PLOCTree: A Fast, High-Quality Hardware BVH Builder

TIMO VIITANEN, MATIAS KOSKELA, PEKKA JÄÄSKELÄINEN, ALEKSI TERVO, and JARMO TAKALA, Tampere University of Technology, Finland

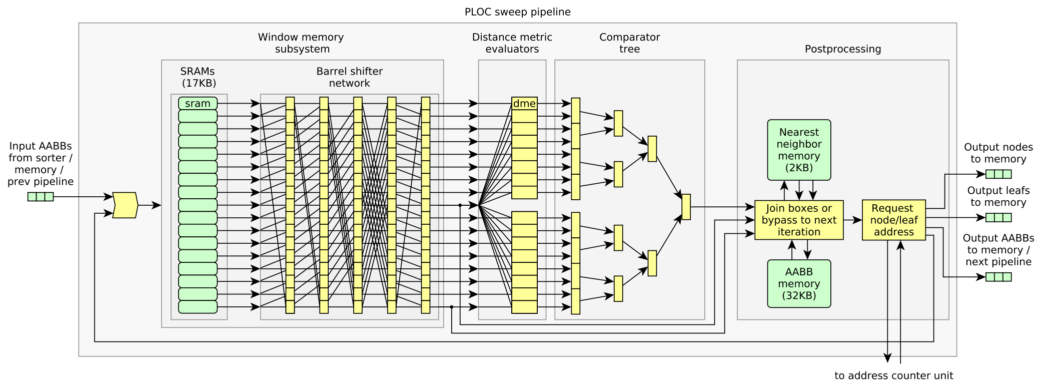

PLOC sweep pipeline
Window memory subsystem
SRAMs (17KB)
sram
Barrel shifter network
Distance metric evaluators
dme
Comparator tree
Postprocessing
Input AABBs from sorter / memory / prev pipeline
Nearest neighbor memory (2KB)
Join boxes or bypass to next iteration
AABB memory (32KB)
Request node/leaf address
Output nodes to memory
Output leafs to memory
Output AABBs to memory / next pipeline
to address counter unit

Fig. 1. Architecture of a PLOC sweep pipeline, the main component of PLOCTree (Nearest-neighbor search radius $R = 8$, sweep count $S = 32$).

In the near future, GPUs are expected to have hardware support for real-time ray tracing in order to, e.g., help render complex lighting effects in video games and enable photorealistic augmented reality. One challenge in real-time ray tracing is dynamic scene support, that is, rebuilding or updating the spatial data structures used to accelerate rendering whenever the scene geometry changes. This paper proposes PLOCTree, an accelerator for tree construction based on the Parallel Locally-Ordered Clustering (PLOC) algorithm. Tree construction is highly memory-intensive, thus for the hardware implementation, the algorithm is rewritten into a bandwidth-economical form which converts most of the external memory traffic of the original software-based GPU implementation into streaming on-chip data traffic. As a result, the proposed unit is 3.9 times faster and uses 7.7 times less memory bandwidth than the GPU implementation. Compared to state-of-the-art hardware builders, PLOCTree gives a superior performance-quality tradeoff: it is nearly as fast as a state-of-the-art low-quality linear builder, while producing trees of similar Surface Area Heuristic (SAH) cost as a comparatively expensive binned SAH sweep builder.

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

Additional Key Words and Phrases: ray tracing, bounding volume hierarchy, dynamic scene, graphics hardware

Authors' address: Timo Viitanen, viitanet@gmail.com; Matias Koskela, matias.koskela@tut.fi; Pekka Jääskeläinen, pekka.jaaskelainen@tut.fi; Aleksi Tervo, aleksi.tervo@tut.fi; Jarmo Takala, jarmo.takala@tut.fi, Tampere University of Technology, Korkeakoulunkatu 10, 33720, Tampere, Finland.

2577-6193/2018/8-ART35 $15.00
https://doi.org/10.1145/3233309

ACM Reference Format:
Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Aleksi Tervo, and Jarmo Takala. 2018. PLOCTree: A Fast, High-Quality Hardware BVH Builder. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 35 (August 2018), 19 pages. https://doi.org/10.1145/3233309

1 INTRODUCTION

Graphics processing units are undergoing a rapid development in order to render 3D scenes with increasingly complex visual effects. In the near future, GPUs are evolving to include hardware acceleration for real-time *ray tracing*, a technique used so far mainly for high-quality offline rendering [McGuire et al. 2018]. Recently hardware architectures for ray tracing have been a subject of intensive research, as detailed in a recent survey by Deng et al. [2017]. One challenge specific to ray tracing is that, in order to reach acceptable performance, the renderer needs the 3D scene geometry to be organized in a spatial acceleration data structure.

Currently, the most popular data structure is the *Bounding Volume Hierarchy* (BVH), where each inner node divides a set of geometric primitives into two or more subsets, and stores the pointers and *Axis Aligned Bounding Boxes* (AABB) of the subset nodes. In an animated scene, the BVH has to be rapidly rebuilt or updated for each animation frame. Several hardware accelerators have been proposed for this purpose [Doyle et al. 2013; Nah et al. 2015; Viitanen et al. 2017; Woop et al. 2006].

Fixed-function hardware accelerators are often 2-3 orders of magnitude more energy- and area-efficient than programmable systems [Hameed et al. 2010]. However, large improvements are more difficult to come by in memory-intensive and floating-point heavy workloads like ray traversal and tree construction. As a result, recent research on ray tracing accelerators revolves around algorithm-level optimizations to reduce memory traffic, such as compressed trees [Vaidyanathan et al. 2016] and optimized data access patterns [Shkurko et al. 2017]. Likewise, the state of the art hardware tree builders [Doyle et al. 2013; Viitanen et al. 2017] are designed at the algorithm level to avoid accesses to external memory.

While the easiest performance and efficiency improvements in circuit design can be obtained at the algorithm level, the state of the art tree update accelerators are based on somewhat archaic algorithms: binned *Surface Area Heuristic* (SAH) sweep, refitting and *Linear BVH* (LBVH). In the recent years, software builders on CPUs and GPUs have seen rapid progress in algorithms, and several methods have been proposed that give clearly better performance-quality tradeoffs, e.g., a quality similar to binned SAH with a runtime closer to LBVH. It seems that these algorithms could translate into improved hardware units. In this paper, we identify a recent algorithm as particularly suitable for hardware implementation, that is, *Parallel Locally-Ordered Construction* (PLOC) by Meister and Bittner [2018], and propose a new hardware builder architecture, PLOCTree.

In addition to having good performance and quality in software, PLOC has characteristics that suggest large potential improvements from hardware acceleration. In particular, PLOC comprises tens of sweeping passes over an input primitive array which, eventually, reduce it bottom-up into a BVH tree. On GPU, each pass incurs multiple kernel launches, which read and write large data buffers. This results in significant external memory traffic when the intermediate buffers are too large to fit in cache. We find that, in a hardware implementation, most of the data traffic of PLOC can be kept on-chip, and multiple iterations can reuse the same hardware pipeline.

The main contributions in this paper are:

- A fully pipelined hardware architecture for a single PLOC sweep, including nearest-neighbor search, merging, and compaction.
- A scheme to share the same pipeline resource between multiple PLOC sweeps, keeping the inter-sweep traffic on-chip, and to parallelize the computation by placing multiple pipelines in a sequence.

- Evaluation of reduced-precision surface area computation for bounding boxes as a power optimization.

