MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing

TIMO VIITANEN, MATIAS KOSKELA, PEKKA JÄÄSKELÄINEN, HEIKKI KULTALA, and JARMO TAKALA,
Tampere University of Technology, Finland

Ray tracing is a computationally intensive rendering technique traditionally used in offline high-quality rendering. Powerful hardware accelerators have been recently developed which put real-time ray tracing even in the reach of mobile devices. However, rendering animated scenes remains difficult, as updating the acceleration trees for each frame is a memory-intensive process. This article proposes MergeTree, the first hardware architecture for *Hierarchical Linear Bounding Volume Hierarchy* (HLBVH) construction, designed to minimize memory traffic. For evaluation, the hardware constructor is synthesized on a 28nm CMOS process technology. Compared to a state-of-the-art binned SAH builder, the present work speeds up construction by a factor of 5, reduces build energy by a factor of 3.2, and memory traffic by a factor of 3. A software HLBVH builder on GPU requires 3.3 times more memory traffic. In order to take tree quality into account, a rendering accelerator is modeled alongside the builder. Given the use of a toplevel build to improve tree quality, the proposed builder reduces system energy per frame by an average 41% with primary rays and 13% with diffuse rays. In large (> 500K triangles) scenes, the difference is more pronounced, 62% and 35%, respectively.

CCS Concepts: • **Computing methodologies → Ray tracing**; **Graphics processors**;

Additional Key Words and Phrases: ray tracing, ray tracing hardware, bounding volume hierarchy, BVH, HLBVH

ACM Reference format:
Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. . MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing. *ACM Trans. Graph.* , , Article ( ), 14 pages.
https://doi.org/0000001.0000001_2

1 INTRODUCTION

Ray tracing is a rendering technique where effects such as shadows, reflection and global illumination are more natural to express than in rasterization. Mainstream use of ray tracing has been restricted to offline rendering, but recent years have seen a concerted effort by the academia and the industry to enable real-time ray tracing of dynamic scenes. One prong of this effort has been the development of dedicated ray tracing hardware architectures, both based on programmable processors [47, 48, 53], fixed-function hardware pipelines [32, 41], reconfigurable pipelines [28] and building on conventional, rasterization-based GPUs [25]. Several works focus on mobile systems, reasoning that the ray tracing approach scales well to drawing complex scenes on small displays [48], or aiming to create mobile augmented-reality experiences with physically based lighting [32]. Recently, a commercial mobile GPU IP with ray tracing acceleration has been announced alongside a programming API [44].

Recent ray-tracing hardware accelerators are able enable real-time ray tracing, so far restricted to high-end desktop GPUs, on mobile devices. However, they have so far been largely restricted to scenes with little or no animated content. The reason is that fast rendering algorithms require the scene to be organized in an acceleration data structure such as a *Bounding Volume Hierarchy* (BVH) tree. When displaying a dynamic scene, the data structure needs to be updated or rebuilt on each frame as the scene changes, posing an additional computational challenge. Given enough animated geometry, construction effort overtakes rendering, as ray tracing scales logarithmically with the number of scene primitives, while construction algorithms have $O(n)$ [30] or $O(n \log n)$ [51] complexity. On the desktop, tree construction has been the subject of intensive research, and fast GPU tree builders now exist which organize large scenes at real-time rates. The fastest builders are based on the *Linear Bounding Volume Hierarchy* (LBVH) algorithm by Lauterbach et al. [30], and the improved *Hierarchical LBVH* (HLBVH) by Pantaleoni and Luebke [42], and are able to organize scenes with millions of triangles in real time. These builders leverage the massive amount of computing resources and memory bandwidth available on desktop GPUs, and are, therefore, not directly applicable on mobile systems with limited resources.

A particular restriction of mobile devices is their limited memory bandwidth: a high-end mobile *System-on-Chip* (SoC) has an order of magnitude less memory bandwidth than a high-end desktop GPU. CMOS logic scaling has allowed increasingly complex on-chip computation to fit in the tight power budgets of mobile SoCs, but the energy cost of off-chip communication has scaled down at a slower pace, and is now very expensive compared to computation. For example, reading the operands of a double-precision multiply-add from external memory and writing back the result costs ca. 200 times more energy than the arithmetic itself [24]. Hence, the design of mobile hardware is an exercise of minimizing memory accesses to work around the memory bottleneck. Mobile GPUs incorporate a slew of special architectural techniques to this end, such as tile-based rendering, texture compression [5] and frame buffer compression [46]. A similar body of memory-conserving techiques is emerging for ray tracing accelerators, including treelet scheduling [3], streaming data models [28] and quantized trees [25]. Tree construction for ray tracing is even more memory-intensive than rendering, as the fast sorting-based build algorithms iterate over datasets of hundreds of megabytes, and perform little computation for each element.

email: timo.2.viitanen@tut.fi.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
0730-0301//-ART $15.00
https://doi.org/0000001.0000001_2

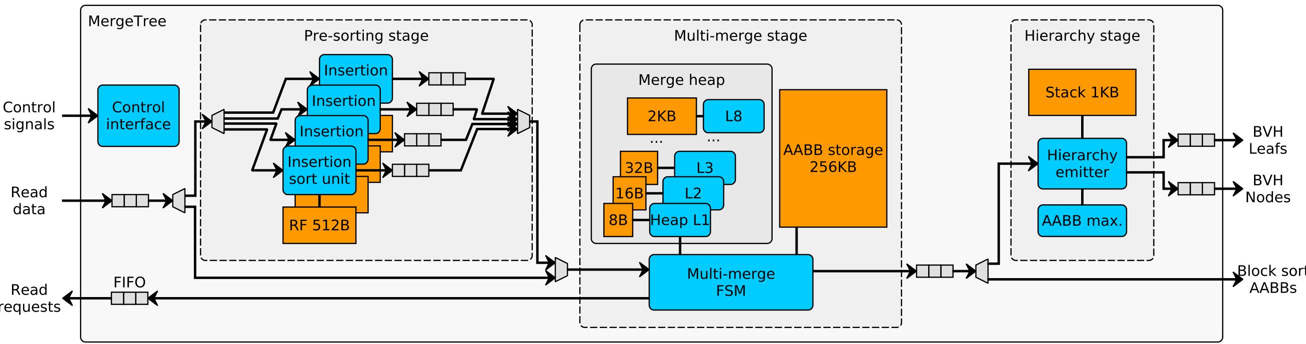


Fig. 1. The proposed hardware architecture, configured for single-pass tree building for scenes of up to 2M triangles. RF: Register File. BVH: Bounding Volume Hierarchy. AABB: Axis Aligned Bounding Box. FSM: Finite State Machine. FIFO: First In First Out buffer.

This article focuses on mobile hardware acceleration of the HLBVH algorithm [42]. HLBVH is interesting as a powerful builder in its own right, and as a component in virtually all build algorithms that aim for a fast build time [12, 18, 23]. HLBVH is also an interesting target for hardware acceleration since it does not make heavy use of floating-point arithmetic, hence, a hardware builder could fit in a small silicon footprint. We investigate whether the high build performance of HLBVH on GPU can translate into energy-efficient operation in a mobile context.

The main contributions of this article are as follows. We propose the first hardware HLBVH builder architecture, named MergeTree. MergeTree incorporates novel architectural techniques to reduce memory bandwidth usage: a hardware-accelerated *external sort* and a novel streaming algorithm for joint hierarchy emission and AABB computation, which operates directly on the sort outputs. The proposed architecture is evaluated via logic synthesis and power analysis on a 28nm *Fully Depleted Silicon on Insulator* (FDSOI) process technology, and by means of a system-level model which includes traversal and intersection hardware. In addition, two toplevel builds are evaluated as inexpensive postprocessing steps to improve tree quality. Early simulation results for MergeTree were reported in a conference brief [49]

Compared to previous work which uses the more expensive binned SAH algorithm [13], MergeTree gives large improvements in build performance, energy efficiency, memory traffic and silicon area, at the cost of reduced tree quality. Toplevel builds are able to recover much of the quality at low cost. System-level modeling shows that with large animated scenes, the energy cost of tree construction becomes comparable to the cost of rendering, and takes up a significant fraction of a mobile power budget. Hence, the build energy savings from MergeTree translate into significant system energy savings despite the slightly lower tree quality. We also observe that most of the energy footprint in hardware-accelerated tree construction is due to DRAM traffic. A direct translation of GPU HLBVH algorithms to hardware, without the proposed memory traffic optimizations, would have energy consumption and runtime similar to [13].

This article is organized as follows. Section 2 discusses background on BVH construction algorithms. Section 3 reviews related work on hardware tree builders and sorting units. Section 4 discusses the basic algorithmic approach in this work and tradeoffs, while Section 5 describes the hardware architecture implementing the chosen algorithm. In Section 6, the architecture is evaluated by means of ASIC synthesis and system-level simulations, and a detailed power analysis is presented. Section 7 discusses limitations of the proposed architecture and future work, and Section 8 concludes the article.

2 PRELIMINARIES

In a BVH, each node subdivides primitives into two disjoint sets, whose *Axis-Aligned Bounding Boxes* (AABB) are stored. If a traced ray does not intersect an AABB, all primitives underneath can be discarded, greatly speeding up the rendering process. A standard way to evaluate the quality of a BVH tree is its *Surface Area Heuristic* (SAH) cost, introduced by Goldsmith and Salmon [19]. The SAH cost of a data structure is the expected cost to traverse a random non-terminating ray through the scene.

