MergeTree: A HLBVH Constructor for Mobile Systems

Timo Viitanen Matias Koskela Pekka Jääskeläinen Heikki Kultala Jarmo Takala [*]

Tampere University of Technology, Finland

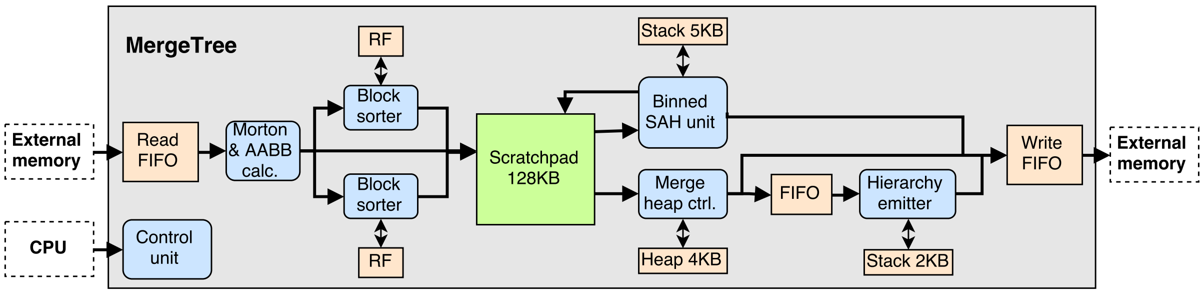


Figure 1: *Proposed hardware architecture. RF: Register File. SAH: Surface Area Heuristic.*

Abstract

Powerful hardware accelerators have been recently developed that put interactive ray-tracing even in the reach of mobile devices. However, supplying the rendering unit with up-to date acceleration trees remains difficult, so the rendered scenes are mostly static. The restricted memory bandwidth of a mobile device is a challenge with applying GPU-based tree construction algorithms. This paper describes MergeTree, a BVH tree constructor architecture based on the HLBVH algorithm, whose main features of interest are a streaming hierarchy emitter, an external sorting algorithm with provably minimal memory usage, and a hardware priority queue used to accelerate the external sort. In simulations, the resulting unit is faster by a factor of three than the state-of-the art hardware builder based on the binned SAH sweep algorithm.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

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

1 Introduction

Ray-tracing is a promising rendering technique for mobile systems. The runtime of the algorithm scales more with the number of pixels than the number of drawn primitives, making it ideal for small displays. Effects such as shadows, reflection and global illumination are more natural to express than in traditional rasterization based architectures. In proposed augmented reality applications physically based ray-tracing would seamlessly overlay virtual objects on a real environment. In recent years, increasingly powerful mobile ray-tracing accelerators have been developed [Lee et al. 2013; Nah et al. 2014]. However, these units have mostly been reliant on a pre-built acceleration datastructure, restricting them to static scenes.

Currently the fastest GPU tree builders such as HLBVH [Garanzha et al. 2011] use a *Bounding Volume Hierarchy* (BVH) datastructure, and are based on sorting primitives according to the *morton codes* of their centroids, so this approach is interesting for a custom hardware unit. However, a direct adaptation of the GPU algorithms to a mobile context is hampered by memory bandwidth limitations: recent high-end mobile SoCs have an order-of-magnitude less memory bandwidth than high-end GPUs. Special techniques are necessary to cope with this environment, for example, mobile GPUs conserve bandwidth by means of tiling and texture compression.In this paper, we propose the first custom hardware architecture for HLBVH, which minimizes bandwidth usage through external sorting and streaming hierarchy emission. The architecture is evaluated by building a cycle-level simulator.

2 Related work

In the RayCore architecture [Nah et al. 2014], the scene geometry is split into two parts with separate acceleration trees: a small dynamic part is rebuilt on each frame using a hardware unit, and a larger static part is pre-built. Each ray traverses both trees to find the nearest intersection. The HART system [Nah et al. 2015] 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 it. Our design would be useful as a component in either system: in a refit-based renderer the faster asynchronous rebuilds would help the system adapt to sudden changes in the scene, while in a two-part system a larger dynamic part could be kept up to date.

