]. It is a MIMD style processor which operates on single rays, rather than packets. A *ray dispatcher* unit generates rays, which are passed to 24 *traversal units* which utilise a kd-tree data-structure. On encountering a leaf, the *list units* fetch primitives for intersection. Intersection is split into two units (IST 1 & 2) such that if a ray fails initial tests in IST1, data need not be fetched for the rest of the procedure in IST2. Each unit possesses a cache, and on cache misses, rays are postponed in a *ray accumulation unit* which collects rays waiting on the same data. Running at 500MHz, simulations indicate that 4 T&I engines together can exceed the ray throughput of a GTX 480 GPU by around 5-10×. A ray-tracing GPU utilising the T&I engine, coupled with reconfigurable hardware shaders and a multicore ARM chip for data-structure construction, has also recently been proposed [Lee et al. 2012].

Hybrid fixed-function/programmable ray-tracing architectures have also been proposed, such as the Ray Processing Unit (RPU) [Woop et al. 2005]. Each RPU consists of multiple programmable *Shader Processing Units* (SPUs), which utilise a vector instruction set. Each SPU is multithreaded and avoids memory latency by switching threads when necessary. Each SPU can be used for a variety of purposes, including intersection tests and shading. SPUs are grouped into *chunks* containing a small number of units. All SPUs in a chunk operate together in a lock-step manner. Multiple asynchronous chunks work in parallel to complete a task. Coupled with each SPU is a fixed-function *Traversal Processing Unit*, which can be accessed by the SPUs via the instruction set and utilises a kd-tree data-structure. A later version of this work, the DynRT architecture [Woop et al. 2006] is designed to provide limited support for dynamic scenes by refitting, but not rebuilding, a B-KD data-structure.

The TRaX architecture represents some of the most recent work on ray-tracing hardware [Spjut et al. 2009]. The design is programmable and consists of a number of thread processors which possess their own private functional units, but which are also connected to a group of shared functional units. Each software thread corresponds to a ray, and the design is optimised for single rays, rather than relying on coherent packets. The advantage of this architecture is that it is entirely programmable and yields good performance for ray-tracing compared to GPUs.

The *Mobile Ray-tracing Processor (MRTP)* [Kim et al. 2012] is a programmable design which takes a unique hardware approach to solving SIMT/SIMD utilisation problems due to divergent code. The basic architecture consists of three *reconfigurable stream multiprocessors (RSMPs)* which are used to execute one of three kernels: ray traversal, ray intersection and shading. Kernels can adaptively be reassigned to RSMPs to enable load balancing. Each RSMP is a SIMT processor consisting of 12 *Scalar Processing Elements (SPE)*. Each SPEs can be reconfigured into either a 12-wide regular scalar SIMT operation, or a 4-wide 3-vector SIMT operation. To improve datapath utilisation due to code divergence, the system uses the regular scalar SIMT mode for traversal and shading, and reconfigures into the vector mode for triangle intersection.

Finally, a number of commercial ventures utilising dedicated ray-tracing hardware have been founded, including ArtVPS [Hall 2001] and Caustic Graphics [Caustic Graphics 2012] which released cards aimed at accelerating ray-traced rendering. These cards appear also to focus on hardware for the actual tracing portion of the pipeline. However, limited technical information is publicly available on these products.

3 Microarchitecture

Figure 1a is a top level diagram showing the major units of our microarchitecture. Visible in this diagram is a DRAM interface consisting of a number of *RAM pairs*. Each RAM pair consists of two memory channels. Before construction begins, scene primitives are divided over the RAM pairs, with one RAM in each pair holding primitives. Below the RAM pairs, is the *upper builder*. The upper builder reads and writes directly to DRAM and is responsible for constructing the upper levels of the hierarchy. Connected to the upper builder is one or more *subtree builders*. The subtree builders are responsible for constructing the lower levels of the hierarchy. The upper builder continues building until a node smaller than a predetermined size is found (typically, several thousand primitives). The primitives corresponding to this node are then loaded into one of the subtree builders, which contain a set of high bandwidth/low latency on-chip internal memories. The subtree builder builds a complete subtree from these primitives. Once all primitives are passed to a subtree builder, the upper builder continues building its upper hierarchy, passing further subtrees to the other subtree builders, stalling if none are available. The upper and subtree builders therefore operate in parallel.

The upper and subtree builders are largely the same hardware, except that the upper builder interacts with external DRAM, while the subtree builders interacts with its internal memory buffer. The core

logic of the subtree builder is actually mostly a superset of the upper builder. Therefore, we first describe in detail the subtree builder, and then describe how it differs from the upper builder. An earlier, limited prototype of the subtree builder was briefly outlined in our poster [Doyle et al. 2012].

3.1 Subtree Builder

The architecture of the subtree builder is shown in Figure 1b. A relatively small instantiation is illustrated, so as to maintain clarity. The architecture is designed to operate on the AABBs of scene primitives, as is common with other hierarchy builders, and is therefore suitable for any primitive type for which an AABB can be calculated.

The subtree builder implements a typical binned SAH recursive BVH construction algorithm, in line with established best practices [Wald 2007]. The subtree builder consists of a number of units which implement the various stages of this recursive algorithm. The first units of interest are the *partitioning units*. Two partitioning units are visible in Figure 1b, labelled $PUnit_0$ and $PUnit_1$. The purpose of the partitioning units is, given a split in a certain axis, to read a vector of primitives from the internal buffers and partition those primitives into one of two new vectors, depending on which side of the splitting plane they reside.