A gold-standard way to construct BVH trees is the SAH sweep [36], a greedy top-down partitioning algorithm which at each step evaluates all possible axis-aligned splits of the primitives, according to their AABB centroids, into two subset. The algorithm then selects the split with the lowest SAH and repeats recursively for each subset. Since the basic SAH sweep has a long runtime, often the binned variation [51] is used instead, which evaluates only, e.g., 8 or 16 possible splits per axis.

Starting with *Linear BVH* (LBVH) by Lauterbach et al. [30], a family of GPU construction algorithms has been proposed which are orders of magnitude faster than SAH-based builders. In LBVH, the scene primitives are first sorted according to the Morton codes of their AABB centroids, and then in the process of *hierarchy emission*, a BVH hierarchy is built which has a binary radix tree topology with regards to the sorted Morton codes. Finally, the AABBs of each node are computed in a bottom-up order.

Pantaleoni and Luebke [42] propose *Hierarchical Linear BVH* (HLBVH) with improved build performance and a more compact

memory layout compared to LBVH. They further suggest improving tree quality by rebuilding upper levels of the tree with a binned SAH sweep, in an approach called HLBVH+SAH. We use this terminology in the present work, though several works use HLBVH to denote HLBVH+SAH. Garanzha et al. [17] and Karras [22] have further optimized the GPU software implementation of HLBVH, especially the hierarchy emission which is non-trivial to parallelize. Most recently, Apetrei [6] combined the hierarchy emission and AABB calculation stages into a single step for a further speedup.

More complex algorithms have been built around HLBVH which further improve tree quality. Karras and Aila [23] divide a HLBVH tree into treelets and rearrange nodes within each treelet in parallel to achieve higher tree quality, in an approach later dubbed *Treelet Restructuring BVH* (TRBVH). In the *Agglomerative TRBVH* (ATRBVH) of Domingues and Pedrini [12], the exhaustive search of treelet permutations in TRBVH is replaced with an agglomerative build, yielding nearly the same tree quality at a fraction of the build time. Garanzha et al. [18] sort primitives according to their Morton codes, but instead of the HLBVH hierarchy emission, they store primitive counts in a multi-level grid and perform an approximate SAH sweep. Both TRBVH and Garangzha et al. [18] also break large triangles with spatial splits to improve tree quality. Recently, Ganestam and Doggett [16] proposed a fast, high-quality preprocessing step for triangle splitting.

3 RELATED WORK

In this section, we introduce related work on hardware architectures for tree construction and sorting.

3.1 Tree Build and Update Hardware

Some hardware builders have been proposed for k-d trees, an alternative acceleration data structure to BVH. The RayCore architecture [39] incorporates a hardware k-d tree builder unit which builds high levels of the hierarchy with a binned SAH sweep, and switches to a sorting-based build when the dataset is small enough to fit in on-chip SRAM. The builder is suitable only for small models, e.g., a 64K triangle scene already takes 0.1 seconds to build. Therefore, they also design their renderer to use two acceleration trees: the smaller tree contains all animated geometry and is rebuilt with the hardware unit. The FastTree unit [34] uses Morton codes for k-d tree construction, and is the fastest k-d tree constructor hardware so far, at ca. 4 times the performance of the RayCore builder. However, tree quality is not evaluated. They use a memory-intensive radix sort, but this does not appear to harm performance, since k-d trees are much more compute-intensive to construct than BVHs, so their memory interface is not stressed.

In the literature, BVH trees are shown to be less expensive to build and update than k-d trees. Doyle et al. [13] propose the first hardware architecture for BVH construction, which implements the recursive binned SAH sweep algorithm. The architecture was recently prototyped on FPGA [14]. The HART rendering system [40] updates the BVH tree with hardware-accelerated *refit* operation instead of running a full rebuild on each frame. Since tree quality degrades with each refit, asynchronous rebuilds are run on the CPU to refresh the tree.

The present work is the first hardware implementation of the HLBVH algorithm, which is the basis for most high-performance GPU builders. In contrast to GPU implementations of HLBVH [6, 17, 22, 42], we adapt the algorithm for a streaming, hardware-oriented implementation with minimal memory traffic. The produced trees remain identical to the original work [42]. Our main point of comparison is the state-of-the-art builder by Doyle et al. [13], which implements binned SAH, a more computationally expensive algorithm. Compared to a refit accelerator [40], the proposed builder is able to handle animations which affect mesh topology, e.g., fluids rendered with Marching Cubes, and can handle animation frames with entirely new geometry.

3.2 Sorting Hardware

The multi-merge sort approach has recently been used to sort large bodies of data with FPGAs to accelerate database operations. Koch and Torrensen [27] implement their multi-merge logic with a tree of comparators, and merge data from up to 102 input buffers. As a main difficulty in comparator tree design, they identify the problem of propagating back-pressure through the tree in a single cycle, and solve the issue by inserting decoupling FIFOs that split the tree into smaller sub-trees. Moreover, they pipeline the compare operations to get a higher operating frequency on FPGA. Casper et al, [9] further increase throughput by augmenting the top of the tree with comparators that produce multiple sorted values per cycle. They demonstrate merges from up to 8K input buffers per cycle, but in this case require over 2 MB of buffer memories. The proposed accelerator implements a novel multi-merge based on a pipelined hardware heap rather than a comparator tree, giving a compact silicon area footprint at the cost of reduced throughput.

4 ALGORITHM DESIGN

In this section we describe how we adapt Pantaleoni and Luebke's [42] HLBVH algorithm to reduce memory traffic.

4.1 Data Structure Design

There are several variations of BVHs in the literature, and the chosen variant can have implications on build performance, so we describe the layout used in this work in detail here. In this layout, the node data structure stores the axis aligned bounding boxes of its two children and pointers to them. Áfra et al. [1] describe this arrangement as MBVH2. A leaf size field determines whether the child is another node or a leaf. Leafs are contiguous sub-arrays in a leaf table, bounded by the index and leaf size fields. Each entry in the table is a pointer to primitive data, which is used for ray-primitive tests and shading. The complete node data structure is 64 bytes long as shown in Fig. 2.

There are many variations on the above details in the literature. Sometimes a node only has its own AABB and two child pointers, but the two-AABB structure is superior in hardware ray tracing [31]. More importantly, many high-performance ray tracers, e.g.. Aila and Laine [4], store primitive data in the leaf table, foregoing the extra indirection per rendering. Often the primitives are also preprocessed for faster intersection testing with, e.g., Shevtsov's [45] method.

```
struct AABB {
        float lb_x, lb_y, lb_z;
        float ub_x, ub_y, ub_z;
        // 0 if child is an inner node.
        int leaf_size;
        // Index of inner node or leaf.
        int child_idx;
};

struct BVHNode {
        AABB aabbs[2];
};

BVHNode nodes[N];
int leafs[M];
```

Fig. 2. BVH data structure.

Table 1. Sorting algorithm comparison for LBVH construction. External memory traffic in bytes.

	Prim. read	Sort	BVH write	Total
Sort Morton codes				
Radix-16 counting sort	32	192	100	324
Multimergesort, 1 pass	32	16	100	148
Multimergesort, 2 passes	32	32	100	164
Sort AABBs				
Radix-16 counting sort	32	768	68	868
Multimergesort, 1 pass	32	64	68	164
Multimergesort, 2 passes	32	128	68	228

Since primitives in the same leaf may not have been contiguous in the input data, this implies rearranging the primitives.

The present work opts to use a reference table for two main reasons. First, we try to make our results compatible with the main prior work by Doyle [13], which to our best understanding has this structure. Secondly, by taking primitive AABBs as input, the resulting architecture is generic to any primitive type for which an AABB can be computed - some examples of useful primitives are the pyramidal displacement mapped surfaces in [39] and indexed-vertex triangle lists in [25]. A primitive-leaf table could be produced as a post-processing step.

4.2 Sorting

Sorting accounts for much of the memory traffic in HLBVH, so we optimized it by referring to literature on *external sorting* data on slow magnetic disc drives. One optimal sorting algorithm in this environment is the *multimergesort* [2]. Given N data elements that reside in slow external memory, a fast local memory of size M, and a preferred read length of B, the multimergesort first performs partial sorts for N/M blocks of size M. After this, the algorithm runs multi-merge passes which merge M/B sorted blocks into a larger block. Table 1 compares the minimum memory accesses of multimergesorting in the context of tree construction to a typical radix-16 sort. Assuming one primitive per leaf, a BVH organizing N primitives has $N-1$ nodes. Any builder, then, has unavoidable memory traffic from primitive input (32B per input AABB) and hierarchy emission (64B per input AABB for nodes and 4B for the leaf table). In addition, primitive sorting requires memory accesses.

GPU implementations often use, e.g. a radix-16 parallel prefix sort, which performs eight passes through the data, each pass reordering the data according to four bits of the sort key. In each pass the entire data array is read twice and written once. Assuming the sort operates on 8B Morton code - primitive reference pairs, it then requires $3 \times 8 \times 8\text{B} = 192\text{B}$ traffic per inpt AABB. Finally, the joint hierarchy emission and AABB computation stage must fetch the primitive AABBs referenced by the sort results, adding 32B to the unavoidable 68B output traffic. Out of the total traffic of $324B$, more than half is produced by the sort. Replacing the radix sort with multimergesort and assuming a small enough scene to sort in a single pass (2M triangles in the proposed design), the sort traffic drops to a negligible $16B$, with an additional $16B$ per pass. Assuming the AABBs from primitive input may be streamed on-chip to the block sort stage, and the results of the multi-merge to the hierarchy emission stage, the sorted pairs are only accessed twice, for $16B$ traffic. The total traffic of $148B$ is less than half that of the radix sort case.