The proposed hardware architecture is evaluated by means of logic synthesis and power analysis on a 28nm process technology.

2 RELATED WORK

The quality of a BVH is often estimated based on the *Surface Area Heuristic* (SAH), which estimates the likelihood of intersection with a given tree node based on the surface area of its bounding box. Traditionally, BVHs are built top-down based on SAH with the top-down *SAH sweep* [MacDonald and Booth 1990] and *binned SAH sweep* [Wald 2007] algorithms. These methods begin with all primitives in a single large leaf, and recursively split it along axis-aligned planes, selecting a plane from several candidates which gives the best SAH. Surface area can also be used to build trees by combining nodes bottom-up, selecting node pairs to merge such that the surface area of the combined AABB is minimized [Walter et al. 2008].

Since SAH based builds are expensive, faster methods have been proposed. The fastest builders are based on the LBVH algorithm of Lauterbach et al. [2009], which sorts triangles according to their Morton codes, and then produces a hierarchy based on Morton code bit-patterns. Increasingly optimized LBVH implementations have been proposed, most recently by Apetrei [Apetrei 2014]. LBVH produces trees of very low quality, so several works attempt to postprocess the tree to recover quality [Domingues and Pedrini 2015a; Garanzha et al. 2011; Karras and Aila 2013; Pantaleoni and Luebke 2010]. Recently, Morton code sorting has also been used to accelerate bottom-up surface-area builds by limiting the nearest-neighbor search to a small window, in the AAC [Gu et al. 2013] and PLOC [Meister and Bittner 2018] builders.

Aside from rebuilding BVHs from scratch, it is possible to reuse the BVH topology from a previous animation frame, and *refit* the bounding volume coordinates to match the geometry in the new frame [Wald et al. 2007]. This is inexpensive compared to a full rebuild, but tree quality tends to degrade over the course of a longer animation. Several works work around this drawback by, e.g., asynchronously refreshing the BVH with full rebuilds [Ize et al. 2007], postprocessing the BVH to recover quality [Kopta et al. 2012], or constructing the initial BVH for good refitting performance [Bittner and Meister 2015]. A drawback is that refitting is limited to mesh deformations, and cannot handle topological changes between animation frames.

Several hardware accelerators have been proposed for tree update. Refitting has been accelerated in the DRPU [Woop et al. 2006], and more recently, HART [Nah et al. 2015] ray tracing systems. Doyle's [2013] accelerator implements the binned SAH sweep algorithm, with loop pipelining to reduce memory traffic. The builder was recently evaluated on FPGA [Doyle et al. 2017]. MergeTree [Viitanen et al. 2017] implements the LBVH algorithm, likewise optimizing bandwidth by using an external sorting algorithm and performing LBVH hierarchy emission and AABB computation as a streaming postprocessing step. The commercial PowerVR Wizard ray tracing GPU includes a BVH builder, though only limited details are available [McCombe 2014]. Builders have also been proposed for k-d trees, another popular ray tracing data structure [Liu et al. 2015; Nah et al. 2014].

This paper proposes the first hardware builder to implement a modern algorithm [Meister and Bittner 2018] based on Morton code sorting and postprocessing. As a result, it has a favorable speed-quality tradeoff compared to previous hardware builders. The proposed builder has an AABB sorting subsystem similar to MergeTree [Viitanen et al. 2017], but replaces the LBVH hierarchy emission with a novel hardware architecture implementing the PLOC algorithm.

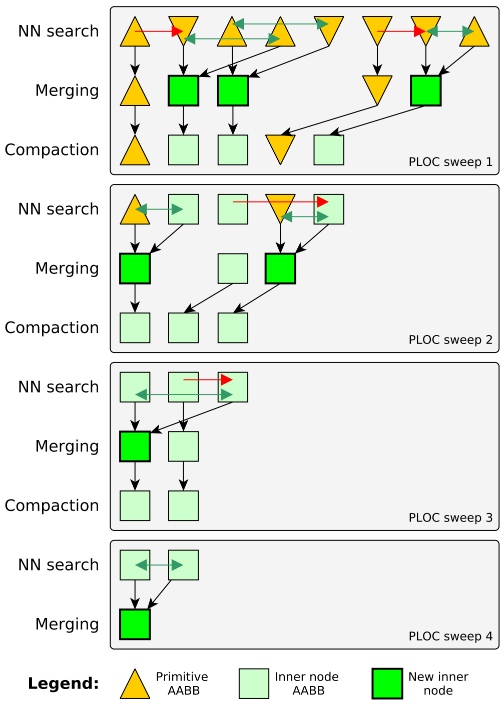


Fig. 2. Demonstration of the PLOC algorithm [Meister and Bittner 2018] for input size $N = 8$.

3 ALGORITHM

The basic concept of the GPU PLOC algorithm [Meister and Bittner 2018] is shown in Fig. 2. The algorithm is given as input an array of N primitive AABBs. The first step is to assign each AABB a Morton code based on its AABB centroid. Next, the input AABBs are sorted in Morton code order. The main part of the algorithm then constructs a BVH tree by applying repeated iterations over the array which we here call *PLOC sweeps*. Each sweep is made up of three steps:

- **Nearest-neighbor search**: For each AABB, compute a distance metric to all AABBs in a local window of radius R. Select the AABB with lowest distance as a nearest neighbor.
- **Merging**: Find pairs of AABBs that are mutual nearest neighbors. Merge each such a pair into a BVH inner node. Place the AABB of the newly created node in one of the original array positions, and leave the other empty. AABBs that do not have a mutual nearest neighbor are left unchanged.
- **Compaction**: Remove empty elements in the array.

PLOC sweeps are repeated until there is only one element left in the AABB array, which corresponds to the root node of the output BVH. The array size shrinks roughly exponentially with the number of iterations.

While several recent build algorithms generate an initial hierarchy similarly to the LBVH algorithm [Lauterbach et al. 2009], using Morton code bit patterns to generate octree-like halfway splits, PLOC instead makes use of the property that the distance on the z-curve is a good proxy for spatial locality. Hence, the global nearest-neighbor for an AABB is likely to be found nearby in the Morton code sorted array.

```
1  for(int idx=0; idx<N; idx++)
2    nearest[idx]
3     = index of nearest neighbor in [idx-R .. idx+R]
4
5  int aabb_out_idx = 0;
6  for (int idx=0; idx<N; idx++) {
7    if(nearest[nearest[idx]] == idx) {
8      if(nearest[idx] > idx) {
9        out_node = {input[idx], input[nearest[idx]]};
10       out_node_aabb = AABB(out_node);
11       out_node_aabb.ptr = node_out_idx;
12       node_output[node_out_idx++] = out_node;
13       output[out_idx++] = out_node_aabb;
14     }
15   }
16   else
17     output[out_idx++] = input[idx];
18 }
```

Fig. 3. Basic serial implementation of a single sweep in the PLOC algorithm.

In the original algorithm, each PLOC sweep consists of multiple kernel executions, each of which reads and writes data buffers in the GPU memory. The data sizes for the first few sweeps are close to the original primitive count and, therefore, operate on inputs too large to cache. The main goal in this paper is to represent PLOC in a harware-oriented form which, as far as possible, replaces traffic to external memory buffers with on-chip data streams, reducing the use of DRAM bandwidth. This is done in two parts. First, a single PLOC sweep is represented as a streaming process, eliminating traffic between the NN search, merging, and compaction stages. Second, multiple sweeps are performed on-chip, eliminating the traffic between the sweeps.