Each partitioning unit is connected to a pair of *primitive buffers*. Two pairs are visible in Figure 1b, and are labelled *Buff0* and *Buff1*. The primitive buffers are a set of on-chip, high bandwidth/low latency buffers (similar to a cache memory). The purpose of the primitive buffers is to hold primitive AABBs as they are processed by the partitioning units. Each buffer pair is hardwired to one partitioning unit. Primitive buffers are organised in pairs to facilitate swift partitioning of AABBs. When the upper builder loads a set of scene primitives into the subtree builder, the primitives are distributed to one of the buffers from each buffer pair, with the opposite buffer in each pair left empty. The partitioning units read AABBs from one of buffers and rewrite the AABBs in the new partitioned order to the opposite buffer. On the next recursive partitioning, the roles of the buffers are reversed, and the primitives are read from the buffer they were last written. This back-and-forth action allows concurrent reading and writing of primitives which leads to swift primitive partitioning. The data width of the interface to these buffers is set large enough for a full primitive AABB to be read in each cycle. They could also be implemented with several narrower memories in parallel.

Below the partitioning units in Figure 1b is the logic which determines the SAH split for the current node. The subtree builder is capable of searching all three axes concurrently for the lowest cost split. We implement the SAH determination with two types of unit: a *binning unit* and an *SAH calculator*. Each partitioning unit is connected to three binning units, one for each axis (labelled Bin x/y/z). The binning units latch data from the output of primitive buffers, and also keep track of the AABB of the current node. The binning operation is performed by calculating the centre of the primitive AABBs and then binning this centre point into the AABB of the current hierarchy node. The binning units output the chosen bin locations in all three axes, and also the original primitive AABB which was used to calculate those bin locations.

Below the binning units in Figure 1b, are the SAH calculators (8 are pictured). Primitive AABBs and their chosen bin positions are fed into the SAH calculators which accumulate an AABB and a counter for each bin in each axis. Once all primitives are accumulated, the SAH calculators evaluate the SAH cost for each possible split, and output the lowest cost split found. Once the split has been chosen, it is fed back to the partitioning units which partition the primitives in their primitive buffers according to the split. The SAH evaluation is expensive, and our design is multithreaded to hide the latency of this unit.

3.2 Sequence of Operations

Now that we have established the function of each of the major units, we now describe the sequence of operations that the subtree builder performs to generate a hierarchy. Sequencing of operations is performed by the *Main Control Logic* shown on the left of Figure 1b. This control logic is not critical to our contribution and we therefore restrict discussion of this to a more high level treatment as given in the following sections.

Before the subtree builder is activated, the upper builder loads AABBs combined with their primitive IDs (as a single data word) into one of the primitive buffers in each buffer pair in a round-robin assignment (i.e. the left buffer only of each pair, leaving the right empty). This results in an approximately equal number of primitives per buffer pair, facilitating load balancing. Primitive IDs are always attached to their associated AABBs as they move between primitive buffers, and are used for tree output. The bounding AABB of all primitives is also loaded into a register at this point. Once all primitives are loaded, an *initial setup phase* is run. All partitioning units are signalled to dump the full contents of their primitive buffers into the binning units. The results of the binning units are fed into a single SAH calculator which calculates the split for the root of the hierarchy. The output of the SAH calculator is the chosen SAH split, the chosen axis and, importantly, the AABBs and primitive counts of the two resulting child nodes. Once these values are obtained, the *main construction loop* can proceed.

3.2.1 Main Construction Loop

The initial split phase produces the split for the root node. Each partitioning unit is then instructed to begin the main construction loop of the builder. Each partitioning unit possesses in its buffer pair a subset of the total primitives which must be partitioned according to the split. Each of the partitioning units cooperate to partition all primitives in a data-parallel manner. Each partitioning unit reads its subset of primitives pertaining to the current node from one of the buffers in its buffer pair. The partitioning unit then determines on which side of the current splitting plane each primitive lies, and then writes the primitives out in partitioned order into the opposite buffer. Partitioning is achieved by maintaining two address registers, a *lower* and an *upper* register, inside each partitioning unit. The lower and upper registers begin at the bottom and top address respectively of the subset of primitives that belong in the node currently being processed. These registers are then multiplexed onto the address of the primitive buffer as appropriate.

After a partition, each partitioning unit has two sublists of primitives residing in its primitive buffers. To continue the recursive procedure, processing must continue with one of these sublists, with the other placed on a stack for future processing. Since we have several partitioning units all partitioning a subset of the current node's primitives in their respective buffers, we really have several partitioned lists, which when added together form the full list. To keep track of this information, we employ a *wide stack*. Wide stack elements include the full AABB of the pushed node, and also separate primitive ranges for each primitive buffer pair detailing where all primitives reside. The stack also stores on which "side" of the primitive buffer pair the primitives of interest reside.

