Eurographics Symposium on Rendering 2017
P. Sander and M. Zwicker
(Guest Editors)

Volume 36 (2017), Number 4

Fast Hardware Construction and Refitting of Quantized Bounding Volume Hierarchies

paper1037

Abstract

There is recent interest in GPU architectures designed to accelerate ray tracing, especially on mobile systems with limited memory bandwidth. A promising recent approach is to store and traverse Bounding Volume Hierarchies (BVHs) [KK86], used to accelerate ray tracing, in low arithmetic precision. However, so far there is no research on refitting or construction of such compressed BVHs, which is necessary for any scenes with dynamic content. We find that in a hardware-accelerated tree update, significant memory traffic and runtime savings are available from streaming, bottom-up compression. Novel algorithmic techniques and hardware implementations are proposed to reduce backtracking inherent in bottom-up compression: modulo encoding, treelet-based compression and minimum scale adjustment. Together, these techniques reduce backtracking to a small fraction. Compared to a separate top-down compression pass, streaming bottom-up compression with the proposed optimizations saves on average 42% of memory accesses for LBVH construction and 56% for refitting of compressed BVHs, over 16 test scenes. In architectural simulation, the proposed streaming compression reduces LBVH runtime by 20% compared to a single-precision build, and 41% compared to a single-precision build followed by top-down compression. Since memory traffic dominates the energy cost of refitting and LBVH construction, energy consumption is expected to fall by a similar fraction.

CCS Concepts
•***Computing methodologies*** $\rightarrow$ *Ray tracing; Graphics processors;*

1. Introduction

In the last decade, ray tracing GPU architectures have been the subject of intensive research and industrial interest. Ray tracing accelerators have been proposed based on fixed-function pipelines [LSL*13, NKK*14, WSS05, LV16], programmable MIMD processors [SKKB09, SKBD12], and augmenting conventional GPUs [Kee14]. There has been a special focus on architectures targeting mobile systems [LSL*13, NKK*14, SKBD12, Pow15], where they might, e.g., enable photorealistic augmented reality applications [LSL*13]. High-performance ray tracing is based on indexing the scene geometry in an acceleration datastructure, typically a *Bounding Volume Hierarchy* (BVH), which speeds up ray-scene collision queries. Recently, a potential breakthrough in ray tracing GPU architecture is to store and traverse BVHs at low arithmetic precision, e.g., 5-6 bits per coordinate. We refer to these structures as *Compressed BVHs* (CBVH). Keely [Kee14] estimates that the use of CBVHs – combined with compact leaf storage and treelet scheduling – reduces the arithmetic energy cost of ray traversal by 23x, and memory traffic by 6–22x.

However, there is no research yet on updating CBVHs in real time to match animated scenes. Many rendering applications include dynamic scene content, and a key advantage of plain BVHs has been the ability to rapidly construct new BVHs, or to *refit* an existing BVH to match deformed geometry [WBB08]. It is interesting whether this advantage is preserved in CBVH. Fast uncompressed BVH update algorithms have been studied extensively for GPUs [Wal07, LGS*09, KIS*12], and hardware accelerators have been proposed [DFM13, VKJ*15, NKP*15]. The reduced cost of rendering in CBVHs will, through Amdahl's law, increase the relative share of tree updates, and make their performance more critical to the system. Meanwhile, compression introduces new complications to tree updates.

Tree updates are highly memory-intensive, and especially in the context of mobile systems and custom hardware pipelines, the amount of external memory traffic they generate is a major factor in their performance and energy efficiency [DFM13, VKJ*15]. Figure 1 shows the basic motivation and goal of this paper. A straightforward way to update a CBVH is to first produce a full-precision BVH tree and then subsequently compress it. We would like to, instead, directly output a CBVH in a *streaming* manner, saving up to ca. 50% of memory accesses for construction and 64% for refitting. Streaming compression is straightforward with a top-down algorithm, but the fastest approaches to tree update, *refitting* [WBS07] and *Linear BVH* (LBVH) [LGS*09], output BVHs in bottom-up or-