3.1 Streaming PLOC

An example serial PLOC sweep is shown in C pseudocode in Fig. 3. Here, `input` and `output` are the input and output AABB arrays of a sweep. The code is split into two passes. First, a local nearest-neighbor is found for each AABB and saved in an array (lines 1 . . . 3). If two neighbors have equal distances, the one with a lower index is selected. Then on a second pass, elements are merged into inner nodes based on the `nearest` array (lines 5 . . . 15). The AABB of a new node is stored in the location of the first component AABB (line 13), and the second component AABB is omitted from the output (line 8). AABBs that are not mutual nearest-neighbors of any other AABB are passed unchanged to the output (line 17). AABBs in the algorithm consist of bounding box coordinates, a child index `ptr` and a leaf marker field. The input AABBs are all marked as leafs. The `AABB()` function computes the bounding box of a two-AABB node, which is marked as an inner node.

In order to make the single-sweep algorithm suitable for hardware implementation, we would like to merge the two passes into a single loop, and inside this loop, eliminate random accesses to large memory arrays. The passes can be merged by noting that the second pass of the algorithm only refers to the elements of the `nearest` array within radius R of the current element. That is, when we have computed up to `nearest[i]`, there is enough information to run the second pass

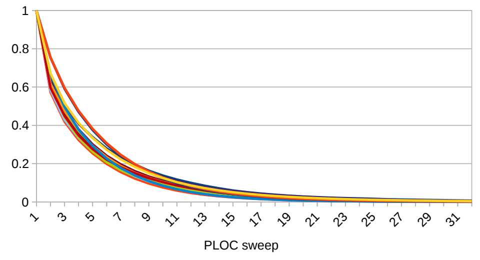
```
AABB input[B];
int nearest[B];
bool pair_l[B], pair_r[B];

int c_idx = 0;
int node_out_idx = 0;
while (not done) {
  input[mask(c_idx+R)] = fifo_in.pop();
  int c_nearest_rel
    = relative index of nearest neighbor in
    input[mask(c_idx-R)] .. input[mask(c_idx+R)]
  int c_nearest = mask(c_idx + c_nearest_rel);
  nearest[c_idx] = c_nearest;

  if(c_nearest_rel<0 && nearest[c_nearest]==c_idx) {
    pair_l[c_nearest] = true;
    pair_r[c_idx] = true;
  }

  int l_idx = mask(c_idx - R);
  int l_nrst = nearest[l_idx];
  if(pair_l[l_idx]) {
    out_node = {input[l_idx], input[l_nrst]};
    out_node_aabb = AABB(out_node);
    out_node_aabb.ptr = node_out_idx++;
    node_output.push( out_node );
    fifo_out.push( out_node_aabb );
  }
  else if(pair_r[l_idx]) {
  }
  else
    fifo_out.push( input[l_idx] );

  pair_l[mask(c_idx-R*2)] = false;
  pair_r[mask(c_idx-R*2)] = false;
  c_idx = mask(c_idx+1);
}
```

Fig. 4. Streaming, hardware-oriented version of a single PLOC sweep.

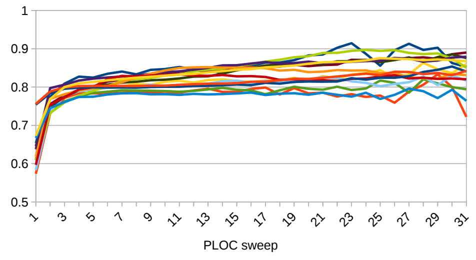
until `i-R`. We can, then, fuse the loops as long as the computations for the second pass stay R elements behind the first.

In order to eliminate random accesses to `input` and `nearest`, note that iteration `i` of the second pass only references elements of `nearest` in range `i-R` ... `i+R`, and `input` in range `i` ... `i+R`. Hence, after computing the second pass up to iteration `i`, we can discard all elements of `input` and `nearest` older than `i-R`. The remaining elements can be stored in, e.g., small ring buffers.

Fig. 4 shows a PLOC sweep algorithm with these modifications, moreover, external data accesses are explicitly made through FIFOs. Here, `input` and `nearest` are small local buffers of size `B` (`B` is a power-of-two). The function `mask()` is a logical *and* operation with `B-1`, causing the buffers to wrap around. `c_idx` corresponds to `pass 1 idx` in Fig. 3 and `l_idx` to `pass 2 idx`. Also, they



(a) PLOC input size by sweep.



(b) Ratio of output to input size per PLOC sweep.

Fig. 5. Behavior of PLOC sweep input size during build, 15 test scenes.

are the indices of the left and center elements in the $2R + 1$ element window searched for the nearest-neighbors in each iteration.

Following our hardware implementation, each iteration of the Algorithm in Fig. 4 finds the possible mutual nearest-neighbor of the input array element at `c_idx`, and then communicates the found pair to the second pass side of the iteration via two additional buffers, `pair_l` and `pair_r`. On encountering an element with `pair_l` set, the second pass joins it to its nearest neighbor to produce a new node (lines 22 ...28). `pair_r` directs the second pass to discard the right-hand element of the pair (29). In retrospect it would be clearer to follow the pass separation in Fig. 3 more closely, finding mutual nearest-neighbors for the element at `l_idx` and omitting the extra buffers.

We now have a loop where each iteration reads one element from an input FIFO, writes up to one element to `fifo_out` and `fifo_node_out`, and other than these external communications, only references constant-sized local data.

3.2 Resource Allocation

From the previous subsection, we have an algorithm suitable for hardware implementation of a single PLOC sweep. A PLOC build consists of tens of sweeps. It is straightforward to keep data traffic between sweeps on-chip by feeding the output stream of one sweep as an input to the next. However, if each sweep is executed by a separate hardware resource, this poses a load balancing problem: sweep sizes vary heavily over the build, from the initial primitive count N down to 2.

We approach load balancing by mapping multiple PLOC sweeps to the same hardware resource: with multiple resources, later resources can be assigned more sweeps to balance the workload. To guide the assignment, we microbenchmark the PLOC algorithm on several test scenes to characterize the behavior of sweep size over the course of a PLOC build. Fig. 5a shows input size as a function of sweep index, and Fig. 5b the ratio between each sweep's output and input size.

Fig. 5b shows that the first sweep reduces the input size by a larger fraction than the later ones, on average 36%, after which subsequent sweeps reduce size by an average of 22%. As a result, sweep size falls roughly exponentially and rapidly becomes insignificant, as shown in Fig. 5a. On average, if the input size of the initial sweep is N, the summed number of elements over all sweeps is $4.1N$. Hence, for example, if we want to load balance between two hardware resources, we should assign enough sweeps to the first resource to have a total workload of ca. $2N$, and the rest of our on-chip sweeps to the second resource. On average, sweeps 1–3 have a total input size of $2.1N$, so in this case it is good to assign ca. 3 iterations to the first resource.

3.3 Reduced Precision Distance Metric

A PLOC sweep pipeline implementing the Algorithm in Fig. 4 with a throughput of, e.g., 1 element per cycle will require tens of floating-point units for distance metric evaluation. In order to save silicon area and power, we compute distance metrics at lower arithmetic precision. Reduced-precision arithmetic was earlier found effective in ray traversal by Keely [2014] and Vaidyanathan et al. [2016]. Simply dropping precision to, e.g., IEEE half-precision has two main failure modes that severely degrade quality:

- Regions of the BVH that are outside the half-precision dynamic range, being clamped to maximum float value
- Catastrophic cancellations occur when subtracting the AABB bounds to compute AABB width along each axis.