When the partitioning units encounter a leaf, instead of recursing again and writing the primitives back into the opposite buffer, they write the primitive IDs into separate output FIFOs. Tree nodes are also written into similar FIFOs. Nodes and primitive IDs are then

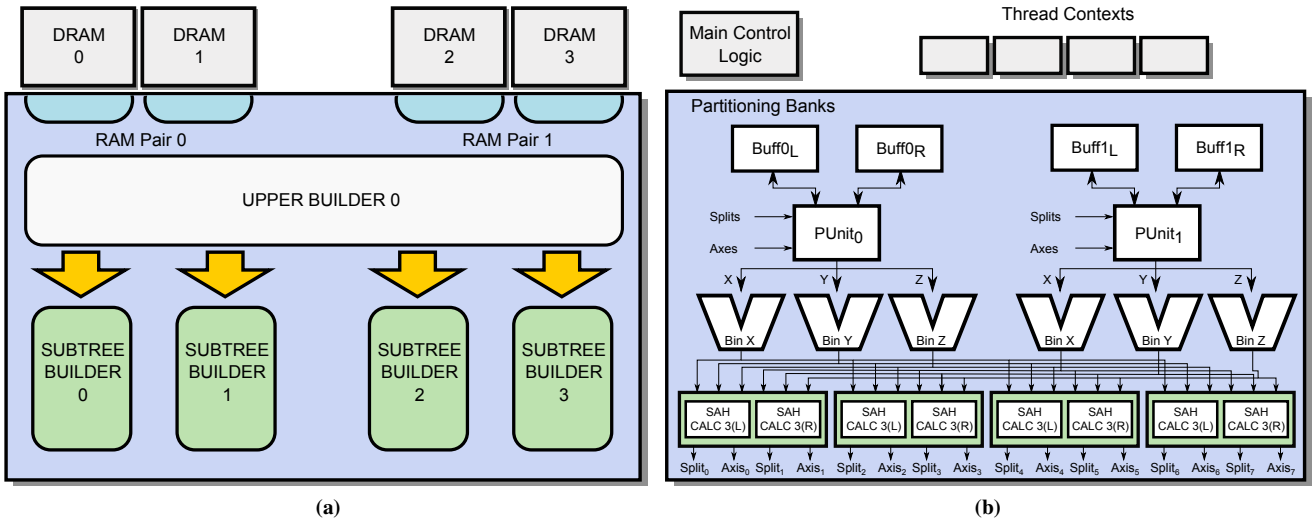

Figure 1: *(a) A top level overview of our binned SAH BVH construction hardware, showing memory interfaces, upper and subtree builders. (b) Detailed diagram of the subtree builder microarchitecture.*

collected from these FIFOs and written out to RAM.

3.2.2 Concurrent Calculation of the Next Split

In addition to partitioning the primitives of the current node, it is also necessary to calculate the splits for these two new nodes. The fact that we are partitioning means that we already have the SAH split information which includes the AABBs of the two resulting child nodes. Therefore, we have all the information necessary to begin binning primitives into the new children concurrently while they are being partitioned. During partitioning, primitives are not only written into the opposite buffer, but are also fed into the binning units. The binning units bin each primitive into either the left or right child, depending on which side of the partition it belongs to, by multiplexing the correct values into the pipeline.

The binning units output the bin decisions and primitive AABBs which are then fed into one of the SAH calculator pairs as shown at the bottom of Figure 1b. SAH calculators are placed in pairs, one for each side of the split. If a primitive was on the left side of the split in the previous node, it is fed into the left SAH calculator of the pair, otherwise the right. Both calculators in a pair operate concurrently. As each partitioning unit processes a subset of the node's primitives, each SAH calculator must monitor the output of each binning unit (each set of three binning units are assigned to a partitioning unit, which is assigned to a primitive buffer pair). After calculating the splits, processing continues with a valid child, normally the left, while the right split information is pushed to the stack for later processing. If the node is a leaf, the stack is popped. This stack contains the split, the axis, the AABB of the node, the resulting child AABBs and primitive counts, the ranges in the primitive buffers corresponding to the node, and a single bit indicating on which side of the primitive buffers the node's primitives reside.

3.2.3 Multithreaded SAH Calculation

Once the partitioning units pass all of their primitives into the binning units, they must wait for all of them to be binned and for the SAH calculator to return the next split so that they can begin partitioning again. In our implementation, the total combined latency of the binning and SAH units is approximately 40 cycles. Stalling would represent a large performance penalty because it would be incurred on every node of the tree. Instead, we hide the latency of the SAH calculation, by taking a multithreaded approach that utilises several SAH calculators. We allow context for multiple threads to be maintained in the system, as shown in the upper half of the diagram. Initially, there is only one thread in the system, representing the root node. As new child nodes are created, we spawn these off as new threads, until a predetermined number of threads is reached. Each thread context stores the ranges in each of the primitive buffers of the primitives in the thread, a split, a stack and stack pointer, an axis and a node AABB (thread elements are similar to stack elements). The new threads represent different subtrees. Each new thread that is created is assigned to a pair of SAH calculators.

Each partitioning unit will hold a subset of the primitives in each thread due to the round-robin assignment in the beginning. When a partitioning unit finishes partitioning a node, instead of stalling for the SAH calculation, it can switch context to the next thread in the system. Once it has completed the last thread, it can return to the first thread for which the split will now be ready. The round-robin assignment means that partitioning units are therefore almost always utilised (even when only one thread is present) and in addition to this, the system is load balanced as the assignment leads to a roughly equal amount of primitives belonging to each thread in each partitioning unit.