The reads and writes in Table 1 are otherwise consequent, except when the hierarchy emission stage loads the primitive AABBs referenced by each sort output, it generates inefficient 32B random accesses. Consequently, it is interesting to directly sort the primitive AABBs instead of references. Direct AABB sorting is clearly inefficient with a radix sort due to the quadrupled sorting traffic. With multimergesort, the extra traffic is smaller (48B), and nearly offset by the removed hierarchy emission loads (32B). Total traffic still increases by ca. 11%, but all memory accesses can now be arranged in long consecutive bursts which are more efficient. If two or more multi-merge passes are needed, the advantage of AABB sorting is less clear. It is, then, desirable to use AABBs as sorting elements and support as wide a merge as practical, so that scenes of interest fit in a single pass. We use the AABB multi-merge approach as the basis for the present work. It should be mentioned that, in the extreme, the primitives themselves could be used as sort elements. It is inexpensive to add hardware to recompute their AABBs and Morton codes on demand, so the main cost would be increased memory traffic and on-chip storage. This approach is efficient for generating a leaf array with primitive data, discussed in the previous section, but foregoes genericity of the architecture for different primitives.

5 HARDWARE ARCHITECTURE

This section describes a hardware architecture named MergeTree which implements the designed construction algorithm. A block diagram of the architecture is shown in Fig. 1. The architecture can be divided into the *pre-sorting* and *multi-merge* stages which sort an array of input AABBs according to their Morton codes, and a *hierarchy stage* which processes the sorted AABBs to emit a BVH tree. It has two operating modes where different sub-modules are active, as shown in Fig. 3. The partial sort mode is used to generate $\frac{N}{M}$ AABB arrays small enough to fit in the on-chip *AABB storage*, while the hierarchy stage is inactive. In the multi-merge mode, the arrays are merged into a final sorted sequence and fed into the

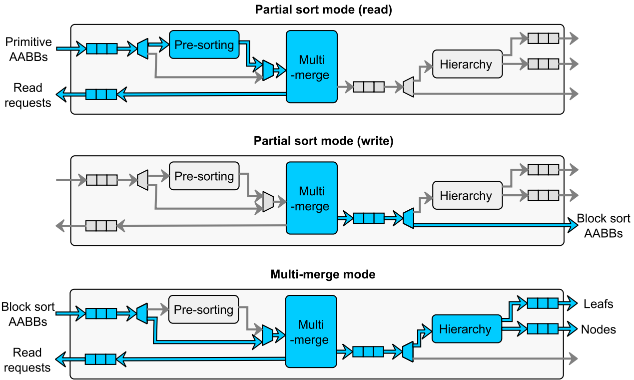


Fig. 3. Operating modes of the proposed architecture.

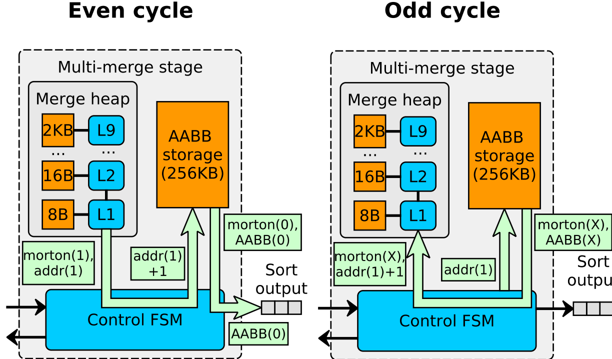


Fig. 4. Steady-state behavior of the multi-merge stage. morton(i): Morton code, addr(i): scratchpad address and AABB(i) AABB of the ith sorted element. After the heap outputs a sorted element (1), the control FSM first fetches its successor (X) from the scratchpad and performs a heap *replace operation*. Next, it fetches the original element (1) for output. The unit alternates between the two states shown: on even cycles a scratchpad read is initialized with an address read from the heap unit, on odd cycles data is fed from the scratchpad to the heap, and on the next even cycle, to the output FIFO.

hierarchy stage. The multi-merge mode forms the backbone of the present architecture: it is a serial process that requires a specialized hardware pipeline for good performance. In contrast, the partial sorts parallelize well, and could be done with the multi-core CPU or mobile GPU in a SoC. The partial sorts are only integrated into the MergeTree so as to reuse hardware from the multi-merge mode. The following subsections first describe the multi-merge mode resources, and then the partial sorting scheme.

5.1 Primitive Input

The main data format used for input and internal storage is an AABB with three lower-bound and three upper-bound coordinates, a memory reference to primitive data, and a Morton code, for a total of 256 bits.

5.2 Heap Unit

The main component of the proposed algorithm which requires hardware acceleration is the multi-merge from many input sequences into a single, sorted output stream. Hardware acceleration is necessary since a sequential heap implementation for a heap of capacity n takes up to $\log n$ compare-swaps to insert an element, which is too slow for our use. Since comparator trees, used by recent sorting accelerators [9, 27], appear unreasonably large when scaled to wide inputs, we use an alternate approach of implementing a pipelined heap data structure in hardware. In software implementations the heap is stored in a single array in memory, where the children of an element can be found with simple arithmetic. However, in custom-designed hardware, each level of the heap can be implemented as a separate memory module, and heap operations would then operate in a pipelined manner, such that a new heap operation can be started while previous operations are still propagating toward deeper levels. This hardware structure was proposed by Bhagwan and Lin [8] for the implementation of large priority queues in telecommunications processors. Ioannou and Katevenis [20] optimize the design for clock speed by overlapping stages of computation. In this work we largely follow the design of the latter work, and we refer the reader to their article [20] for details. We support the *insert* and *replace* operations, and implement *remove* as a *replace* with a large special-case value ∞. Replace operations have a maximum throughput of one value per two clock cycles, since fully pipelined operation would need an expensive set of global bypass wires between heap levels.

Detailed comparison to comparator trees is outside the scope of this article, but it should be noted that trees have $O(n \log n)$ comparators with regards to the merge width, and the present design has $O(\log n)$. The heap is, consequently, easier to scale to the wide merges desired in this work. Our throughput is lower than an optimized comparator tree, but since we have a much higher clock frequency available on ASIC than FPGA, and are sorting larger data elements, this is less of an issue than in the FPGA works. If more throughput is desired, it may be interesting to use a hybrid scheme similar to [9], combining a small toplevel comparator tree with subtrees implemented as heaps.

5.3 Multi-Merge Unit

The pipelined heap is connected to the rest of the system by a hardware finite state machine which initializes the heap, requests replacement data from the memory and feeds it into the heap, and emits output data. To limit the size of the heap, the full primitive AABBs are stored in a SRAM scratchpad memory, and the heap only contains Morton codes and DRAM addresses of the corresponding primitives. The scratchpad memory is organized into a set of double-buffered queues for each block being multi-merged: when one buffer has been processed, a memory read is queued to replace it, but processing can continue from the other side of the double buffer. Only if the second half of the double buffer is also consumed before replacement input arrives for the first half, does the multi-merge unit need to stall. With double buffering, the multi-merge unit provides a degree of memory latency hiding, as the merge process is likely to visit multiple other buffers between the B elements of a single buffer. This property depends on the heap accesses being

well distributed between the blocks: if, e.g., the input data is already sorted, performance may suffer.

At the outset of partial sort and multi-merge modes, the FSM makes memory requests to fill the scratchpad, and inserts the initial elements of each buffer into the empty heap. In steady-state operation, on each even cycle, the finite state machine reads the top element of the heap, bit manipulates its DRAM address to find the scratchpad location of the following data element in the same block, and begins a SRAM read at that location. On the following cycle the SRAM data is available and used for a *replace* heap operation. On odd cycles, the AABB corresponding to the previously read heap-top value is read from scratchpad, and is written to the output FIFO on the next even cycle, simultaneously with the next heap-top read, in a pipelined manner. Like the heap unit, this design is also half-pipelined, producing one sorted AABB per two cycles in the absence of stalls.

Fig. 4 demonstrates this steady-state behavior. On the example even cycle, the scratchpad address addr(1) of a sorted element is read from the heap, incremented, and used to initialize a scratchpad read of the element's successor in the same buffer. On the following odd cycle the successor's Morton code morton(X) is available from the scratchpad, and is used to replace the top element of the heap. Simultaneously, a scratchpad read is initialized with the original (unincremented) address. On the next even cycle, the sorted element's AABB would be read from the scratchpad and written to the output, as is done for the previous sorted element (0) in Fig. 4.

A separate register array tracks which buffers in the scratchpad are valid: if the finite state machine attempts to fetch invalid data, execution instead stalls until the referenced data is available. Locations are invalidated when consumed, and validated again when replaced from memory. When the final element from the given block or scene is read, a *remove* operation is performed instead of a *replace*. To avoid special-case handling for the possible small buffer at the end of the scene, it is padded to full size with special-case AABBs with a higher Morton code than is generated for normal scene geometry: these are, then, sorted to the end of the scene and ignored by subsequent processing.

5.4 Hierarchy Emitter

In order to minimize external memory traffic, we stream sorted AABBs to a hardware finite state machine which implements a serial HLBVH hierarchy emission algorithm. The hardware unit implements Algorithm 1, such that a single stack operation is performed per cycle.