The first error source can be worked around by keeping exponent size at the 8 bits of IEEE single-precision floating point. Since most of the complexity of floating-point units is in the significand datapath, this has little effect on the savings from reduced precision. The second error source can be eliminated by using full-precision floats for the initial width computation, then truncating the results, and using low precision for the rest of the surface area computation.

3.4 Adaptive Collapsing

One key component of GPU PLOC [Meister and Bittner 2018] is an adaptive, SAH-based treelet collapsing step to produce large leafs. It is straightforward to make a collapsing decision in a streaming manner, but more complex to generate the output leaf array. When merging two large leafs, they are not guaranteed to map to consecutive regions of the leaf array: hence, a naive implementation would have to read back both component leafs to be compressed, and write a consecutive large leaf, leaving undesirable gaps in the leaf array. In this work, we only collapse in the easy case of merging two single-triangle leafs. Moreover, we omit SAH computation, and always generate a 2-triangle leaf when a PLOC sweep iteration is merging two triangles.

4 HARDWARE ARCHITECTURE

In this Section, we describe a hardware architecture for a tree builder based on the streaming PLOC version described in Section 3.1. We first look into the detailed implementation of a single PLOC sweep pipeline, and then describe a complete system architecture based on the pipeline.

4.1 PLOC sweep pipeline architecture

The hardware architecture of a PLOC sweep pipeline is shown in Fig. 1. The pipeline is fed with the AABBs of the PLOC input array in Morton code order, and produces BVH node and leaf outputs, as well as an AABB stream for the next sweep. Each AABB is 32B and consists of 6 single-precision coordinates, an index, and a leaf size indicator. A leaf size of 0 indicates the AABB is of an inner node, and the index is to the output node array – otherwise, it points to the triangles or leafs. The inputs of the first sweep are the Morton code sorted AABBs of triangles, with leaf size set to 1. The pipeline is designed for a throughput of 1 input AABB per cycle.

The pipeline is mostly a translation of the algorithm of Fig. 4 to hardware, but we split the `input` buffer into two copies: one optimized as a sliding-window memory for the purpose of supplying nearest-neighbor search with a window of $2R + 1$ consecutive AABBs, and another optimized for random single-AABB reads.

The sweep pipeline consists of a sliding-window input buffer, followed by nearest-neighbor search and AABB merging. In order to share a pipeline between S PLOC sweeps, the window memory contains the sliding-windows of more than one sweep. Each input AABB is tagged with a

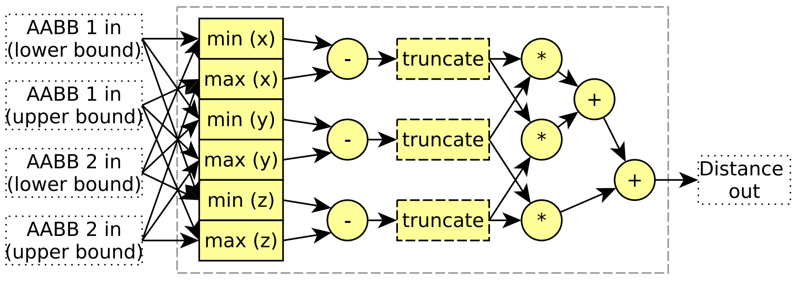


Fig. 6. Distance metric evaluator.

$\lceil log_2 S \rceil$ bit sweep identifier, which selects the sweep to use, and is propagated through the pipeline. Input AABBs have a sweep identifier of 0. If a firing of the pipeline produces an output AABB, and its identifier is $S - 1$, the AABB is output from the pipeline. If the identifier is less than $S - 1$, the identifier is incremented, and the AABB is fed back to the start of the pipeline. An arbiter at the start of the pipeline gives a higher priority to feedback AABBs, avoiding the need for a FIFO buffer.

4.1.1 Window Memory. The task of the window memory is to maintain a set of sliding-window buffers, one for each iteration assigned to the pipeline. When an AABB is input to the pipeline, it first goes to the right side of the corresponding sliding window, pushing all elements previously in the window to the left. The memory then outputs the full window of $2R + 1$ AABBs.

In the current work, the window memory is implemented as a multi-bank memory followed by a barrel-shifter shuffle network. Consecutive inputs to the same window are written to successive banks, and the output is rotated so that latest input always appears in the same output position. Small window memories are implemented as flip-flops and large (>16 elements per bank) as SRAMs. For small S, it may be interesting to instead implement the window memory as a set of shift registers followed by a multiplexer.

4.1.2 Distance Metric Evaluator. The $2R + 1$ output AABBs from the window memory are fed into a bank of $2R$ *distance metric evaluator* (DME) units. Each DME takes two AABBs as input, one of which is the center box of the window, and computes the surface area of an AABB that covers both input boxes. Optionally, reduced precision may be used for part of the computation, as detailed in Subsection 3.3. The structure of a DME is shown in Fig. 6. The DME bank accounts for all the floating-point computation in the system, and much of the pipeline latency.

4.1.3 Comparator Tree. The comparator tree receives $2R$ distances from the DME bank as inputs, and outputs the relative index of the AABB with the lowest distance, ranging from $-R$ to R. The index is also used as a tiebreaker, selecting the leftmost of AABBs with equal distance.

4.1.4 Postprocessing. The previous stages of the pipeline find the relative nearest-neighbor indices for each input aray element. In addition to the nearest-neighbor index, the postprocessing unit receives the left and center AABBs of the corresponding sliding window as input. The unit then performs the rest of the work of the algorithm in Fig. 4. At this point, the nearest-neighbor indices are stored and used to find the AABB pairs that are mutual nearest-neighbors. Based on the found pairs, AABBs are either output as they are, consumed, or joined with another AABB, emitting node and leaf data. Node and leaf addresses are requested from an address counter unit shared between the PLOC sweep pipelines, if there is more than one pipeline in the design.

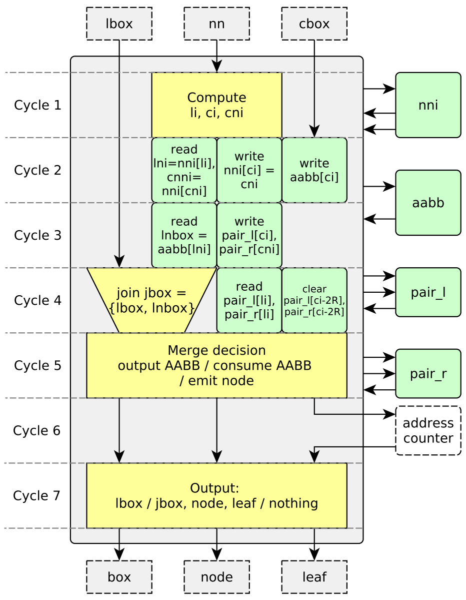

lbox
nn
cbox
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Compute li, ci, cni
read lni=nni[li], cnni= nni[cni]
write nni[ci] = cni
write aabb[ci]
read lnbox = aabb[lni]
write pair_l[ci], pair_r[cni]
join jbox = {lbox, lnbox}
read pair_l[li], pair_r[li]
clear pair_l[ci-2R], pair_r[ci-2R]
Merge decision output AABB / consume AABB / emit node
Output: lbox / jbox, node, leaf / nothing
nni
aabb
pair_l
pair_r
address counter
box
node
leaf

Fig. 7. Postprocessing pipeline architecture.