3.2.4 Upper Builders

As already explained, the upper builder and the subtree builder are very similar. The upper builder also contains partitioning units, binning units and an SAH calculator pair which are only modified slightly from their counterparts in the subtree builder. We can now explain the difference between the subtree builder and the upper builders. The principal difference is that the upper builder contains no multithreading support (only one thread context) and utilises the RAM pairs in place of the partitioning buffer pairs. It achieves efficient use of DRAM by reading primitives in bursts and buffering writes into bursts before they are requested. Multithreading is unnecessary for the upper builder because it only constructs the uppermost nodes of the hierarchy which contain possibly thousands of primitives which are read in long streaming consecutive reads. Therefore

the stall incurred by waiting on the SAH calculator (around 40 cycles) is negligible. It is therefore not necessary to spend resources on multithreading for the upper builder.

3.2.5 SAH Calculators

We now examine the SAH calculators in more detail. A block diagram for this unit is shown in Figure 2. The input to the SAH calculator is a vector of AABBs and a vector of bin decisions. Each AABB and each bin of these two vectors comes from a separate binning unit. The first stage of the SAH calculator consists of multiple blocks of *buffer/accumulators*, labelled (1) in the diagram. One block exists for each binning unit in the design. There are three buffer/accumulators per block, one for each axis. The purpose of the buffer/accumulator is to take a sequence of primitive AABBs and bin decisions from the binning units and accumulate the bin AABBs and bin counts from this sequence into a small buffer. As each buffer/accumulator block processes primitives from one binning unit, it computes a partial vector. Our current subtree builder utilises 16 bins per axis, making one buffer accumulator 416 bytes in size.

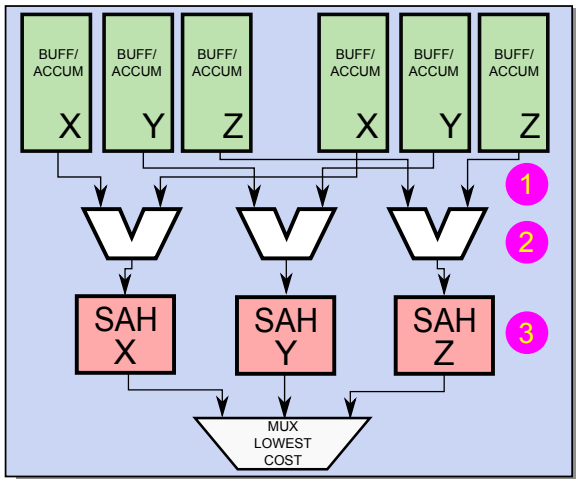

Figure 2: *Architecture of the SAH Calculator.*

Once all primitives have been accumulated, each buffer/accumulator is instructed to dump its contents in order. The contents of all blocks are then merged into a new vector containing the complete bin AABBs and counts by the units labelled (2). Note that there is a separate list of bins for each axis, so we see three of these units in the diagram. These three lists are then fed into three SAH evaluators (one per axis, labelled (3)), which perform the actual SAH evaluation and keep track of the lowest cost split so far. The output of each evaluator is the lowest cost split in that axis. Finally, the global lowest cost split is computed by examining these three values and the SAH calculator signals to the rest of the circuit that the split is ready.

4 Results and Discussion

To evaluate our architecture, we implemented it as a cycle-accurate, synthesizable VHDL model at the RTL level. All results are simulated with Questasim 6.6 from Mentor Graphics. To model our floating-point units, we utilised the Xilinx Floating-Point library available with the Xilinx ISE development software. We chose these cores as they have realistic properties, are proven in real chips, and will allow us in future to quickly adapt our design to reconfigurable systems once adequate equipment becomes available to us. Our simulations allow us to count the exact duration of the computation in clock cycles. Our code is highly configurable, allowing attributes such as the number of partitioning units, the number of threads, bin sizes etc to be altered independently. There is therefore a large number of possible instantiations of the subtree builder. We present a "standard instantiation" for each subtree builder which utilises four partitioning units and sixteen SAH calculators (eight threads). Primitive buffers are set to hold 2048 primitives each, yielding a maximum capacity for each subtree builder of 8192 primitives.

We model these buffers with Xilinx Block RAM primitives, which are single ported RAMs with a memory width of 216 bits (one 32-bit floating-point AABB and one primitive ID), a latency of one cycle, and a throughput of one word per cycle. The total capacity of the eight buffers is therefore 432 KB and the maximum internal bandwidth is 216 bytes/cycle. We instantiate two such subtree builders for our performance comparisons in Table 1. For our upper builder, we choose an instantiation that utilises two RAM pairs (four DDR ports) which determines an upper builder with two partitioning units, two binning units and one SAH calculator pair.

We wish to estimate the performance of our design if implemented in a dedicated ray-tracing or other graphics processor. For this reason, we follow the assumptions made by earlier work on ray-tracing hardware [Spjut et al. 2009; Nah et al. 2011] and assume a 200 mm^2 die space @ 65nm and a clock frequency of 500MHz. This is 2.8× lower than the shader cores of a GTX480, which is the part of the GPU used by all hierarchy construction implementations on that platform.