Doyle et al. [2013] propose the first BVH construction hardware unit, which performs a binned *Surface Area Heuristic* (SAH) sweep. Our proposed design uses a similar but scaled-down unit as a subcomponent for the HLBVH top-level build stage.

The FastTree unit [Liu et al. 2015] uses Morton codes for k-d tree construction, and is the fastest k-d tree constructor hardware so far. 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.

[*]e-mail:{timo.2.viitanen, matias.koskela, pekka.jaaskelainen, heikki.kultala, jarmo.takala}@tut.fi

3 Algorithm

Table 1: *Sorting algorithm comparison in tree construction, 32B IO words / primitive.*

	Prim. read	Sorting	BVH write	total
Sort Morton codes (8B per element)				
Radix-16 counting sort	10	48	24	82
Multimerge, 1 pass	10	4	24	38
Multimerge, 2 pass	10	8	24	42
Sort AABBs (28B per element)				
Radix-16 counting sort	10	168	14	192
Multimerge, 1 pass	10	14	14	38
Multimerge, 2 pass	10	28	14	52

The LBVH algorithm [Lauterbach et al. 2009] produces BVH trees by sorting the input primitives according to their Morton codes. After this, a BVH hierarchy can be emitted by binning primitives based on the bits of their Morton codes: e.g., the highest level split is generated, by placing primitives with MSB 0 and 1 in the left and right children of the top node, respectively. Lower hierarchy levels correspond to lower bits. This process is called *hierarchy emission*. The HLBVH algorithm [Garanzha et al. 2011] improves tree quality by generating the top levels of the tree with the slower binned SAH sweep algorithm.

Sorting accounts for much of the memory traffic in HLBVH, so we attempt to optimize it by referring to literature on *external sorting* data on slow magnetic disc drives. One optimal sorting algorithm in this environment is the *multimergesort* [Aggarwal and Vitter 1988]. 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 M-sized blocks. After this, the algorithm runs multimerge passes which merge M/B sorted blocks into a larger block. Table 1 compares the minimum memory accesses of multimerge sorting in the context of tree construction to a typical radix-16 sort which takes eight passes through the data. If the scene can be processed in one pass, the multimerge sort uses less memory by a factor of two.

In addition to the choice of algorithm, we must also choose whether to sort the original primitives (40 bytes per item), their *Axis-Aligned Bounding Boxes* (AABBs) (28 bytes), or Morton code-reference pairs (8 bytes). At first Morton code sorting appears optimal, but it has the drawback that, when generating the hierarchy, we need to random access every triangle in the scene to generate AABBs. Table 1 shows that for a 1-pass multimerge sort, AABB sorting uses as much memory bandwidth as Morton code sorting, and the memory accesses are split more evenly between different stages of the algorithm. Therefore, we focus on AABB sorting in this work. However, this causes some overhead for large scenes that require more than one pass.

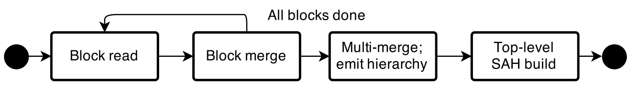


Figure 2: *Control state diagram.*

4 Architecture

The overall proposed MergeTree architecture is showin in Figure 1, and it operates according to the state diagram of Figure 2. The unit first performs $\lceil \frac{N}{M} \rceil$ partial sorts, which are split into *block read* and *block merge* states. Next, the sorted M-sized buffers are merged together, and the result is fed into a streaming hierarchy emitter, which produces BVH nodes. Finally, a high-level hierarchy is built with a separate binned SAH sweep unit.

4.1 Primitive input

We internally store data elements as AABBs augmented with a memory reference and a Morton code, for a total of 256 bits. The Morton codes are computed on the fly when reading in data, and omitted when writing out to memory.