Fig. 5 shows a visual example of the algorithm in operation. The algorithm reads in a sorted stream of AABBs, and computes the highest differing bit between each pair of successive Morton codes, which is interpreted as a hierarchy level for an inner node to be generated. Based on each input's hierarchy level, the unit either concatenates the input into a large leaf (lines 4..8), pushes it into a small hardware stack, or combines it with the top AABB in the stack to generate a node. The latter process is then repeated with the AABB of the generated node used as the input node, until a higher-level element is found in the stack, or the stack is empty. Each node is output when sufficient primitives have been read to determine its child bounding boxes, resulting in a bottom-up, depth-first order. Stack entries represent inner nodes whose right child is unknown: they consist of a left child AABB and a hierarchy level. Optionally, a *highlevel_threshold* parameter can be supported by the hardware in order to emit high-level nodes for a separate toplevel build (lines 15..19), as in the HLBVH+SAH by Pantaleoni and Luebke [42]. The unit then generates separate trees for each highlevel grid cell, and outputs their AABBs to a buffer for postprocessing. When the parameter is greater than the highest Morton code bit, the unit's reverts to conventional HLBVH and outputs a single tree.

ALGORITHM 1: Streaming hierarchy emission algorithm

```
1  while True do
2      input ← nextInput ;
3      read nextInput from FIFO;
4      while input and nextInput have the same Morton codes do
5          input ← nextInput ;
6          read nextInput from FIFO;
7          input ← combine ( input, nextInput );
8      end
9      diff ← highest diff. Morton bit of input and nextInput ;
10     while ¬ stack.empty() ∧ stack.top().diff < diff do
11         BVHNode n( stack.pop().aabb, input ) ;
12         input ← n.aabb ;
13         output.push( n ) ;
14     end
15     if diff > highlevel_threshold then
16         highlevel_output.push( input ) ;
17     else
18         stack.push( input, diff ) ;
19     end
20 end
```

An example of the algorithm is shown in Fig. 5. In Cycles 1 and 2 in Fig. 5, stack entries for nodes on hierarchy levels 2 and 1 are pushed to the stack: the left child of the level-2 node is the primitive a, and child of the level-1 node, b. The next encountered hierarchy level of 3 is higher than the stack-top at level 1, so the stack-top node A can be completed. On Cycle 4, the next node in stack can be completed. On Cycle 5, a stack entry containing the entire subtree constructed so far, is pushed to the stack: it will become the root of the tree. Cycles 6 through 9 finish the right-side subtree in the same manner, except that primitives d and e have the same Morton code, and so are combined to the same leaf. Finally on Cycle 10, the top node is emitted. Cycles 8..10 also show that processing finishes with a special-case input AABB with higher Morton code value than in the rest of the geometry: this causes the remaining stack entries to be popped. The generated inner node topology in this example corresponds to the Morton code bits shown in Fig. 5.

It is visible that generating n inner nodes requires $2n$ stack operations, while enlarging a leaf takes 1 cycle. Since a BVH organizing m leafs has at most $m - 1$ nodes, the worst-case runtime of the emitter is $2m - 2$ cycles for for m inputs, when there is exactly one primitive per leaf. The average throughput of the hierarchy emitter is, then, the same or higher as that of the multi-merge unit, but data is consumed at an uneven rate depending on inputs, so a FIFO buffer

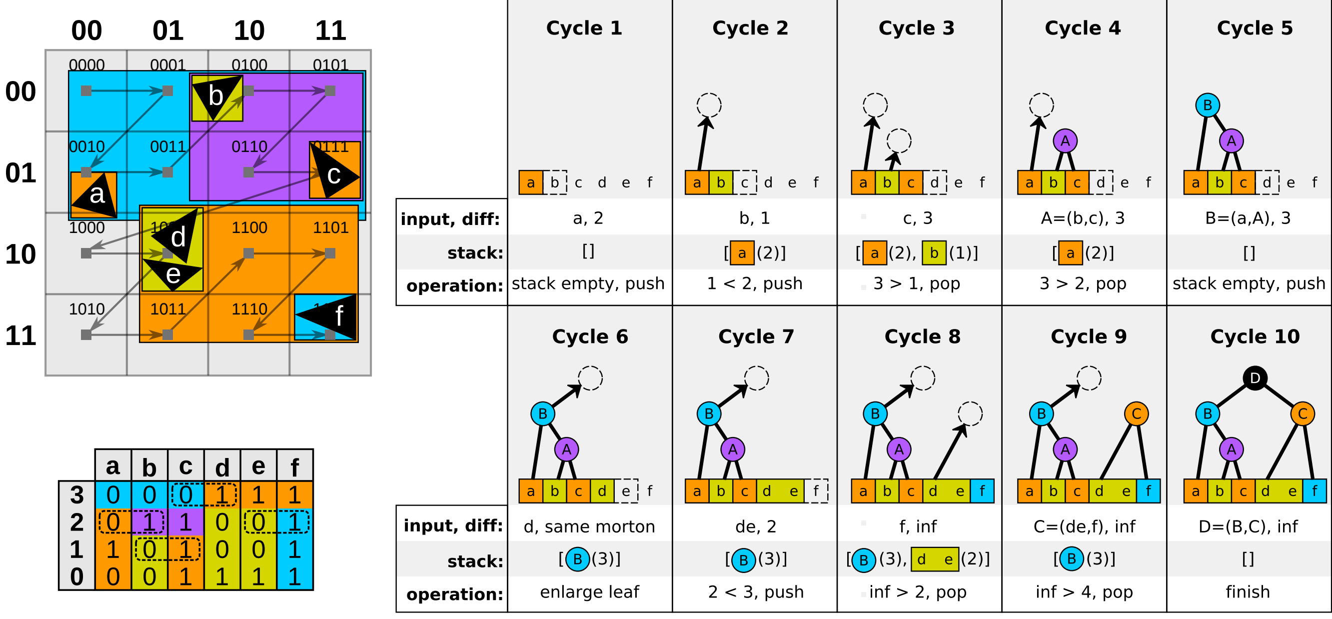


Fig. 5. Example of streaming hierarchy emission, which processes the given sequence of primitives (a-f) and their Morton codes (bottom left) to produce inner nodes (A-D) and leaf table entries for each primitive. Primitives, Morton curve and generated hierarchy are shown in the top left. *diff* is the highest differing bit in the morton codes of *input* and the following primitive, and determines the hierarchy level of the corresponding inner node. If *current* is an inner node, it has the *diff* of the last primitive it contains: e.g. inner node B in Cycle 5 contains primitives a, b, and c, and has the *diff* of c, 3.

is required between the two units. Hierarchy emission finishes by assuming the highest differing bit after the final sorted primitive is ∞, which causes all remaining stack entries to be popped. The maximum required stack depth is the same as the number of Morton code bits, i.e., possible hierarchy levels.

5.5 Partial Sort

The multi-merge stage described above is straightforward to reuse for the scratchpad-sized partial sort, by configuring the merge heap so that each double-buffer in the AABB storage is the final one in its block, and no further buffers are fetched. Then the only additional hardware needed is logic to sort every buffer-sized sub-block prior to merging. This is easy to implement concurrently with data reading, by streaming the read data into a small number of buffer-sized pre-sorters: our implementation consists of a hardware state machine performing an insertion sort at a rate of one compare-and-swap per second, plus one cycle of overhead for each inner loop of the sort. Multiplexers are inserted to bypass the insertion sorters in the multi-merge mode, and the hierarchy emitter in the partial sort mode.

5.6 Toplevel build

Trees produced by HLBVH likely require post-processing for acceptable quality. The earliest idea proposed in this direction is HLBVH+SAH [42], where top levels of the BVH hierarchy, corresponding to high bits of the Morton code, are rearranged with a higher-quality algorithm. This toplevel build concept is of particular interest in a memory-constrained system, as the datasets are small enough to fit on-chip.

In order to evaluate toplevel builds, we add a configuration option to our simulated hardware to emit an array of high-level nodes, which can then be passed to a separate software or hardware builder. Two toplevel builders are evaluated in this work: a binned SAH build using the accelerator of Doyle et al. [13], and a software implementation of ATRBVH [12]. ATRBVH is itself a LBVH postprocessing step, but the usage in this paper is novel in that we apply the algorithm only to the high-level nodes rather than processing the entire tree: it is then sufficiently lightweight to give real-time performance on a mobile CPU.

6 EVALUATION

This section describes first models used for comparison against the state-of-the-art BVH builder [13]. We then evaluate the performance, silicon area and power characteristics of the proposed builder architecture in isolation. Finally, the builders are modeled as components of a larger rendering system.

6.1 Binned SAH builder model

The closest point of comparison for the proposed builder is the binned SAH architecture proposed by Doyle et al. [13]. The SAH builder is also of interest as a top-level builder used to improve the tree quality of HLBVH trees output by the current work. For

Table 2. Memory traffic model validation. Traffic reported for binned SAH builder [13] is compared against traffic predicted by the model.

Scene	Mem. traffic (MB)	
	Builder [13]	Estimated
Toasters	2	1
Fairy	25	25
Conference	120	125
Dragon	380	379

evaluation, it is interesting to examine the builder in more scenes than reported in [13], and to consider its energy consumption.

First, memory traffic is modeled by instrumenting a software binned SAH builder to record build statistics. Traffic is caused by the BVH output, and by *large sweeps* whose datasets are too long to fit in local primitive buffers. The architecture instance in [13] has buffer capacity for 8192 primitive AABBs. However, during each sweep, the hardware simultaneously produces input data for two child sweeps: in some cases both children could individually fit in the buffers, but are together too large. In this case, we assume only the smaller child to be a large sweep. As shown in Tab. 2, this model comes close to replicating the memory traffic results in [13]. A fixed size threshold of 4096 slightly overestimates traffic, while a threshold of 8192 underestimates it. Floating-point operations are counted based on Wald's algorithm [51], and then combined with memory traffic to obtain a lower-bound memory model.