We model our DRAM interfaces with a generic DDR model from DRC computer written in Verilog. This DDR model provides an interface with address and data lines, a read/write signal, burst length etc. Each DRAM at peak is capable of delivering one 192-bit word per cycle and also operates at 500MHz. The total bandwidth to each DRAM in our simulations is just over 11 GB/s, and with our four ports (two RAM pairs) is thus 44GB/s max, although our logic does not request this value for much of the BVH construction. This value is only a fraction of what can be found on a modern mid-range GPU. We intend that our microarchitecture would reside on-chip with the rendering logic and therefore we do not time any communication with a host CPU or GPU. We always bin with 16 bins on all three axes and terminate at four triangles per leaf. We compare to both full binned SAH BVH implementations as well as lower quality hybrid SAH builders. In all cases, we compare to the highest-performing software implementations that exist to our knowledge. Simulating our hardware was a time-consuming process (several days for one build), and so we were unable to build all frames of our animated test scenes (e.g. Cloth). Therefore, we chose the middle keyframe from these animations.

4.1 Performance Analysis

Table 1 summarises our performance results. Our implementation exhibits strong performance relative to the two full binned SAH implementations. We note a raw performance improvement of about 4-10× over these manycore implementations. With HLBVH, a direct comparison is difficult because they are two different algorithms. The original idea of HLBVH was to remove much of the expensive SAH calculation in order to improve performance, while preserving reasonable quality. As a result of this, HLBVH is typically 10-15× faster than binned SAH on the same GPU. Regardless, we see that our architecture is actually faster for the Conference scene than HLBVH when measured by performance per clock cycle (extrapolating from the clock frequency of the GPU and the build time).

Overall, we observe that our implementation can deliver high-

quality, high-performance builds at speeds faster than current many-core implementations. We believe that our high performance is achieved through our low-latency/high-bandwidth primitive buffers delivering very efficient streamed data access for the rest of the circuit, which consists of a set of very fast dedicated units for the expensive SAH evaluation and binning.

4.2 Bandwidth Utilisation

We instrumented our simulations to record the total bandwidth consumed over hierarchy construction. These values are shown in Table 1, and include reads and writes.

Bandwidth figures are typically not given in hierarchy construction papers, and the only figures we could find were those of the original HLBVH [Pantaleoni and Luebke 2010]. We exhibit around 2-3× less bandwidth consumed than this implementation. We build only the uppermost levels of tree in external DRAM, and output the tree during construction. No other values are read or written to DRAM. In addition, our memory footprint is also quite low, with the peak footprint being 2× the scene size, which corresponds to about 40MB for the Dragon scene, excluding the tree itself.

We propose that these bandwidth and footprint savings would be an advantage when running other tasks in parallel with the builder such as concurrent rendering/hierarchy construction.

4.3 Scalability

Figure 3 shows the scaling for the Cloth scene in our builder. We begin with one subtree builder and one RAM pair, and scale to four subtree builders and four RAM pairs, doubling the size each time (i.e. 1, 2 and 4 subtree builders/RAM pairs). Unfortunately, scaling past this point caused our simulations to run out of memory on our test machine. However, as the graph shows, our scalability is quite good over these three instantiations, and is very close to linear within this range. Very little overhead is associated with assigning tasks to subtree builders, and design is naturally load balanced as subtree builders only ask for work when idle.

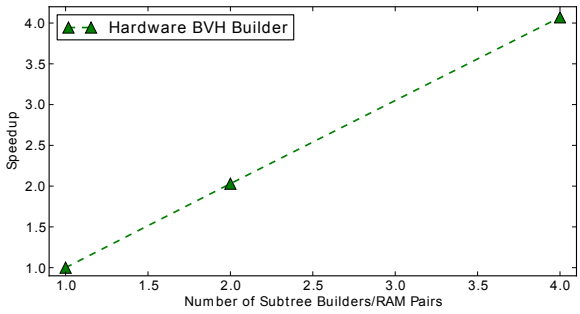

Figure 3: *Scalability of our architecture in the Cloth scene.*

4.4 Tree Cost

We also measure the SAH cost of the trees produced by our builder and compare these costs to competing approaches in Table 2. Our builder precisely follows a classical binned SAH build, with no adjustments, thus ensuring high quality. The only builder in our comparison for which this is also true is Sopin et al., as Wald performs quantization of vertices and HLBVH methods only perform the SAH on a small fraction of the nodes. We express the SAH cost as a ratio to a full SAH sweep build. The sweep build cost is set at 100%, and lower values are better.

As Table 2 shows, we exhibit high tree quality, with tree costs quite close to a full sweep build in many cases. This ensures high efficiency in rendering, which represents a further performance advantage that our design can offer, along with minimising hardware resources and very fast build times. The exception to this is the Conference scene, which is not surprising as other authors have reported lower quality with this scene in binned SAH builders [Wald 2007; Lauterbach et al. 2009].

4.5 Hardware Complexity

Finally, we estimate the hardware resources required for our microarchitecture. We first estimate the resources required for the subtree builder as it represents the majority of the design. Table 3 shows the required number of floating-point cores and register space need for each major design unit in the subtree builder. These values in themselves represent a technology-generic expression of required resources.