4.2 Multi-merge hardware

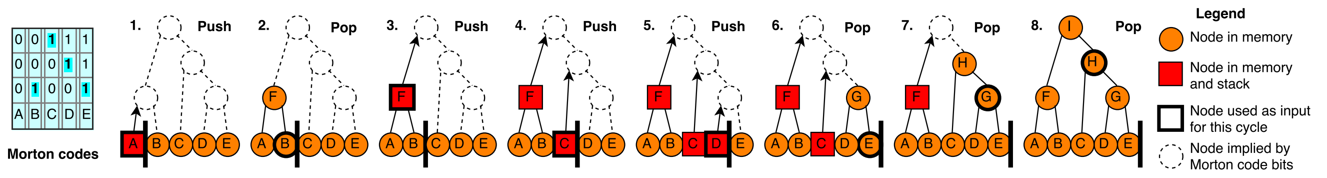
A standard software implementation of the multimergesort algorithm places merge candidate values from each buffer into a heap datastructure. On each iteration, a minimum value is taken from the top of the heap, and the next value from the same block is inserted. A sequential heap implementation takes $\log n$ compare-swaps to insert an element, which is too slow. Fortunately, the process can be pipelined by inserting separate memory blocks and control hardware for each level of the heap. A similar hardware structure was proposed by Moon et al. [2000] to implement a hardware priority queue for networking hardware: we refer to their paper for a full discussion of the design tradeoffs.

It is difficult to fit any logic on the same clock cycle with the access delay of a large scratchpad, so we have the merge hardware output AABBs once per two cycles: one cycle is reserved for the memory access, and the other is used to perform the top-level heap insertion, which produces the next read address. The heap insertion cycle can be used to move data from the read queue into the scratchpad. This allows high clock frequencies to be reached, and also fits well with the streaming hierarchy emitter whose straightforward implementation processes, on average, one input per two cycles. The scratchpad is split into at least two banks, to allow concurrent reads of the selected AABB (for hierarchy emission) and the next AABB (for heap insertion).

We schedule memory reads by means of *double buffering*: each block in the main memory is represented by two buffers on the scratchpad, and when one buffer has been processed, a read is queued to replace it. If two buffers are processed from the same block before replacement data arrives, the merge heap stalls. It would also be interesting to evaluate the other well-known read scheduling technique of *forecasting*, where a second heap stores the final element of each buffer, and predicts which buffer will have to be replaced first.

4.3 Streaming hierarchy emission

In order to minimize external memory traffic, we stream sorted AABBs to a hardware state machine which implements a serial LBVH hierarchy emission algorithm. Figure 3 shows a visual example of the algorithm in operation. The generated inner node topology corresponds to the shown Morton code bits. Each node is output when sufficient primitives have been read to determine its child bounding boxes, resulting in a bottom-up order. Stack entries represent inner nodes whose right child is unknown: they consist of a left child AABB and a hierarchy level. For example, the final read leaf in Figure 3 gives sufficient information to generate the last 3 inner nodes on cycles 6, 7 and 8. Nodes with identical Morton codes



Morton codes
Push
Pop
Legend
Node in memory
Node in memory and stack
Node used as input for this cycle
Node implied by Morton code bits

Figure 3: *Example of streaming hierarchy emission, which processes the given sequence of leaf nodes (A-E) and morton codes (left) to produce inner nodes (F-I). Hierarchy level of inner nodes is determined by (emphasized) highest differing bit of corresponding Morton codes.*

are handled by extracting differing bits from their indices, which generates a somewhat balanced subtree. Generating n inner nodes requires $2n$ stack operations. We assume the hardware can perform one stack operation per cycle. The full algorithm is as follows:

```
while True do
    input ← nextInput ;
    read nextInput from FIFO;
    diff ← highest diff. bit of input and nextInput ;
    while ¬ stack.empty() ∧ stack.top().diff < diff do
        BVHNode n(stack.pop().aabb, input) ;
        input ← n.aabb ;
        output[idx++] ← n ;
    end
    stack.push(input, diff) ;
end
```

4.4 Partial sort

The merge sorting hardware described above is straightforward to reuse for the scratchpad-sized partial sort, by configuring the merge heap so that each buffer on scratchpad is the final one is in its block, and no further buffers are fetched. Then the only additional work needed is to sort every buffer-sized subblock prior to merging. This is easy to implement concurrently with data reading, by streaming the read data into a small number of buffer-sized subblock sorters, which use a state machine to perform an insertion sort at a rate of one compare-and-swap per second.