Finally, the runtime of a simplified binned SAH builder is modeled so as to give a lower bound for toplevel build performance. The simplified builder operates serially and has a single partitioning unit. It alternates between partitioning and binning a sweep at a rate of one input AABB per cycle, and SAH computation, which takes 32 cycles at the end of each sweep. The unit simplified in this way is substantially slower than the original, but as toplevel trees have only ca. 500-2000 nodes in our test scenes, toplevel build has neglible effect on runtime.

6.2 Implementation and Power Analysis

In the graphics hardware community, hardware complexity is often estimated by counting arithmetic units and memories in the design, but in this case we are especially interested in the energy of the proposed architecture, and whether it can reach a high clock frequency. Therefore we wrote a prototype RTL description of the proposed architecture and synthesize it on a CMOS technology. All components in Fig. 1 were implemented in SystemVerilog, and SRAM macros were used for the AABB storage.

The tree builder was synthesized on a 28nm FDSOI technology with Synopsys Design Compiler. The parameters of the builder were set at $M = 8192$, $B = 16$, resulting in a unit with a 256KB scratchpad memory, which handles up to 2M triangles in one pass, reads data in 512B increments and performs a 256-way merge. We include eight partial sorters for scalability. To determine the buffer size B, we experimented with the DRAMPower model [10] and found that increasing consecutive access size is clearly beneficial at least up to 512B (16 AABBs). The target frequency is set at 1GHz, supply voltage at 1V, and operating temperature at 25°C. Clock gating and multi-threshold voltage optimizations are enabled.

In order to evaluate performance, we run RTL simulations of the builder unit with various input scenes and memory interfaces. The external memory is modeled with the DRAM simulator Ramulator [26]. For the memory organization, we select 64-bit, 1- and 2-channel LPDDR3-1600 (*slow*, *medium*) which are the closest devices to state-of-the-art mobile device memory for which we are able to perform power analysis. In addition, we simulate a 64-bit, 4-channel LPDDR3-1333 memory (*fast*) which gives a bandwidth close to Doyle [13] to facilitate a direct runtime comparison, but this interface is not representative of mobile systems. We subtract from all memory power figures a static power term computed from an idle memory transaction trace, corresponding to, e.g., refresh power, to isolate the extra dynamic power added by the tree build. Ramulator is integrated to the RTL testbench through the SystemVerilog Direct Programming Interface wrapper, such that memory requests from the simulated builder map into Ramulator transactions, and input data is fed into the builder when the corresponding read transaction completes. RTL simulation was run on 14 test scenes, and the resulting trees were verified in a software ray tracer.

We have also implemented a C++ architectural simulator which gives results within ca. 5% of the RTL simulation at a 20× faster runtime, and allows easier observation of the build process. Fig. 6 shows example simulation traces generated with the C++ simulator. The different execution states are visible: first the unit alternates between partial sort reads which utilize the insertion sorters, and writes which utilize the merge heap. Most of the execution time is spent on the multi-merge phase, which is clearly memory-limited in the slower memory options. In the fast memory option, the throughput of the multi-merge hardware starts to limit performance. Finally, a toplevel SAH build is shown, which is more compute-intensive and uses little memory.

The build times, memory traffic and tree quality of the proposed builder are compared to related work. As a desktop benchmark, we compare against the high-quality ATRBVH builder by Domingues and Pedrini [12] with default settings, and their freely available implementation of Karras' HLBVH algorithm [22], set to use 32-bit Morton codes. The GPU builders are run on a computer with a GeForce GTX 1080 GPU and a Intel Core i7-3930K CPU, counting only kernel execution times. In tree builder hardware, we compare against the state of the art binned SAH builder by Doyle et al. [13] and the k-d tree builder FastTree [34]. We use the performance figures from their article, and generate memory traffic as described in the previous subsection. Memory traffic for GPU HLBVH was extracted with *nvprof*.

Tree quality is evaluated based on the SAH cost of produced trees [19]. The SAH cost C of a BVH can be computed as:

$$C = C_i \sum_{i=0}^{n_{\text{nodes}}} \frac{A(N_i)}{A(R)} + C_l \sum_{i=0}^{n_{\text{leafs}}} \frac{A(L_i)}{A(R)} + C_t \sum_{i=0}^{n_{\text{leafs}}} \frac{P_i A(L_i)}{A(R)},$$

where $A(N_i)$ and $A(L_i)$ are the surface areas of the given inner nodes and leafs, $A(R)$ the surface area of the scene AABB, P_i is the primitive count within a given leaf, C_i is the cost of traversing a node, C_l the cost of traversing a leaf, and C_t the cost of a

Table 3. Build performance and quality comparison. SAH costs are relative to a full SAH sweep. Average build time is normalized to GTX 1080 HLBVH. BW denotes system memory bandwidth. The proposed builder (HW HLBVH) is evaluated with three DRAM configurations. Toplevel builds (HW binned SAH, TK1 ATRBVH) show build time *in addition to HW HLBVH*.

		Toasters (11K)		Bunny (70K)		Cloth (92K)		Fairy (174K)	
	BW (GB/s)	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**
GTX 1080 HLBVH [12]	320	0.27	166%	0.45	103%	0.52	123%	0.82	129%
GTX 1080 ATRBVH [12]	320	1.44	76%	2.67	89%	3.35	92%	6.03	87%
HW binned SAH [13]	44	1	103%	-	104%	3	103%	-	102%
HW k-d tree [34]	12.8	-	-	5.1	-	-	-	10.8	-
HW HLBVH (proposed)	12.8	0.16	160%	1.11	122%	1.50	122%	2.14	118%
—"—	25.6	0.10	"	0.66	"	0.91	"	1.46	"
—"—	42.7	0.08	"	0.49	"	0.66	"	1.08	"
HW Binned SAH topl.	12.8	0.03	112%	0.11	113%	0.05	114%	0.05	104%
TK1 ATRBVH topl.	14.9	4.60	102%	28.92	116%	9.31	113%	6.35	109%

		Crytek (262K)		Conference (283K)		Sportscar (301K)		Italian (375K)	
	Mem. BW	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**
GTX 1080 HLBVH [12]	320	1.10	151%	1.25	150%	1.22	184%	1.45	214%
GTX 1080 ATRBVH [12]	320	8.11	87%	10.01	96%	9.45	89%	11.69	85%
HW binned SAH [13]	44	-	107%	11	103%	-	120%	-	105%
HW k-d tree [34]	12.8	-	-	17.2	-	-	-	-	-
HW HLBVH (proposed)	12.8	3.60	142%	2.83	154%	4.17	142%	4.80	208%
—"—	25.6	2.40	"	2.05	"	2.77	"	3.26	"
—"—	42.7	1.71	"	1.88	"	1.98	"	2.35	"
HW Binned SAH topl.	12.8	0.06	109%	0.02	106%	0.08	121%	0.04	135%
TK1 ATRBVH topl.	14.9	10.05	99%	4.33	89%	15.20	116%	8.62	117%

		Babylonian (500K)		Kitchen (761K)		Dragon (871K)		Buddha (1087K)	
	Mem. BW	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**
GTX 1080 HLBVH [12]	320	1.80	208%	2.49	155%	2.99	136%	3.66	133%
GTX 1080 ATRBVH [12]	320	14.69	91%	19.80	91%	23.22	90%	29.55	84%
HW binned SAH [13]	44	-	104%	-	101%	30.00	102%	-	102%
HW k-d tree [34]	12.8	-	-	-	-	52.5	-	65.5	-
HW HLBVH (proposed)	12.8	6.52	202%	8.61	149%	13.08	135%	15.60	132%
—"—	25.6	4.43	"	6.07	"	8.55	"	10.32	"
—"—	42.7	3.21	"	4.46	"	6.04	"	7.32	"
HW binned SAH topl.	12.8	0.04	147%	0.02	113%	0.08	120%	0.06	120%
TK1 ATRBVH topl.	14.9	7.54	128%	3.98	94%	17.60	123%	12.84	122%

		Livingroom (1459K)		Lion (1604K)		Geom. Mean			
	Mem. BW	**Time (ms)**	**SAH**	**Time (ms)**	**SAH**	**Norm. time**	**SAH**		
GTX 1080 HLBVH [12]	320	4.41	160%	5.07	144%	0.68	153%		
GTX 1080 ATRBVH [12]	320	39.84	82%	42.37	88%	5.14	89%		
HW binned SAH [13]	44	-	101%	-	103%	6.40	104%		
HW k-d tree [34]	12.8	-	-	-	-	9.39	-	-	
HW HLBVH (proposed)	12.8	16.76	144%	24.05	139%	2.02	148%		
—"—	25.6	11.76	"	15.76	"	1.33	"		
—"—	42.7	8.56	"	11.12	"	1.00	"		
HW binned SAH topl.	12.8	0.06	116%	0.28	128%	0.03	118%		
TK1 ATRBVH topl.	14.9	9.69	109%	45.75	119%	4.94	111%		

Table 4. Area and power breakdown of synthesized design. Power results from Lion scene, fast memory model.

	Area (mm^2)	Power (mW)
Insertion sorters	0.24	25.8
Multi-merge unit	1.14	17.2
Hierarchy emitter	0.03	2.2
FIFOs and muxes	0.99	16.0
Total	2.41	61.2

Table 5. Hardware comparison with quadratic process scaling.