A stage-by-stage architecture of the postprocessing pipeline is shown in Fig. 7. The pipeline initially computes `c_idx`, `c_nearest` and `l_idx` as in Fig. 4 (here shortened to `ci`, `cni` and `li`, respectively) and writes the AABB and nearest-neighbor index to local SRAMs. The nearest-neighbor memory is then read to determine whether the current center node is the right side of a nearest-neighbor pair (Cycle 2). A detected pair is marked in the `pair_l` and `pair_r` registers (Cycle 3). The rest of the pipeline deals with the left AABB: based on `pair_l` and `pair_r`, it is either consumed (if `pair_r` is set), joined with its nearest-neighbor to emit a node (if `pair_l` is set) or output as-is (if both are false). The joint node AABB is speculatively computed on Cycle 4, and the merge decision itself is made on Cycle 5. On Cycle 6, if the pipeline is emitting a node, it requests write addresses to the node and leaf output tables from the address counter unit; these addresses are used to finalize the output node and AABB data on Cycle 7.

4.1.5 Boundary Conditions. In order to handle the end of the input stream, after feeding the complete input array to the pipeline, we add $2SR$ dummy AABBs with lower and upper bounds set to minimum and maximum representable single-precision floating-point values, respectively. This special-case AABB is always its own nearest neighbor, and never the nearest neighbor of any of the valid AABBs. This forces the algorithm to run also with the final input AABBs as the center and left elements of the nearest-neighbor search window. Special logic is placed at the input of the first PLOC pipeline to insert the dummy AABBs. It should be noted that this is a source of inefficiency in later, small PLOC sweeps, where DMEs are often evaluating distances to dummy AABBs. However, the large majority of the work in PLOC is done in the middle of early, large sweeps, where the DME banks can be fully utilized.

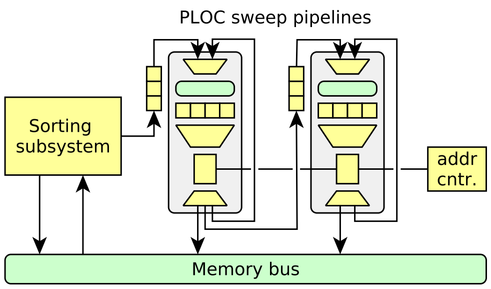


Fig. 8. PLOCTree system architecture with 2 builders.

4.2 System architecture

The basic system architecture of PLOCTree is shown in Fig. 8, and its main components are a *sorting subsystem* followed by one or more *PLOC sweep pipelines*. Each pipeline has a separate parameter S_i for the maximum number of sweeps it can handle. Moreover, an *address counter* module is included to assign memory addresses to nodes output from multiple sweep pipelines.

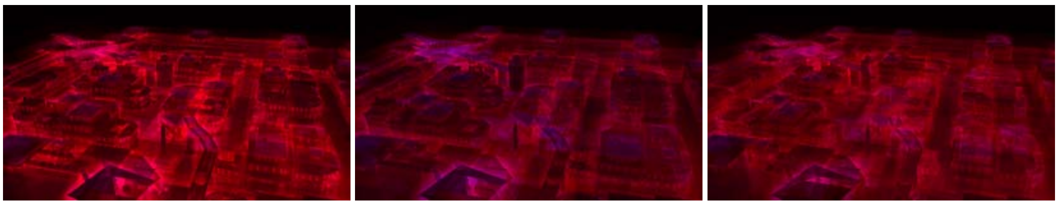
Our main requirements for a hardware sorter are that it should produce an ordered output stream (as opposed to, e.g. a parallel radix sort), and preferably be based on a bandwidth-economical external sorting algorithm. We use the AABB sorting subsystem of the MergeTree builder [Viitanen et al. 2017] which has these properties. Instead of feeding the sorted AABBs to a LBVH hierarchy emitter, they are input into the first PLOC sweep pipeline. As with MergeTree, the input array of PLOCTree consists of primitive AABBs annotated with precomputed Morton codes. In a practical use, though, it may be preferable to directly read primitive data, and include hardware for AABB and Morton code computation.

The MergeTree sorter has two main operating states: *block sort* (divided into read and write substates), which sorts blocks of AABBs small enough to fit in the on-chip scratchpad, and *multimerge*, which produces the final sorted array. In the proposed system, the PLOC sweep pipelines are active during the multimerge state and inactive in the block sort state. Moreover, we add a third operating state, *repeat*, where an AABB array is read directly from memory and written to the PLOC pipelines, bypassing the sorter. In case the PLOC build of a scene needs more sweeps performed than the number that can be run on-chip ($\sum S_i$), an intermediate AABB array is written out to external memory, and the repeat state is used one or more times to finish the build. Given a $\sum S_i$ of tens of iterations, the repeat sweep sizes are very small.

The build is parallelized by increasing the number of sweep pipelines in sequence. As discussed in Section 3.2, the total work is ca. 4.1 times the input triangle count. Assuming full utilization and good load balancing between pipelines, a single-pipeline builder has a throughput of ca. $\frac{1}{4}$, a two-pipeline builder of $\frac{1}{2}$, and a four-pipeline builder has a throughput close to 1. It should be noted that these are very idealized assumptions: in real use, load balancing is imperfect, and the tree build has startup and wind-down phases where some of the resources are under-utilized. The MergeTree sorter has an ideal throughput of $\frac{1}{2}$, so in order to roughly match its throughput, we focus on a two-input builder design.

5 EVALUATION

For evaluation, PLOCTree is implemented at the *register transfer level* (RTL) in SystemVerilog. We evaluate an instance with two PLOC sweep pipelines of radius $R = 8$, configured to handle



(a) LBVH with binned SAH toplevel build. (n_{box} = 102.5, n_{tri} = 7.1)

(b) Binned SAH build. (n_{box} = 76.8, n_{tri} = 10.4)

(c) PLOC build (**proposed**). (n_{box} = 90.3, n_{tri} = 7.9)

Fig. 9. Traversal cost example, *Italian* scene. Red: ray-AABB tests, blue: ray-triangle intersection tests. n_{box}: mean ray-AABB tests per pixel, n_{tri}: mean ray-triangle tests per pixel.

S_1 = 4 and S_2 = 32 sweeps, respectively. The sorter has the same configuration as in [Viitanen et al. 2017], with a 256KB scratchpad and 32-bit Morton codes, and capable of sorting arrays up to 2M AABBs. Enhanced Morton codes [Vinkler et al. 2017] are used. Moreover, distance metric evaluation is computed partly at reduced arithmetic precision as discussed in Section 3.3. The low-precision FPUs use 17-bit floats with 8-bit exponents and significands. In order to gauge the effect of reduced-precision DME, all benchmarks are also rerun with a single-precision, otherwise identical builder.

The builder is synthesized on a 28nm FDSOI CMOS technology. Operating conditions were set at 1V, temperature at 25°C, and the target clock frequency at 1 GHz. Contrary to MergeTree [Viitanen et al. 2017], which used a SRAM macro from the technology foundry, we use SRAMs with area, power and delay modeled with CACTI-P [Li et al. 2011]: this is intended to help reproducibility, and make results more comparable with previous builders [Doyle et al. 2013; Liu et al. 2015; Nah et al. 2011] that are evaluated based on CACTI estimates.