Using this tabulation, we follow closely the procedures of earlier work [Nah et al. 2011] and utilise published figures on a 65nm library [Spjut et al. 2009] to perform an area estimate of our design. Table 4 summarises our results. A 4 KB register file is included in this library and we determine that we require register space equivalent to 120 of these. The other major component of our subtree builder is the primitive buffers. We note the similarity between a cache memory and our primitive buffers and model them using the CACTI cache modelling software as a direct-mapped cache (cache size 55296 bytes, line size 27 bytes, associativity 1, number of banks 1, and technology 65nm). This is probably an overestimate, as our primitive buffers are simple RAMs and do not require any caching logic. The CACTI tool reports a size of $0.94mm^2$ for one buffer.

Control logic also requires resources. We once again base our estimates on earlier work [Nah et al. 2011; Muralimanohar et al. 2007], and model this as 35% overhead of the FP cores. Finally, we choose also the same estimate as these authors for wiring overhead at 69%. We thus estimate the total die space of the subtree builder to be $31.88mm^2$ at 65nm, or 16% of our very conservative $200mm^2$ assumed die size, and only around 6% of the GTX 480's die size, (which actually uses a smaller feature size of 40nm [NVIDIA 2010], so our design would probably consume even less than this). Comparing to the T&I engine [Nah et al. 2011], we find that one builder is about 2.6× the size of a T&I core, which consumes $12.12mm^2$. We note that four T&I cores @ 500MHz obtain a 5 - 10× performance increase over a GTX480 GPU implementation in terms of ray throughput. Table 1 shows that we can obtain a similar factor for building binned SAH hierarchies with only two subtree builders. Performing a similar analysis reveals that our upper builder only adds about another $5mm^2$ to this. Therefore, we can conclude that our resource consumption is comparable to this traversal engine.

4.6 General Discussion

4.6.1 Comparison with Refitting

We believe it is important to consider the advantages of our system compared to refitting operations. For deformable scenes, refitting methods are quite useful, but exhibit a few drawbacks. Firstly, refitting usually results in lower quality trees. Secondly, these approaches can exhibit edge cases, where performance diminishes to the point where full rebuilds actually give a faster time to image [Lauterbach et al. 2006; Kopta et al. 2012]. Furthermore, our system is already competitive with these schemes. For example, we build the Cloth scene in 3ms whereas recent rotation methods spend around 2.98ms in updating this scene [Kopta et al. 2012]. Finally,

Table 1: *Absolute build times in milliseconds and bandwidth usage for our BVH builder compared to software implementations. A - indicates that the scene was not tested in that work.*

	Intel MIC 1000MHz [Wald 2012]	GTX 480 1400MHz [Sopin et al. 2011]	GTX 480 1400MHz [Garanzha et al. 2011]	Hardware BVH 500MHz (ours)	Hardware BVH BW Usage
Toasters (11k)	9ms	13ms	-	1ms	2MB
Cloth (92k)	19ms	19ms	-	3ms	25MB
Conference (282k)	41ms	98ms	6.2ms	11ms	120MB
Dragon (871k)	-	-	8.1ms	30ms	380MB

Table 2: *Comparison of the SAH costs produced by our builder with those of competing implementations. Unfortunately, Sopin et al. did not provide tree quality measurements in their work, but their tree costs would probably compare quite closely to ours, as they use a very similar approach. Tree costs for HLBVH are taken from both the original HLBVH by Pantaleoni and Luebke and also Garanzha et al. to allow for more data points for comparison. Wald gives cost ratios compared to a binned builder with a large number of bins, whereas we compare to a full sweep builder. Although running simulations was extremely time-consuming for us, we were able to use our CPU builder which gives identical output to our hardware to get extra quality results.*

	[Wald 2012]	[Pantaleoni and Luebke 2010]	(ours)
Toasters	-	-	99%
Cloth	-	-	101%
Conference	101%	117%	114%
Exp. Dragon	103%	-	105%
Armadillo	-	109%	101%
Dragon	-	112%	101%

Table 3: *Total number of floating-point cores and register space.*

	Part. Unit	Bin. Unit	SAH Calc.	**Total**
# Used	4	4	16	
FP ADD	1	3	9	160
FP SUB	3	9	9	192
FP MUL	2	6	12	224
FP INV	1	3	0	16
FP CMP	0	0	144	2304
REG	80KB	4KB	9KB	480KB

there are applications (e.g. photon mapping) where refitting may not be appropriate.

4.6.2 Comparison with HLBVH

The HLBVH method is probably the fastest software method known for building BVHs. However, like refitting, it results in lower quality trees (with SAH costs of around 110% - 115%). As already stated, it is possible for us to construct a hierarchy in many cases in fewer clock cycles than a GPU implementation of HLBVH, despite all of our hardware resource disadvantages and using a much more expensive algorithm. Interestingly, the HLBVH actually performs a binned SAH similar to our own for the upper levels of the hierarchy, consuming as much as 26% of the build time [Garanzha et al. 2011]. One could envision our builder as part of a hardware or hybrid hardware/software solution to HLBVH also. Our work would be an ideal starting point for further research on the hardware implementation of HLBVH or other algorithms. We see our microarchitecture as a fixed-function module that could be integrated into any heterogeneous computing platform, especially a ray-tracing GPU. Our design could represent a full BVH construction subsystem in itself, or be part of a larger subsystem that is capable of building different types of data-structure.