Architecture	Area (mm^2)	Node (nm)	Area @28nm (mm^2)	Mem. BW (GB/s)
Geforce GTX 1080	314	16	962	320
HW binned SAH [13]	31.9	65	5.9	44
HW k-d tree [34]	1.4	28	1.4	43
MergeTree	2.4	28	2.4	43

Table 6. Memory traffic comparison (MB)

Scene	Proposed	HW Binned SAH	GPU HLBVH
Toasters	1.7	1.1 (0.7x)	1.7 (1.0x)
Bunny	10.6	18 (1.6x)	27 (2.5x)
Cloth	14.2	25 (1.7x)	36 (2.5x)
Fairy	19.4	70 (3.6x)	74 (3.8x)
Crytek	32.9	116 (3.5x)	115 (3.5x)
Conference	28.9	125 (4.3x)	125 (4.3x)
Sportscar	38.6	110 (2.9x)	135 (3.5x)
Italian	43.9	145 (3.3x)	169 (3.9x)
Babylonian	60.2	219 (3.6x)	230 (3.8x)
Kitchen	77.1	436 (5.6x)	352 (4.6x)
Dragon	124.8	379 (3.0x)	413 (3.3x)
Buddha	147.9	493 (3.3x)	520 (3.5x)
Livingroom	150.4	833 (5.5x)	685 (4.6x)
Lion	230.1	800 (3.5x)	766 (3.3x)
Geom. mean	-	(3.0x)	(3.3x)

primitive intersection test [23]. We use the SAH cost parameters $C_i = 1.2, C_l = 0, C_t = 1$ given for GPUs by Karras and Aila [23] in order to be comparable with previous work. SAH costs are normalized to a full SAH sweep [52]. Quality was not evaluated for FastTree.

Two toplevel builds are evaluated to recover quality, as discussed in Subsection 4.6. HLBVH+SAH is modeled with a cycle-level simulator, while for HLBVH+ATRBVH, we run a single-threaded C++ implementation on a NVidia Jetson TK1 board with a Tegra K1 SoC. The ATRBVH build runs two iterations over the toplevel nodes and uses a treelet size of 8.

For power analysis, we extract *switching activity information files* from the previous simulations, and perform power analysis with Synopsys Design Compiler. The constructed trees are loaded

Table 7. Power analysis results, average of 14 scenes.

Max. BW (GB/s)	12.8	25.6	42.7
Mem. traffic (GB/s)	9.2	13.9	18.9
Logic power (mW)	58.1	61.8	62.7
DRAM power (mW)	507.2	814.8	1112.4
Total power (mW)	565.3	876.6	1175.2

into a software ray tracer to verify correctness and compute tree quality. External memory power is determined by exporting DRAM command traces from Ramulator to DRAMPower [10]. We estimate the power and energy consumption of a 64b DRAM component by doubling the figures for a 32b component, as these could be combined into a 64b component.

6.3 Results

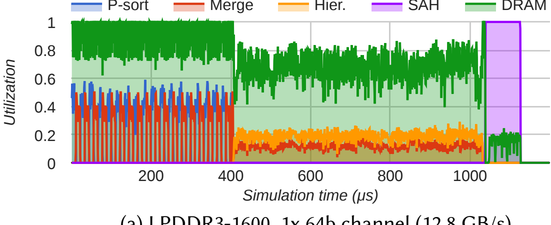
In Table 3, the resulting build performance and tree quality is compared to related work. Even with the *slow* memory option, the present design is over 2× faster than the state of the art binned SAH unit, and with the *fast* memory option, 5× faster (6.3× including the Toasters scene, but the 1ms runtime reported in that scene is too imprecise for comparison). Compared to the state of the art k-d tree builder by Liu et al. [34] with the same 12.8 GB/s bandwidth, MergeTree is 4.7× faster. With the fast memory option, the proposed unit is within a factor of two of the desktop GPU HLBVH builder, which has 7.5× more memory bandwidth, and an orders of magnitude larger chip area and power envelope. ATRBVH gives a higher tree quality, but is, on average, 7.4× slower than HLBVH.

6.3.1 Area, Power and Memory Traffic. The unit was successfully synthesized and meets timing constraints at 1GHz. The cell area and power breakdown of the synthesized unit is shown in Table 4. Table 5 shows an area comparison to related work. The proposed unit has ca. 2.5× less area than a binned SAH builder [13].

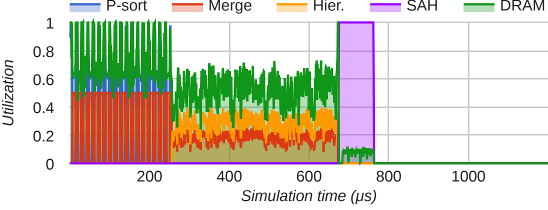
The results of power analysis are shown in Table 7. The main result is that over 90% of total power consumption in the design comes from the DRAM interface. There are some straightforward optimizations to reduce the on-chip power: for example, a multi-bank scratchpad could be used in place of the current expensive dual-port SRAM. However, since the power consumed by the hardware unit itself is negligible compared to DRAM, the improvements from optimization would also be marginal.

Table 6 compares our memory traffic to related work. The proposed builder generates 3.0× less traffic than hardware binned SAH, and 3.3× less than a GPU build - the radix sort stage of the GPU build alone generates roughly as much traffic as our complete build. Our builder also achieves very high bus utilizations of 72%, 54% and 44% of the memory bandwidth on the slow, medium and fast interface options, respectively.

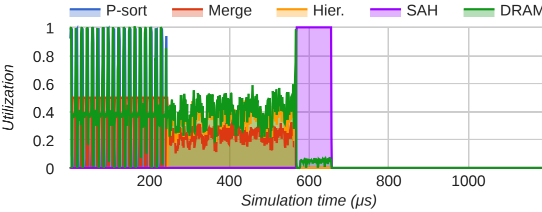
The above results show that the energy consumption and build speed of MergeTree are largely determined by the amount of memory traffic generated. A straightforward conversion of HLBVH to hardware, without the proposed memory traffic optimizations, would likely have a ca. 3× higher energy consumption and runtime, almost as high as the binned SAH builder [13].



(a) LPDDR3-1600, 1x 64b channel (12.8 GB/s)



(b) LPDDR3-1600, 2x 64b channel (25.6 GB/s)



(c) LPDDR3-1333, 4x 64b channel (42.7 GB/s)

Fig. 6. Cycle-level simulation traces for the *sibenik* scene with three memory interface options. Utilizations of the multi-merge and hierarchy emitter components cap at 50%. Computation consists of partial sorts, a multi-merge phase, and an optional toplevel SAH build. With the slow bus, performance is limited by memory bandwidth. With the fast bus, memory latencies and compute pipeline throughputs become the bottleneck, particularly in the partial sort stage.

6.3.2 *Tree quality.*

MergeTree builds slightly higher-quality trees than the GPU HLBVH builder since we generate large leafs for primitives with identical Morton codes, as in [42], while the GPU builder organizes these primitive ranges with an arbitrary subtree and emits one leaf per triangle. Nevertheless, our plain HLBVH trees have low quality, on average 148%, compared to 104% for Doyle et al. [13].

The evaluated toplevel builds give significant tree quality improvements: HLBVH+SAH to an average of 118% and HLBVH+ATRBVH to 111%, only ca. 5% worse than a binned SAH build. The hardware binned SAH builder has an insignificant runtime compared to the LBVH, but consumes extra chip area. Our naíve single-threaded software implementation of toplevel ATRBVH is already fast enough for real-time construction on a mobile SoC, and could run all scenes at 30FPS except for Lion, which has demanding geometry.

6.4 System Level Comparison

From the previous results, it is apparent that MergeTree gives a similar tradeoff as desktop GPU HLBVH: it is very fast and energy-efficient compared to prior work, at the cost of reduced tree quality, which can be mostly recovered with postprocessing toplevel builds. We can conjecture that a binned SAH builder is advantageous in small scenes where the build effort is minuscule relative to rendering, and the proposed builder becomes advantageous in larger scenes. The exact tradeoff depends on the particular scene and visual effects being displayed. This subsection further quantifies the system-level effects of builder selection by modeling a larger system which includes rendering hardware. The model focuses on system energy consumption per frame, as it is a main figure of interest in mobile systems and simplifies modeling. We start out from the premise that the BVH tree for the complete scene is rebuilt for each frame, and a viewpoint is then rendered at a 1280x720 resolution. The scenes and viewpoints used are shown in Table 3. Benchmarks are run for primary ray rendering, as well as diffuse lighting with one sample per pixel, limited to three bounces. The latter is representative of incoherent secondary rays.