We run RTL simulations for builds of various scenes, perform power analysis based on switching activity information extracted from simulation, and record the build times, memory traffic, and estimated build energy. Fifteen test scenes are used. In order to verify the correctness of resulting trees, they are extracted from the RTL simulation as hex dumps, loaded to a software ray tracer, validated, and used to render each scene.

Tree quality is evaluated by computing SAH costs of the output trees, with node traversal cost set at 1.2 and triangle intersection cost at 1.0, as in Karras and Aila [2013]. The SAH cost is normalized to full SAH sweep. It should be noted that SAH cost is an imperfect proxy of ray tracing performance. Aila et al. [2013] have found that especially bottom-up builders tend to have worse performance than predicted by SAH. Hence, our results should not be taken as a definitive quality comparison.

The timing of the external memory interface is modeled with Ramulator [Kim et al. 2015], and the power consumption with DRAMPower [Chandrasekar et al. 2012]. Ramulator is integrated via the Systemverilog DPI to RTL simulation, and activity logs printed from it are fed to DRAMPower for analysis. We use a memory configuration with four 64-bit banks of 1333MHz LPDDR3 DRAM, which gives a theoretical peak bandwidth of 42.7 GB/s, the same as [Viitanen et al. 2017] and conveniently close to the 44 GB/s interface of [Doyle et al. 2013] for comparison. Also, we subtract the idle energy of the DRAM interface – obtained by generating an idle DRAMPower trace with only refresh commands – in order to isolate the dynamic increase in DRAM energy consumption caused by the tree build.

We compare to two state-of-the-art hardware builders: MergeTree [Viitanen et al. 2017] and the binned SAH builder by Doyle et al. [2013]. MergeTree is resynthesized with the same settings (and CACTI SRAMs) as PLOCTree, giving somewhat smaller area and higher on-chip power than the original work, and then run through the same benchmarks, using the same power estimation methodology. Moreover, we make the optimistic assumption that the output BVHs from MergeTree are post-processed with a high-level binned SAH build, at no extra runtime or energy cost. For Doyle et al. [2013], we run a software binned SAH build, and estimate memory bandwidth based on extracted binned SAH sweeps lengths as in [Viitanen et al. 2017]. For energy, only DRAM energy is considered, and it is scaled from the PLOC DRAM results after excluding refresh power, assuming linear scaling with memory traffic.

Moreover, two GPU builders are included as reference, PLOC and LBVH. The PLOC builder [Meister and Bittner 2018] is configured for similar behavior as our hardware, with radius 8, maximum leaf size 2, 32-bit Morton codes, and adaptive collapsing disabled. For LBVH, we use Domingues and Pedrini's [2015b] implementation of Karras' [2012] algorithm with 32-bit codes. However, the resulting trees are not identical to the hardware builders, due to the use of enhanced Morton codes, reduced-precision distance metric evaluation in PLOC, and an unstable hardware sort which may reorder elements with equal sort keys. GPU PLOC and LBVH are benchmarked on a Geforce GTX 1080 Ti GPU, and memory traffic is extracted with *nvprof*. We report kernel runtimes. Some kernels in each builder correspond to tasks which are not performed by the hardware units: these include Morton code computation, scene extent computation and triangle conversion to Woop's [2005] format. These kernels are removed from the bandwidth and memory results.

6 RESULTS

The main results are summarized in Table 1. The detailed results for performance, memory bandwidth usage, and tree quality are listed in Table 2, and power analysis in Table 3. In the benchmark conditions, PLOCTree consumes 1.4–1.9W of power, including DRAM traffic. To give a rough frame of reference, mobile GPUs in a recent game benchmark [Pathania et al. 2014] consume 1–3W, thus the builder appears to fit in a mobile power budget with the reported build performances. However, while the builder is in use at full performance, there is little bandwidth or power budget left for rendering. On desktop, PLOCTree would account for less than 1% the silicon area and thermal design power of a high-end GPU, and less than 10% of the maximum bandwidth, so it would be practical to constantly run it at full performance.

The closest point of comparison is MergeTree [Viitanen et al. 2017]. It is visible that PLOCTree is able to keep up with the throughput of the sorting subsystem, as it is only 7% slower than the LBVH builder. As drawbacks, PLOCTree consumes, on average, 4.5× the on-chip power and 4.8× the on-chip build energy of MergeTree, and has a 37% area overhead. A large increase in logic power is expected since MergeTree mostly performed computationally light sorting, while PLOCTree adds computationally intensive postprocessing logic. However, since in both cases the on-chip power consumption is dwarfed by DRAM interface power, and both builders have similar DRAM traffic, the effect on total build energy is comparatively small, 32%. DRAM accounts for an average of 94% of build energy for MergeTree and 79% for PLOCTree. The memory traffic of PLOCTree differs from MergeTree mainly due to leaf sizes. In large scenes, PLOCTree tends to give smaller leafs than 32-bit LBVH and, therefore, writes inner nodes, leading to 10% higher traffic on average. Moreover, some extra traffic is due to reading and writing intermediate arrays as described in Sec. 4.2.

It should be noted that mobile SoCs and desktop GPUs are beginning to switch to newer memory technologies such as LPDDR4 and HBM2. Convenient power models were not available for these technologies. LPDDR4 is according to one estimate [Lee et al. 2017] 39% more power-efficient than LPDDR3. Efficient external memory would increase the relative importance of core power.

Compared to a binned SAH builder [Doyle et al. 2013], PLOCTree gives similar SAH cost, but is ca. 5× faster, with 3× less bandwith usage, and 5× less silicon area. However, as noted in Sec. 5, SAH cost is an imperfect tree quality measure. Fig. 9 shows an example scene where practical

Table 1. Summary of results.

	GPU PLOC, GTX 1080 Ti	HW SAH [Doyle et al. 2013]	HW LBVH [Viitanen et al. 2017]	HW PLOC (**proposed**)
Process	16nm	65nm	28nm	28nm
Area (mm^2)	610[b]	68.76[d]	1.77	2.43
Scaled area	1868	12.76	1.77	2.43
Clock (GHz)	1.6	0.5	1.0	1.0
BW (GB/s)	484	44	42.7	42.7
FPU count[a]	3584[c]	1184[e]	0	256[f]
TDP (W)	250	-	0.085[g]	0.362[g]
Avg SAH	105%	104%	116%	104%
Build time	386%	550%	93%	100%
Mem. traffic	772%	278%	91%	100%
Energy	-	221%	70%	100%

[a] Number of FPU add, subtract, multiply and reciprocal units.
[b] Assuming same as Tesla P100 [Harris 2016].
[c] Assuming 1 FPU per CUDA core.
[d] 2 subtree builders á 31.88 mm^2 and upper builder 5 mm^2.
[e] Each subtree builder has 160 add, 192 sub, 224 mul, 16 reciprocal units. Upper builder not counted.
[f] $\frac{5}{8}$ of these FPUs are using low arithmetic precision.
[g] Highest on-chip power encountered in benchmarks.

performance is worse than predicted by SAH. On average, when rendering the test scenes with primary rays, PLOCtree gave 12% higher box test counts than binned SAH, compared to 31% for LBVH. The original PLOC paper [Meister and Bittner 2018] measures practical GPU ray tracing performance and finds a large improvement over LBVH. Compared to a GPU software PLOC builder, PLOCTree is ca. 4× faster and uses 8× less bandwidth. It should be noted that MergeTree gave far more modest improvements over a GPU LBVH build; ca. 3× less bandwidth and a slower build speed. Hence, the results show PLOC is highly suitable for hardware acceleration.