4.5 Top-level SAH build

The HLBVH algorithm improves tree quality by constructing the highest levels of the tree with a binned SAH sweep. This can be performed with the hardware unit proposed by Doyle et al. [2013], but since the input size is small, a scaled-down version could be used with fewer computational resources. To evaluate the design, we use 8 bins, one parallel worker, and six pipelines for split AABB generation from bin AABBs, SAH computation, and split plane computation. For evaluation, we assume conservatively that the unit takes 32 cycles to do these tasks after processing a range of primitives, but it is likely that an optimized design can interleave some of this computation with another processing sweep.

5 Evaluation

Table 2: *Memory traffic comparison (MB).*

	[Doyle et al. 2013]	Proposed
Cloth (92K)	25	15
Conference (331K)	120	53
Dragon (871K)	380	140

In order to evaluate the architecture, we developed a cycle-level C++ simulator. The main components are simulated with cycle-accurate state machines. We model the external memory using the GDDR3 memory controller model from GPGPUSim, configured as 1GHz, 32-bit, dual channel, which is close to LPDDR3 in recent SoCs [Lee et al. 2013]. We assume an operating frequency of 1GHz, which we think is realistic at least in a recent process technology. The parameters of the hardware unit are set at $M = 4096$, $B = 8$, resulting in a unit with a 128KB scratchpad memory, which handles up to 1M triangles in one pass, and performs a 256-way merge. We include four block sorters for scalability. For comparison, Liu et al. [2015] have a 172KB scratchpad and Doyle et al. [2013] use 432KB.

The simulator was run on five test scenes, and the resulting trees were verified in a software ray-tracer. In Tables 3 and 2, the performance and memory traffic are compared to related work. Our construction speed is ca. 3 times faster than previous BVH hardware, 5 times faster than k-d tree hardware, and catches up with a GPU implementation of HLBVH except for the largest Dragon scene, though we are still behind the latest GPU implementation of LBVH. The speedup can be attributed to our choice of sorting algorithm: as shown in Table 1, a HLBVH based on a conventional radix sort would use 2× more memory bandwidth and, therefore, construction time. Figure 4 shows an example run of the simulator, using the small *sibenik* scene for clarity. The different execution states are visible: first the the unit runs block sorts which alternate between utilizing the subblock sorters and the merge heap. Most of the execution time is spent on the multimerge phase, which is clearly memory-limited: the merge heap runs at less than one-third of maximum capacity. Finally, the top-level SAH build is more compute-intensive and uses little memory. The unit utilized 82% of theoretical maximum bandwidth in this scene, and an average of 87% across all scenes, where the SAH build is less significant. Much of the idle time was due to switching between reads and writes.

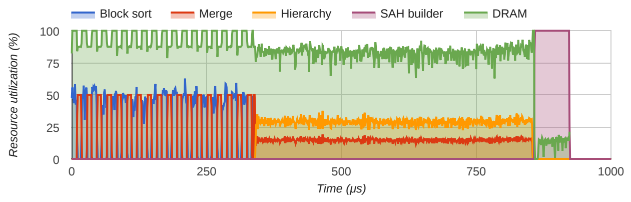


Figure 4: *Simulation trace for the sibenik scene.*

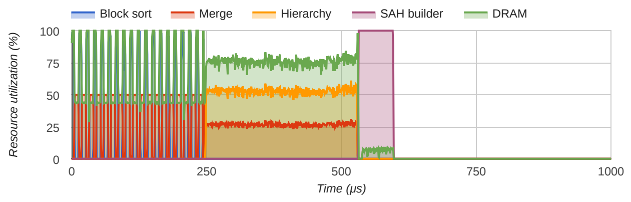


Figure 5: *Simulation trace with memory bus doubled to 32GB/s.*

Table 3: *Performance comparison: build times in milliseconds.*

	GTX 480 HLBVH [Garanzha et al. 2011]	GTX 480 LBVH [Karras 2012]	Hardware k-d tree [Liu et al. 2015]	Hardware SAH BVH [Doyle et al. 2013]	Hardware HLBVH (proposed)	
Mem. BW (GB/s)	133.9	133.9	16	36	16	32
Sibenik (75K)	-	-	6.6	-	0.92	0.60 (-35%)
Cloth (92K)	4.8	-	-	3	1.02	0.62 (-39%)
Fairy (174K)	-	0.99	10.8	-	1.98	1.24 (-37%)
Conference (331K)	6.2	1.45	17.2	11	3.79	2.54 (-33%)
Dragon (871K)	8.1	3.64	52.2	30	9.74	5.93 (-39%)