The main components of the present model are a fixed-function rendering accelerator, combined with MergeTree, binned SAH and ATRBVH hardware builders. Moreover, toplevel build combinations of HLBVH+SAH and HLBVH+ATRBVH are tested which combine two hardware builders. For MergeTree, we use accurate energy figures based on post-synthesis power analysis and DRAMPower. For the binned SAH builder, we estimate memory traffic and FPU operation counts as described in Subsection 5.1, and then obtain a lower-bound energy model by assuming fully utilized FPUs and long, consecutive burst accesses to DRAM. For the ATRBVH builder, memory accesses and FPU operations are likewise counted from program code. No high-performance hardware architecture has been published for ATRBVH, but given the input size for toplevel builds, even a serial hardware unit performing one FPU operation per cycle is sufficient to process all test scenes at over 100fps. Finally, the rendering accelerator is modeled after the traversal and intersection unit of SGRT [32], and simulated at cycle-level as described in [50]. Some common assumptions are used when modeling the hardware units. The units reside on a mobile SoC which is fabricated with a 28nm process technology, and equipped with the 25.6GB/s memory interface described earlier. As with MergeTree, off-chip memory accesses are modeled with Ramulator and DRAMPower. Caches and SRAMs are parametrized with CACTI 6.5 [38]. Floating-point unit energy is based on the figures of Galal et al. [15], with linear process scaling, as in the GPUSimPow simulator [35]. Reciprocal calculation is estimated to take as much energy as 3 FLOPs, as in [33]. All units beside MergeTree operate at 500MHz.

6.4.1 *Rendering hardware model.*

The modeled accelerator architecture is shown in Fig. 8: it consists of separate fixed-function pipelines for tree traversal and primitive intersection. Scenes are rendered with a software ray tracer, from which a traversal trace is extracted and fed to a cycle-level hardware simulator, which traces utilizations for all components in Fig. 8. The power consumption of each component is determined by multiplying a dynamic power term with the utilization and adding a static power term. For the

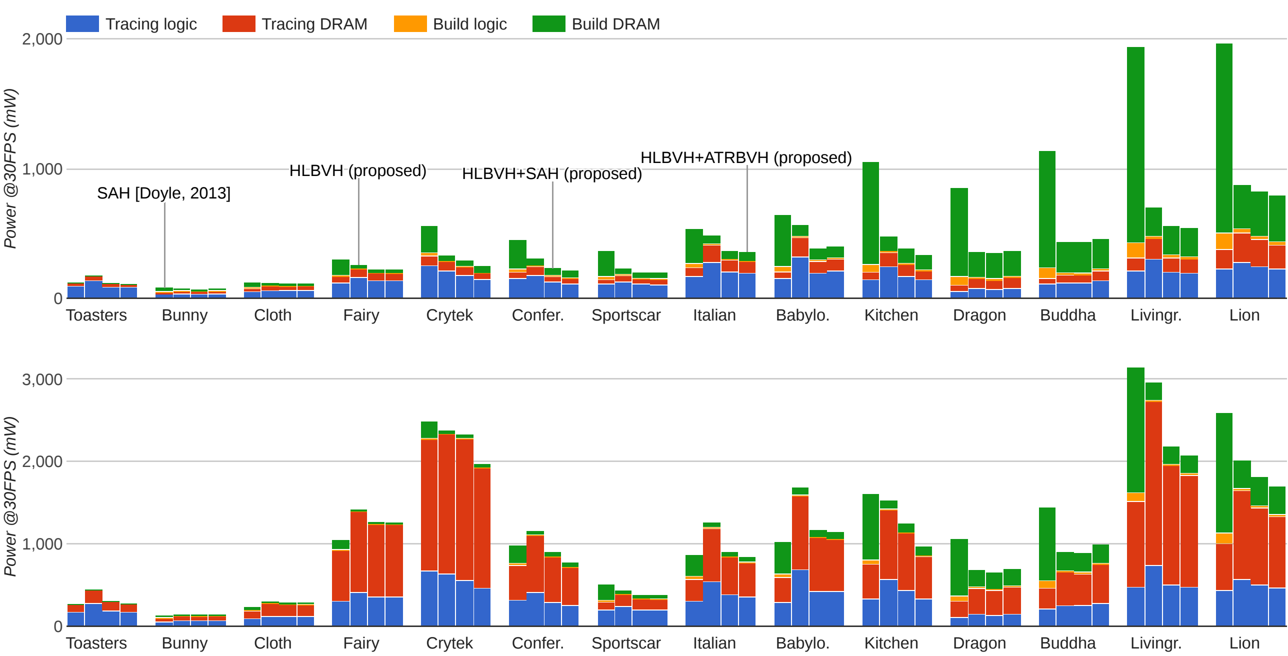


Fig. 7. System energy with only primary rays (top) and diffuse secondary rays (bottom), expressed as power at 30FPS for comparison to mobile GPUs. Four alternatives are modeled: HLBVH only, binned SAH only, and HLBVH with binned SAH and ATRBVH toplevel builds. HLBVH build energy is from RTL simulation, while tracing and binned SAH energy are based on higher-level models. The proposed HLBVH builder reduces build energy at the cost of worse tree quality, which increases tracing energy. Toplevel builds remove much of the quality penalty. Tracing complexity is weakly related to scene size, while build energy grows, and becomes dominant in large scenes. Averaged results are shown in Table 8.

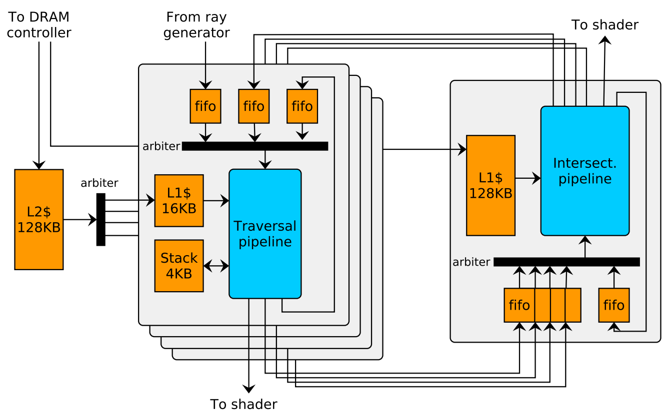


Fig. 8. Ray tracing accelerator simulated for system level energy modeling. The accelerator is modeled after the traversal and intersection unit in SGRT [32], with a two-AABB node layout [31]. All shown components are simulated at cycle level.

fixed-function pipelines, the energy consumption of floating-point adds, multiplications and reciprocals is counted. The architecture and simulation flow is described in more detail in [50], and is unchanged from that work, except the SRAM and FPU models are updated to a 28nm process.

In addition to tree construction and traversal, shading is the third main component in the rendering process. Shading has a very wide range of complexity: we experimented with adding the energy cost of minimal shading and pixel output to the model, i.e., Phong shading with a directional light, but this had minimal effect on the total power. On the other hand, sufficiently complex shading may dominate the rendering process [29]. Mobile ray tracing would likely opt for inexpensive shaders at first. As shading cost is independent of tree quality, we omit it from the model.

6.4.2 Results. The system-level energy results are shown scene by scene in Fig. 7, and summarized in Table 8. It is visible that the energy cost of tree construction scales asympotically faster with scene size than traversal, and with binned SAH, dominates the energy profile in large scenes. Though the binned SAH builder performs significant floating-point computation, most of its energy consumption is also due to DRAM accesses. MergeTree uses on average ca. 3.2× less energy. In primary ray tracing, the build energy savings are sufficient to make HLBVH preferred in all scenes except Toasters. In large scenes, tree construction dominates the system energy, and HLBVH gives significant savings.

With incoherent secondary rays, the energy footprint of ray tracing grows significantly, and is dominated by memory traffic. As such, tree quality has a larger effect on system energy. Moreover, the tracing energy penalty of the proposed builder is larger than predicted by SAH cost. Toplevel builds reduce system energy, but less than predicted by SAH. Regardless, in large (>500K triangle)

Table 8. System energy with different builders, main results. Energy normalized to binned SAH [13], averaged over 14 scenes.

Primary rays, all scenes

Energy	Binned SAH	HLBVH	HLBVH +SAH	HLBVH +ATRBVH
Build	100%	**32%**	33%	33%
Trace	**100%**	143%	114%	112%
Total	100%	71%	59%	**58%**

Primary rays, large scenes

Energy	Binned SAH	HLBVH	HLBVH +SAH	HLBVH +ATRBVH
Build	100%	**22%**	22%	22%
Trace	**100%**	148%	122%	122%
Total	100%	41%	37%	**37%**

Diffuse rays, all scenes

Energy	Binned SAH	HLBVH	HLBVH +SAH	HLBVH +ATRBVH
Build	100%	**32%**	33%	33%
Trace	**100%**	163%	134%	128%
Total	100%	110%	91%	**87%**

Diffuse rays, large scenes

Energy	Binned SAH	HLBVH	HLBVH +SAH	HLBVH +ATRBVH
Build	100%	**22%**	22%	22%
Trace	**100%**	165%	141%	137%
Total	100%	79%	68%	**65%**

scenes the cost of tree construction is significant enough that the proposed builder consistently reduces system energy compared to binned SAH.

For comparison with mobile power budgets and GPUs, the energy results in Fig. 7 are presented as power at a fixed 30FPS frame rate. Diffuse, animated ray tracing in our model with HLBVH+ATRBVH dissipates between 143..2077mW of power, and primary ray tracing between 72..791mW. For a point of comparison, in the benchmarks of Pathania et al. [43], recent mobile games on an Odroid-XU+E platform dissipate ca. 2..3W, of which ca. 0.8..1.8W is used in the mobile GPU. These results suggest that MergeTree allows the ray tracing of large (>500K triangle) dynamic scenes in a mobile power envelope. However, in the most demanding scenes there is only limited margin for complex shading. A binned SAH builder would already use most of the mobile power budget for these scenes.