An area and power breakdown of the system is shown in Table 4. Here, it is visible that the PLOC sweep pipelines are compact but very power-intensive compared to the sorter. Furthermore, the iteration count has a large effect on pipeline cost: the second pipeline (S_1=32) has an over 2× larger area and power footprint than the first (S_2 = 4). The window memory of the second pipeline is the largest single power consumer in the builder, and accounts for over one-third of on-chip power.

Reduced-precision DME has the effect of saving 8.3% silicon area – a full-precision builder is 2.7mm^2 – and an average of 12.9% of on-chip build energy. However, due to large share of DRAM energy, total build energy only falls by 3%. The effect on SAH quality is on average 0.2% and ranges between -1.2% …2.5%. Extended Morton codes reduce SAH cost by 2.8%. If Morton code computation was done in hardware, this would incur the cost of a bounding-box diagonal length computation, which is inexpensive compared to the current design's 128 FPUs.

7 LIMITATIONS AND FUTURE WORK

The main limitation of the current builder compared to software PLOC is the lack of large leaf collapsing. If adaptive collapsing as in [Meister and Bittner 2018] could be implemented in hardware,

Table 2. Results and comparison: build performance, memory traffic and tree quality.

		GPU LBVH [Domingues and Pedrini 2015b]			GPU PLOC [Meister and Bittner 2018]			HW LBVH [Viitanen et al. 2017]			HW SAH [Doyle et al. 2013]			HW PLOC (**proposed**)		
Scene	tris.	ms	MB	SAH	ms	MB	SAH	ms	MB	SAH [1]	ms	MB	SAH	ms	MB	SAH
Toasters	11K	0.3	2.1	166%	3.9	2.8	97%	0.1	1.6	114%	1	1.1	103%	0.1	1.5	97%
Bunny	70K	0.4	19.8	122%	3.5	44.5	112%	0.5	10.0	116%	3	17.5	104%	0.5	9.0	111%
Elephant	85K	0.4	26.1	135%	4.3	63.0	111%	0.6	11.6	115%	-	21.8	104%	0.6	11.0	109%
Cloth	92K	0.4	29.3	123%	4.0	73.9	109%	0.7	13.3	112%	-	24.6	103%	0.7	12.2	108%
Fairy	174K	0.6	58.4	129%	7.0	168.1	103%	1.1	17.9	105%	-	70.1	102%	1.3	22.7	103%
Armadillo	213K	0.7	73.0	128%	6.0	220.4	111%	1.4	26.8	116%	-	68.8	103%	1.4	27.9	109%
Crytek	262K	0.8	95.7	151%	7.2	260.8	95%	1.7	28.5	101%	-	115.5	107%	1.9	33.9	94%
Conference	283K	0.9	108.2	165%	7.0	276.3	87%	1.9	28.0	103%	11	124.9	110%	2.1	36.3	86%
Sportscar	301K	0.9	114.1	150%	7.8	350.7	107%	2.0	35.5	118%	-	110.5	103%	2.1	39.5	107%
Italian	374K	1.1	144.0	214%	7.5	426.5	97%	2.5	40.8	123%	-	145.3	105%	2.6	48.9	97%
Babylonian	500K	1.4	196.9	208%	8.4	572.3	98%	3.3	55.3	129%	-	219.0	104%	3.5	64.8	99%
Hand	655K	1.7	263.8	130%	8.6	749.9	114%	4.1	73.4	124%	-	271.8	104%	4.4	85.9	112%
Dragon	871K	2.3	360.4	136%	13.6	1287.7	112%	5.5	99.9	123%	30	379.3	102%	6.0	118.8	112%
Buddha	1087K	2.8	456.9	133%	15.1	1578.8	111%	7.0	122.8	124%	-	492.9	102%	7.7	148.7	111%
Lion	1604K	4.0	683.4	144%	18.8	2125.6	107%	10.4	174.0	122%	-	800.3	103%	11.3	212.6	107%
Average		**61%**	271%	149%	386%	772%	105%	93%	**91%**	116%	550%[2]	278%	**104%**	100%	100%	104%

[1] SAH assuming a top-level binned SAH build to improve quality (not included in runtime or memory traffic).
[2] *Toasters* omitted from the average due to low precision.

Table 3. Power analysis results and comparison.

		HW LBVH [Viitanen et al. 2017]					HW PLOC (**proposed**)					HW SAH [Doyle et al. 2013]
		Power (mW)		Energy (mJ)			Power (mW)		Energy (mJ)			Energy (mJ)
Scene	tris.	Logic	DRAM	Logic	DRAM	Total	Logic	DRAM	Logic	DRAM	Total	DRAM
Toasters	11K	79.6	1326.2	0.0	0.1	0.1	347.9	1073.3	0.0	0.1	0.1	0.1
Bunny	70K	85.4	1486.1	0.0	0.7	0.8	348.7	1376.2	0.2	0.7	0.8	1.3
Elephant	85K	80.6	1414.2	0.0	0.8	0.9	354.0	1343.9	0.2	0.8	1.0	1.6
Cloth	92K	84.8	1547.2	0.1	1.0	1.1	348.0	1465.9	0.2	1.0	1.2	1.9
Fairy	174K	80.7	1114.5	0.1	1.3	1.4	355.9	1253.0	0.5	1.6	2.1	5.0
Armadillo	213K	82.4	1573.1	0.1	2.2	2.3	355.8	1566.2	0.5	2.3	2.8	5.6
Crytek	262K	80.6	1176.1	0.1	2.0	2.2	351.6	1280.7	0.7	2.4	3.1	8.2
Conference	283K	78.6	1044.0	0.1	2.0	2.1	342.4	1212.9	0.7	2.6	3.3	8.8
Sportscar	301K	79.1	1267.4	0.2	2.6	2.7	361.7	1327.7	0.8	2.8	3.6	7.9
Italian	374K	75.5	1213.8	0.2	3.0	3.2	348.1	1334.0	0.9	3.5	4.4	10.5
Babylonian	500K	75.4	1235.8	0.2	4.1	4.3	344.1	1341.6	1.2	4.7	5.9	15.8
Hand	655K	74.8	1322.2	0.3	5.4	5.7	342.5	1418.4	1.5	6.2	7.7	19.7
Dragon	871K	74.6	1439.2	0.4	7.9	8.3	356.3	1516.2	2.1	9.1	11.2	28.9
Buddha	1087K	74.5	1387.7	0.5	9.7	10.2	355.1	1479.0	2.7	11.3	14.0	37.5
Lion	1604K	73.3	1188.4	0.8	12.4	13.2	354.6	1320.4	4.0	14.9	18.9	56.2
Average		100%	100%	100%	100%	100%	447%	104%	481%	111%	132%	298%[1]

[1] Relative to LBVH *total* build energy.

Table 4. Area and power breakdown. Note: Due to preserving hierarchy in resynthesis, area and power is significantly higher than in the main results.