We performed chip area estimation using the same methodology as [Doyle et al. 2013; Liu et al. 2015]. The results indicate an area of $5.6mm^2$ at 65nm and $1.4mm^2$ when scaled to 28nm as in Liu et al. The proposed unit is expected to have low power consumption even compared to previous custom hardware since it performs fewer external memory accesses as shown in Table 2. The quality of the produced trees was estimated by computing their SAH and comparing against a binned SAH sweep with 16 splits. The HLBVH trees were 7% worse on average. The top-level SAH sweep improved estimated rendering performance by an average of 11% and took an average of 2% of the runtime, so it appears to be a good tradeoff except for very small scenes. We investigated how well the design scales to near-future mobile memory buses by rerunning the simulations with a doubled bus clock rate. As shown in Table 3, the build time decreased on average by 37%. Figure 5 shows the example *sibenik* simulator run repeated with the doubled bandwidth. The main scaling bottleneck is the block merge state, where the maximum output of the merge hardware is insufficient to saturate the bus. A possible optimization is to use double buffering, i.e., read data to one half of the scratchpad while merging the other half.

6 Limitations

One difficulty in the proposed design is handling scenes of over $\frac{M^2}{2B}$ primitives (1M with the evaluation setup), as they require more than one multimerge pass. It is simple to add control logic for multiple passes, but the AABB-sorting algorithm 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 16M triangles. We also have a built-in assumption that the application software can supply at least approximate scene bounds for generating efficient Morton codes: otherwise they need a separate pass through the inputs, increasing memory accesses by ca. 26%.

7 Conclusion and Future Work

We described a HLBVH-based BVH constructor architecture optimized for use in mobile devices. The unit is three times faster than the state of the art, showing that although HLBVH is associated with massively parallel GPU implementations, it is also suitable for a fast serial hardware design. The memory usage of the unit is close to a theoretical lower bound for a sorting-based tree construction, and simulations indicate that it runs fast enough to saturate current mobile memory buses, therefore, the build performance is close to maximum achievable in current mobile systems for this type of algorithm. The unit was evaluated as a standalone builder, but it could also be used as part of a refit-based system such as [Nah et al. 2015]. We are in the early stages of building an FPGA prototype, and are interested in also performing CMOS synthesis and place&route for more accurate area and power estimates.

Acknowledgements

This research is supported by the TUT doctoral program, Finnish Funding Agency for Technology and Innovation (project Parallel Acceleration 2, funding decision 40081/14), and ARTEMIS JU under grant agreement no 621439 (ALMARVI).

References

AGGARWAL, A., AND VITTER, J. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9, 1116–1127.

DOYLE, M., FOWLER, C., AND MANZKE, M. 2013. A hardware unit for fast SAH-optimized BVH construction. *ACM Transactions on Graphics 32*, 4, 139.

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*, ACM, 59–64.

KARRAS, T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, Eurographics Association, 33–37.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, vol. 28, Wiley Online Library, 375–384.

LEE, W., SHIN, Y., LEE, J., KIM, J., NAH, J., JUNG, S., LEE, S., PARK, H., AND HAN, T. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, 109–119.

LIU, X., DENG, Y., NI, Y., AND LIL, Z. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 1595–1598.

MOON, S.-W., REXFORD, J., AND SHIN, K. 2000. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers 49*, 11, 1215–1227.

NAH, J., KWON, H., KIM, D., JEONG, C., PARK, J., HAN, T., MANOCHA, D., AND PARK, W. 2014. RayCore: a ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics 33*, 5, 132.

NAH, J., KIM, J., PARK, J., LEE, W., PARK, J., JUNG, S., PARK, W., MANOCHA, D., AND HAN, T. 2015. HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3, 389–401.