7 LIMITATIONS AND FUTURE WORK

One difficulty in the proposed design is handling scenes of over $\frac{M^2}{2B}$ primitives (2 million triangles with the evaluation setup), as they require more than one multi-merge pass. It is simple to add control logic for multiple passes, but the use of AABBs as sorting elements is then suboptimal. Another possibility is to enlarge the scratchpad: doubling the memory size M quadruples the model size that can be processed in one pass. In our experience at least a 512KB scratchpad memory can run at 1GHz; this would be sufficient for scenes of 8M triangles. It would also be interesting to evaluate, as a replacement for double-buffering, the other well-known multi-merge sort read scheduling technique of *forecasting*, used by, e.g., Barve et al. [7]. Forecasting introduces a second heap which stores the final elements of each buffer, and uses it to fetch replacement buffers in an optimal order. Forecasting would, again, double the size of scene that can be sorted in a single pass with a given scratchpad size.

Recent trends in ray tracing accelerators are toward techniques which reduce the cost of ray traversal while complicating tree construction, e.g., compressed BVHs [25] and treelet scheduling [3]. In future architectures incorporating these features, the cost of tree construction will be emphasized even more than in straightforward single-precision traversal as evaluated in this article. We are extending MergeTree to generate compressed trees.

This article focused on mobile ray tracing, but the design also has interesting applications in, e.g., collision detection as in [11], and with minor modifications, construction of point set k-d trees [22] and data sorting.

8 CONCLUSION

In this article, we proposed MergeTree, the first hardware accelerator architecture for the HLBVH algorithm, which forms the basis for the highest-performance GPU tree construction algorithms. Novel techniques were proposed to adapt HLBVH into a streaming, serial hardware form, which is suitable for mobile systems with limited power budgets, due to reduced memory traffic. Our results show significant improvements over previous state of the art [13] in terms of build performance, silicon area, memory traffic and energy consumption, at the cost of reduced tree quality, which can be mitigated with inexpensive toplevel builds.

The proposed architecture substantially increases the size of animated scenes which could be rendered by a mobile ray tracing accelerator in real time, and also has applications outside ray tracing. System level modeling showed that the cost of tree construction begins to rival the cost of real-time rendering in large scenes. MergeTree gives significant system energy savings in these scenes.

ACKNOWLEDGMENTS

This work was financially supported by the TUT graduate school and the Academy of Finland (decision #297548, PLC). The 3D models used are courtesy of Ingo Wald (Fairy), Andrew Kensler (Toasters), Yasutoshi Mori (Sportscar), Frank Meinl (Crytek Sponza), Jonathan G. (Italian, Babylonian), Anat Grynberg and Greg Ward (Conference), Naga Govindaraju, Ilknur Kabul and Stephane Redon (Cloth), the Stanford Computer Graphics Laboratory (Bunny, Dragon), and the SceneNet library [21] (Livingroom, Kitchen). Crytek Sponza and Dragon have modifications courtesy of Morgan McGuire [37].

REFERENCES

[1] Attila T Áfra and László Szirmay-Kalos. 2014. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum* 33, 1 (2014), 129–140.
[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.
[3] Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics.* 113–122.
[4] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High-Performance Graphics.* 145–149.
[5] Tomas Akenine-Möller and Jacob Strom. 2008. Graphics processing units for handhelds. *Proc. IEEE* 96, 5 (2008), 779–789.
[6] Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC).*
[7] Rakesh D Barve, Edward F Grove, and Jeffrey Scott Vitter. 1997. Simple randomized mergesort on parallel disks. *Parallel Comput.* 23, 4 (1997), 601–631.

[8] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 2. 538–547.
[9] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proc. ACM/SIGDA Int. Symp. Field-programmable gate arrays*. 151–160.
[10] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. DRAMPower: Open-source DRAM power & energy estimation tool. (2012). Retrieved Feb 30, 2017 from http://www.drampower.info
[11] Erwin Coumans. 2017. Bullet physics library. (2017). Retrieved Mar 6, 2017 from http://www.bulletphysics.org
[12] Leonardo R Domingues and Helio Pedrini. 2015. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proc. High-Performance Graphics*. 13–20.
[13] Michael Doyle, Colin Fowler, and Michael Manzke. 2013. A hardware unit for fast SAH-optimized BVH construction. *ACM Transactions on Graphics* 32, 4 (2013), 139:1–10.
[14] Michael Doyle, Ciaran Tuohy, and Michael Manzke. 2017. Evaluation of a BVH Construction Accelerator Architecture for High-Quality Visualization. *IEEE Transactions on Multi-Scale Computing Systems* (2017).
[15] Sameh Galal and Mark Horowitz. 2011. Energy-efficient floating-point unit design. *IEEE Transactions on computers* 60, 7 (2011), 913–922.
[16] Per Ganestam and Michael Doggett. 2016. SAH guided spatial split partitioning for fast BVH construction. In *Computer Graphics Forum*, Vol. 35. 285–293.
[17] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and faster HLBVH with work queues. In *Proc. High-Performance Graphics*. 59–64.
[18] Kirill Garanzha, Simon Premože, Alexander Bely, and Vladimir Galaktionov. 2011. Grid-based SAH BVH construction on a GPU. *The Visual Computer* 27, 6-8 (2011), 697–706.
[19] Jeffrey Goldsmith and John Salmon. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7, 5 (1987), 14–20.
[20] Aggelos Ioannou and Manolis GH Katevenis. 2007. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking* 15, 2 (2007), 450–461.
[21] Ilan Kadar and Ohad Ben-Shahar. 2013. SceneNet: A Perceptual Ontology Database for Scene Understanding. *Journal of Vision* 13, 9 (2013), 1310–1310.
[22] Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proc. High-Performance Graphics*. 33–37.
[23] Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proc. High-Performance Graphics*. 89–99.
[24] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011), 7–17.
[25] Sean Keely. 2014. Reduced precision hardware for ray tracing. In *Proc. High-Performance Graphics*. 29–40.
[26] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* PP, 99 (2015), 1–1.
[27] Dirk Koch and Jim Torresen. 2011. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*. 45–54.
[28] Daniel Kopta, Konstantin Shkurko, J Spjut, Erik Brunvand, and Al Davis. 2015. Memory considerations for low energy ray tracing. *Computer Graphics Forum* 34, 1 (2015), 47–59.
[29] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: wavefront path tracing on GPUs. In *Proc. High-Performance Graphics*. 137–143.
[30] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.
[31] Jaedon Lee, Won-Jong Lee, Youngsam Shin, Seokjoong Hwang, Soojung Ryu, and Jeongwook Kim. 2014. Two-AABB traversal for mobile real-time ray tracing. In *SIGGRAPH Asia Mobile Graph. Interact. Appl.* 14.
[32] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. High-Performance Graphics*. 109–119.
[33] Wei Liu and Alberto Nannarelli. 2012. Power efficient division and square root unit. *IEEE Trans. Comput.* 61, 8 (2012), 1059–1070.
[34] Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *Proc. Design, Automation & Test in Europe Conference & Exhibition*. 1595–1598.
[35] Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. 2013. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *Proc. IEEE Int. Symp. Perf. Analysis Syst. Software*. 97–106.
[36] J David MacDonald and Kellogg S Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 3 (1990), 153–166.
[37] Morgan McGuire. 2011. Computer graphics archive. (2011). Retrieved Feb 30, 2017 from http://graphics.cs.williams.edu/data/meshes.xml
[38] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. *CACTI 6.0: A tool to model large caches*. Technical Report. HP Laboratories. 22–31 pages.
[39] J. Nah, H. Kwon, D. Kim, C. Jeong, J. Park, T. Han, D. Manocha, and W. Park. 2014. RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices. *ACM Transactions on Graphics* 33, 5 (2014), 162:1–15.
[40] Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, Dinesh Manocha, and Tack-Don Han. 2015. HART: A Hybrid Architecture for Ray Tracing Animated Scenes. *IEEE Transactions on Visualization and Computer Graphics* 21, 3 (2015), 389–401.
[41] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2011. T&I engine: traversal and intersection engine for hardware accelerated ray tracing. *ACM Transactions on Graphics* 30, 6 (2011), 160.
[42] Jacopo Pantaleoni and David Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. High-Performance Graphics*. 87–95.
[43] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. 2015. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *Proc. Design Automation Conference*. 201.
[44] PowerVR. 2015. PowerVR Ray Tracing. (2015). Retrieved Feb 30, 2017 from https://imgtec.com/powervr/ray-tracing/
[45] Maxim Shevtsov, Alexei Soupikov, Alexander Kapustin, and Nizhniy Novorod. 2007. Ray-triangle intersection algorithm for modern CPU architectures. In *Proc. of GraphiCon*, Vol. 2007. 33–39.
[46] Hojun Shim, Nachyuck Chang, and Massoud Pedram. 2004. A compressed frame buffer to reduce display power consumption in mobile systems. In *Proc. Asia and South Pacific Design Automation Conf.* 819–824.
[47] Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: a multicore hardware architecture for real-time ray tracing. *Trans. Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1802–1815.
[48] Joseph Spjut, Daniel Kopta, Erik Brunvand, and Al Davis. 2012. A mobile accelerator architecture for ray tracing. In *Proc. Workshop on SoCs, Heterogeneous Architectures and Workloads*.
[49] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2015. MergeTree: a HLBVH constructor for mobile systems. In *SIGGRAPH Asia Technical Briefs*. 12.
[50] Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, and Jarmo Takala. 2016. Multi bounding volume hierarchies for ray tracing pipelines. In *SIGGRAPH Asia Technical Briefs*. 8.
[51] Ingo Wald. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE Symp. Interactive Ray Tracing*. 33–40.
[52] Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics* 26, 1 (2007), 6.
[53] Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics* 24, 3 (2005), 434–444.

Received March 2017; accepted June 2017