Component	Area (mm^2)		Power (mW)	
Sorting subsystem	1.65	(65%)	86	(18%)
FIFOs, muxes	0.43	(17%)	6	(1%)
Insertion sorters	0.24	(9%)	27	(6%)
Multimerge unit	0.97	(38%)	52	(11%)
PLOC pipe 1	0.24	(10%)	110	(23%)
Window memory	0.09	(3%)	26	(6%)
Distance metric evaluators	0.10	(4%)	28	(6%)
Comparator tree, delay pipe	0.02	(1%)	35	(7%)
Postprocessing	0.04	(1%)	19	(4%)
PLOC pipe 2	0.51	(20%)	270	(57%)
Window memory	0.25	(10%)	173	(37%)
Distance metric evaluators	0.10	(4%)	27	(6%)
Comparator tree, delay pipe	0.02	(1%)	35	(7%)
Postprocessing	0.13	(5%)	32	(7%)
Misc. toplevel logic	0.12	(5%)	4	(1%)
Total	2.53	(100%)	471	(100%)

our initial software experiments indicate that the builder could give a ca. 5% better quality as measured with SAH cost. We are working on a hardware architecture to allow streaming adaptive collapsing. Other limitations include that the present hardware has a hardwired search radius $R = 8$ and a Morton code bitwidth of 32, which may be too inaccurate for large scenes. Moreover, the sorter configuration only handles scenes of up to 2M primitives. It would be particularly expensive to increase R since this requires replicating nearly every hardware resource in the builder. However, in the original PLOC paper [Meister and Bittner 2018], R only had a minor effect on tree quality.

Several opportunities remain for optimization. For example, the current architecture evaluates each distance metric twice – it may be more efficient to only compute distances to previous R AABBs, though this adds complexity to the postprocessing stage. Moreover, the PLOC sweep pipelines are idle when the sorter is running a block sort, so it may be interesting to decouple the block sorts from the merge logic of the sorting subsystem. The build speed of PLOCTree can be scaled up by adding pipelines, but we expect that the sorting subsystem would then become the bottleneck, and need to be replaced with a faster sorter, e.g., in the vein of Casper and Olukotun [2014]. Instead of the current pipeline-style parallelization, it may be preferable to run multiple elements of the same PLOC sweep in parallel, as this shares window memory hardware between the parallel resources, and may give better load balancing.

8 CONCLUSION

This paper proposed PLOCTree, a fast, high-quality hardware BVH tree builder for real-time ray tracing, based on the PLOC algorithm by Meister et al. [Meister and Bittner 2018]. The proposed builder has a better performance-quality tradeoff relative to previous work, giving a large tree quality improvement over a hardware LBVH build [Viitanen et al. 2017] with a modest runtime and energy overhead. The resulting trees have SAH costs similar to a binned SAH builder [Doyle et al. 2013], though practical ray tracing performance may be lower [Aila et al. 2013]. Moreover, PLOCTree gives a significant speedup and reduction in memory bandwidth usage compared to a software implementation of the same algorithm running on a high-end desktop GPU.

The proposed builder might be used as a component of either a desktop or mobile ray tracing GPU to good effect. On desktop, it would take up a small fraction of the silicon area and TDP of a high-end GPU, and free up the computational resources of the GPU for other tasks, while speeding up tree builds by a factor of 3.9. The power footprint of the builder is small enough to fit in a mobile platform. This work demonstrates that an algorithm from the recent crop of fast, high-quality builders, based on Morton code sorting and postprocessing, can be succesfully adapted to hardware.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their insightful comments, and the TUT Ultra Video group for the use of a high-end GPU. We gratefully acknowledge financial support by the Tampere University of Technology Graduate School, Business Finland (FiDiPro Program funding decision 40142/14), HSA Foundation, and the Academy of Finland (funding decision 297548).

REFERENCES

Timo Aila, Tero Karras, and Samuli Laine. 2013. On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics*. ACM, 101–107.

Ciprian Apetrei. 2014. Fast and simple agglomerative LBVH construction. In *Proc. Computer Graphics and Visual Computing*.

Jirí Bittner and Daniel Meister. 2015. T-SAH: Animation optimized bounding volume hierarchies. In *Computer Graphics Forum*, Vol. 34. 527–536.

Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proc. ACM/SIGDA Int. Symp. Field-programmable gate arrays*. 151–160.

Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. DRAMPower: Open-source DRAM power & energy estimation tool. Retrieved Feb 30, 2017 from http://www.drampower.info

Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. 2017. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *Comput. Surveys* 50, 4 (2017), 58.

Leonardo Domingues and Helio Pedrini. 2015a. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proc. High-Performance Graphics*. 13–20.

Leonardo Domingues and Helio Pedrini. 2015b. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proc. High-Performance Graphics*. 13–20.

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.

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).

Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and faster HLBVH with work queues. In *Proc. High-Performance Graphics*. 59–64.

Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH construction via approximate agglomerative clustering. In *Proc. High-Performance Graphics*. 81–88.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 37–47.

Mark Harris. 2016. Inside Pascal: NVIDIA's newest computing platform. Retrieved April 9, 2018 from https://devblogs.nvidia.com/inside-pascal/

Thiago Ize, Ingo Wald, and Steven G Parker. 2007. Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proc. Eurographics Conf. Parallel Graphics and Visualization*. 101–108.

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

Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proc. High-Performance Graphics*. 89–99.

Sean Keely. 2014. Reduced precision hardware for ray tracing. In *Proc. High-Performance Graphics*. 29–40.

Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters* PP, 99 (2015), 1–1.

Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler. 2012. Fast, effective BVH updates for animated scenes. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games*. 197–204.

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.

Sukhan Lee, Yuhwan Ro, Young Hoon Son, Hyunyoon Cho, Nam Sung Kim, and Jung Ho Ahn. 2017. Understanding power-performance relationship of energy-efficient modern DRAM devices. In *Proc. IEEE Int. Symp. Workload Characterization*. 110–111.

Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*. 694–701.

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.

J David MacDonald and Kellogg S Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 3 (1990), 153–166.

James McCombe. 2014. New Techniques Made Possible by PowerVR Ray Tracing Hardware. GDC Talk.

Morgan McGuire, Petrik Clarberg, and Nir Benty. 2018. The ray + raster era begins - an R&D roadmap for the game industry. (2018). Game Developers Conference Talk.

Daniel Meister and Jiří Bittner. 2018. Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics* 24, 3 (2018), 1345–1353.

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.

Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. 2014. RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics* 33, 5 (2014), 162:1–15.

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 (Dec. 2011), 160:1–10.

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.

Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. 2014. Integrated CPU-GPU power management for 3D mobile games. In *Proc. Design Automation Conf.* 1–6.

Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. 2017. Dual streaming for hardware-accelerated ray tracing. In *Proc. High Performance Graphics*. 12.

Karthik Vaidyanathan, Tomas Akenine-Möller, and Marco Salvi. 2016. Watertight ray traversal with reduced precision. *Proc. High-Performance Graphics* (2016).

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2017. MergeTree: A fast hardware HLBVH constructor for animated ray tracing. *ACM Transactions on Graphics* 36, 5 (2017), 169.

Marek Vinkler, Jiri Bittner, and Vlastimil Havran. 2017. Extended Morton codes for high performance bounding volume hierarchy construction. In *Proc. High-Performance Graphics*. 9.

Ingo Wald. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE Symp. Interactive Ray Tracing*. 33–40.

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.

Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. 2008. Fast agglomerative clustering for rendering. In *Proc. IEEE Symp. Interactive Ray Tracing*. 81–86.

Sven Woop, Erik Brunvand, and Philipp Slusallek. 2006. Estimating performance of a ray-tracing ASIC design. In *Proc. IEEE Symp. Interactive Ray Tracing*. 7–14.

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.