Table 4: *Total area estimation of our subtree builder in mm^2.*

Unit Type	Area (mm^2)	# Used	Total Area (mm^2)
FP ADD	0.003	160	0.48
FP SUB	0.003	192	0.58
FP MUL	0.01	224	2.24
FP INV	0.11	16	1.76
FP CMP	0.00072	2304	1.66
REG 4K	0.019	120	2.28
Prim Buffer	0.94	8	7.52
Ctrl etc.	2.35	-	2.35
Wiring	13.02	-	13.02
Total			31.88

4.6.3 Power Consumption

An important consideration for any microarchitecture is power consumption, and indeed power is likely to dominate architecture designs in the near future. Although at the time of writing we were unable to perform a detailed power analysis of our architecture due to software availability limitations, we can identify several characteristics of our design which are likely to make it much more power-efficient than a GPU or multicore approach.

To perform a more one-to-one comparison of power efficiency, we scaled down the design presented in Section 4 such that its performance would approximately match the two full binned SAH implementations in Table 1. This resulted in an instantiation of only one RAM pair and one subtree builder, operating at the slower speed of 250MHz. The subtree builder in this instance used the same parameters as the design in Section 4 (number of units, threads etc).

The first such characteristic is clock frequency. Power consumption is linearly dependent on clock frequency. A value of 250MHz is only one quarter the speed of an Intel MIC and around one fifth the speed of the shader cores of the GTX480.

The second characteristic of the design is its estimated circuit size as shown in Table 4. The GTX 480 utilises a 529 mm^2 chip size and publications indicate that the vast majority of this space is spent on shader cores and the cache [Wittenbrink et al. 2011] (i.e. the resources utilised in software implementations of BVH construction). Our proposed downsized implementation would not be much larger than the value of $31.88mm^2$ shown in Table 4, making it around 10-15$\times$ smaller. On top of this, the GTX 480 uses a smaller feature size (40nm) [NVIDIA 2010], whereas our estimates are based on 65nm libraries, so the actual difference should be even larger. The significance of this is that much fewer transistors would be needed to implement our design, consequently consuming less power.

One possible confounding of this observation may be a difference

in the level of switching activity between a GPU and our hardware, and a resulting difference in dynamic power per circuit element. To investigate this, we utilised data from our RTL simulations to calculate the average *activity* of each class of FP core and the primitive buffer read and write ports in our design. The activity refers to the proportion of clock cycles in which a unit actually produces a result. For example, one result every two cycles would result in an activity of 50%. In each case, our switching activity was within 20%, a typical value for many circuits. We would therefore not expect the design to exhibit unusually high dynamic power.

Finally, and perhaps one of the most significant observations is to do with data access. It is known among chip designers that off-chip data access to DRAM is around two orders of magnitude more expensive than accessing a local buffer in terms of power consumption, and even accessing a cache across the chip can be well over one order of magnitude more expensive [Dally 2011]. In addition, the power consumption of off-chip memory accesses is known to be more than an order of magnitude more expensive than floating-point operations [Dally 2009]. Moving data on and off the chip thus becomes a substantial portion of the total power consumption. Table 1 and Section 4.2 show that our design generates about half the number of data accesses to external memory than competing software approaches for the same scene, and this could be reduced further by increasing the size of the primitive buffers. In addition to this, all of our internal accesses are highly local to the primitive buffers, indicating high power efficiency once again.

Taking all of these observations into account, we believe it is likely that our design offers a much more power-efficient alternative to software algorithms running on manycore processors. The prediction of many in the computer architecture [Esmaeilzadeh et al. 2011; Dally 2011] and graphics communities [Johnsson et al. 2012] is that scaling of future processor designs will be limited by power consumption. We argue, as other authors have argued [Chung et al. 2010; Venkatesh et al. 2010], that judicious use of fixed-function may form part of a solution to this problem. Based on our results and observations, we propose that our design would be a strong contender for this purpose, especially as acceleration data-structure construction is useful in a broad range of applications, including other rendering algorithms and collision detection.

5 Conclusion

In this paper, we presented one solution to a custom microarchitecture for the construction of binned SAH BVHs for ray-tracing. We propose that this logic could be included as an efficient alternative to software-based builds in a heterogeneous graphics processor. Our results indicate that this approach offers:

- Acceleration of up to 10× for binned SAH BVHs compared to manycore platforms.
- Low memory bandwidth due to explicit management of on-chip buffers and local register file, and the elimination of instruction fetches. A low memory footprint is also observed.
- Our design consumes minimal hardware resources, representing a large efficiency improvement over software BVH builds which typically engage almost the entire chip.
- Preliminary estimates indicate that power efficiency is likely to compare favourably to software implementations running on manycore processors.

These observations indicate that our microarchitecture is overall more efficient at this task than a software approach. Ultimately we would like to see our design integrated into a programmable graphics processor, and we believe we can offer these advantages to such a system. The field of custom hardware for ray-tracing is in its infancy and we believe it is important to explore alternative hypotheses for how each task could be performed.

The primary disadvantage of our design is that it is fixed-function. However, our design is quite configurable and many parameters of the build can be changed. Furthermore, the construction process for other spatial index structures such as kd-trees is highly similar, and only relatively small modifications would be required to add this functionality to our hardware. Additionally, as previous authors have argued, we propose that our design should be coupled with programmable cores, yielding even greater flexibility. Our design could then be used in any technique for which a binned SAH forms part of the process (e.g. HLBVH, selective restructuring, software-controlled construction order for out-of-core methods). Thirdly, although we focus on ray-tracing in this paper, our builder is also immediately applicable to other applications, such as collision detection and even photon mapping by utilising the BVH-based techniques of [Fabianowski and Dingliana 2009].

We also consume hardware real estate. However, we have demonstrated that this is comparable to previous traversal hardware and we argue that this would make sense especially in the context of a ray-tracing processor as it is a core algorithm in this application.

For our future work, we are actively pursuing more accurate area and power estimates, as well as a number of circuit optimizations which should further improve performance and efficiency. It would also be interesting to investigate generalization of the design to construct other hierarchies such as kd-trees and hybrid LBVH/SAH BVHs and see to what degree efficiency can be preserved.

Acknowledgements

We would like to thank the reviewers for offering their time and expertise to evaluate our submission.

References

CAUSTIC GRAPHICS, 2012. Caustic Graphics Company Website. https://caustic.com/. [Online; accessed 15-November-2012].

CHUNG, E. S., MILDER, P. A., HOE, J. C., AND MAI, K. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *MICRO-43: Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture*.

DALLY, B. 2009. Power efficient supercomputing (presentation). In *Accelerator-based Computing and Manycore Workshop*.

DALLY, B. 2011. Power, programmability, and granularity: The challenges of exascale computing (keynote presentation). In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*.

DOYLE, M. J., FOWLER, C., AND MANZKE, M. 2012. Hardware accelerated construction of sah-based bounding volume hierarchies for interactive ray tracing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, 209–209.

ERNST, M., 2012. Embree: Photo-realistic ray tracing kernels. http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels. [Online; accessed 29-March-2013].

ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, 365–376.

FABIANOWSKI, B., AND DINGLIANA, J. 2009. Interactive global photon mapping. *Computer Graphics Forum 28*, 4, 1151–1159.

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, 59–64.

HALL, D. 2001. The AR350: Today's ray trace rendering processor. In *Proceedings of the EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware - Hot 3D Session*.

JOHNSSON, B., GANESTAM, P., DOGGETT, M., AND AKENINE-MÖLLER, T. 2012. Power efficiency for software algorithms running on graphics processors. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12, 67–75.

KARRAS, T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *High Performance Graphics*, 33–37.

KIM, H.-Y., KIM, Y.-J., AND KIM, L.-S. 2012. MRTP: Mobile ray tracing processor with reconfigurable stream multiprocessors for high datapath utilization. *Solid-State Circuits, IEEE Journal of 47*, 2 (feb.), 518 –535.

KOPTA, D., IZE, T., SPJUT, J., BRUNVAND, E., DAVIS, A., AND KENSLER, A. 2012. Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, 197–204.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *IEEE Symposium on Interactive Ray Tracing 2006*, 39 –46.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. *Comput. Graph. Forum 28*, 2, 375–384.

LEE, W.-J., LEE, S.-H., NAH, J.-H., KIM, J.-W., SHIN, Y., LEE, J., AND JUNG, S.-Y. 2012. SGRT: a scalable mobile GPU architecture based on ray tracing. In *ACM SIGGRAPH 2012 Posters*, SIGGRAPH '12, 44:1–44:1.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *IEEE/ACM International Symposium on Microarchitecture*, 3–14.

NAH, J.-H., PARK, J.-S., PARK, C., KIM, J.-W., JUNG, Y.-H., PARK, W.-C., AND HAN, T.-D. 2011. T&I engine: traversal and intersection engine for hardware accelerated ray tracing. *ACM Trans. Graph. 30*, 6 (Dec.), 160:1–160:10.

NVIDIA. 2010. NVIDIA GeForce GTX 480/470/465 GPU datasheet. *NVIDIA Datasheet*.

PANTALEONI, J., AND LUEBKE, D. 2010. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 87–95.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July), 66:1–66:13.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of Graphics Hardware*, 95–106.

SOPIN, D., BOGOLEPOV, D., AND ULYANOV, D. 2011. Real-time SAH BVH construction for ray tracing dynamic scenes. In *Proceedings of the 21th International Conference on Computer Graphics and Vision (GraphiCon), 2011*.

SPJUT, J., KENSLER, A., KOPTA, D., AND BRUNVAND, E. 2009. TRaX: a multicore hardware architecture for real-time ray tracing. *Trans. Comp.-Aided Des. Integ. Cir. Sys. 28*, 12 (Dec.), 1802–1815.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, 7–13.

VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., AND TAYLOR, M. B. 2010. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, 205–218.

WALD, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, 33–40.

WALD, I. 2012. Fast construction of SAH BVHs on the Intel Many Integrated Core (MIC) architecture. *Visualization and Computer Graphics, IEEE Transactions on 18*, 1 (jan.), 47 –57.

WITTENBRINK, C., KILGARIFF, E., AND PRABHU, A. 2011. Fermi GF100 GPU architecture. *IEEE Micro 31*, 5059.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3 (July), 434–444.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 67–77.