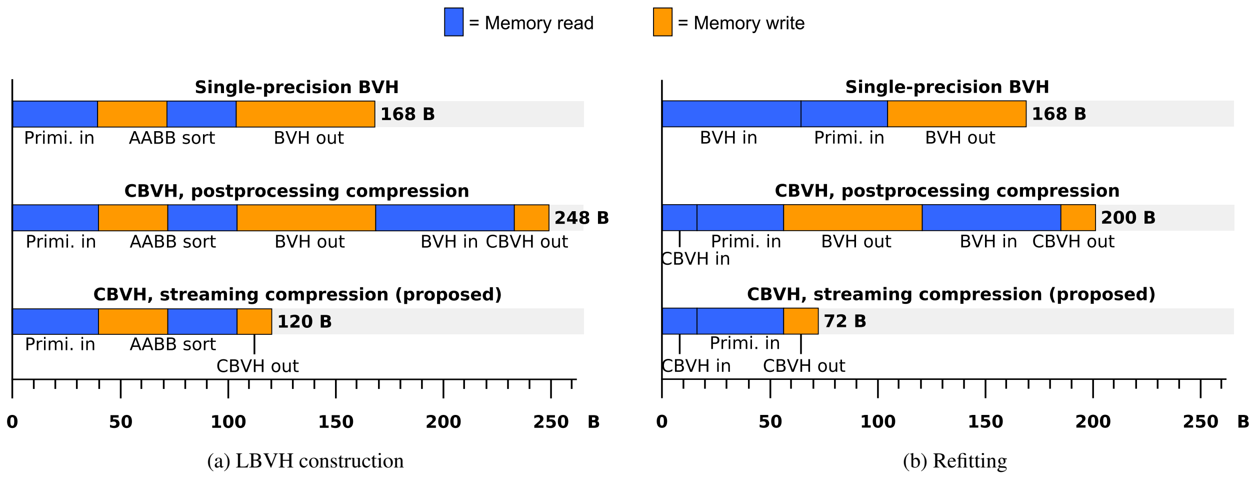


Figure 1: *External memory traffic of different hardware tree update strategies, bytes per input primitive; assuming LBVH unit similar to [VKJ*15] and trees with one primitive per leaf. Updating or refitting to single-precision BVH and postprocessing into CBVH has a large overhead compared to direct CBVH output. (Up to ca. 2x more traffic for LBVH and 2.6x for refit.)*

der. Bottom-up compression, in turn, is nontrivial since each CBVH node is encoded with reference to its parent. Consequently, in order to emit a node, we need to make assumptions about its parent, and backtrack to repair the hierarchy when said assumptions are falsified. The memory traffic from backtracking can eclipse the traffic saved by streaming compression.

In this article, we investigate the bottom-up streaming compression of CBVHs, and especially techniques to reduce backtracking. The main contributions of this paper are as follows:

- A novel *modulo encoding* for CBVH coordinates which reduces the backtracking overhead by ca. $\frac{2}{3}$.
- A treelet-based bottom-up compression algorithm, which eliminates roughly a further $\frac{2}{3}$ of backtracking per level of treelet depth used.
- An analysis of minimum scale adjustment to eliminate backtracking due to sections of geometry with a degenerate axis.
- Hardware architectures which implement the above algorithmic techniques.

Together, these techniques reduce backtracking to a small fraction, so that streaming compression gives close to ideal traffic savings.

This paper is organized as follows. Section 2 discusses related work. In Section 3, we describe the baseline CBVH encoding. Section 4 discusses the proposed approach of bottom-up streaming compressions, and proposes optimizations to reduce backtracking memory traffic inherent in the approach. Section 5 gives an evaluation of the proposed techniques, and Section 6 concludes the paper.

2. Related work

BVH compression with quantized coordinates has been investigated in software by Mahovsky [MW06] and later Segovia [SE10] for the purpose of out-of-core ray tracing of very large static scenes. Keely [Kee14] proposed a hardware architecture based on augmenting a conventional GPU with CBVH traversal hardware. A *traversal point update* method was proposed to traverse CBVHs with low-precision arithmetic computations, allowing traversal with compact hardware accelerators. Vaidyanathan et al. [VAMS16] approached CBVHs from a more formal framework and gave an alternate, provably watertight traversal algorithm, while reusing AABB coordinates from parent nodes as in [FD09] to further shrink the memory footprint and arithmetic cost of CBVHs.

The quality of BVH trees is typically measured with the *Surface Area Heuristic* (SAH) [GS87]. The classical build methods of SAH sweep and binned SAH sweep [Wal07] recursively partition the scene primitives guided by SAH values of candidate splits. Most tree construction methods aiming for fast build speed are based on the *Linear BVH* (LBVH) approach of Lauterbach et al. [LGS*09], which has been optimized by several authors [PL10, GPM11, Kar12, Ape14]. LBVH is a fast, low-quality build algorithm, and state-of-the-art builders further process the resulting tree to improve quality; e.g., the *Agglomerative Treelet Restructuring BVH* algorithm (ATRBVH) algorithm [DP15] rearranges an LBVH tree to give better tree quality than SAH sweep at a fraction of the runtime.

Another broad approach to tree update is to refit an existing tree to new geometry [WBS07]. This is a faster operation than full rebuild, but restricted to animations that conserve mesh topology. Tree quality tends to deteriorate over many refits, so it is often periodically refreshed with an asynchronous high-quality rebuild [IWP07], or maintained with tree rotations during the refit [KIS*12].

Hardware accelerators have been proposed to update trees especially on mobile platforms, where energy and memory bandwidth constraints prevent the use of GPGPU algorithms. Nah et al. [NKP*15] propose a scheme similar to [IWP07] based on refitting and asynchronous rebuilds, however, refitting is accelerated with hardware *Geometry and Tree Update* (GTU) units. Doyle et al. [DFM13] proposed a hardware BVH builder based on the binned SAH sweep algorithm, which optimizes memory traffic by performing stages of the algorithm in a pipelined manner. Viitanen et al. [VKJ*15] have proposed an LBVH builder which gives a similar tradeoff as desktop LBVH compared to [DFM13]: the build is much faster, at the cost of reduced tree quality, which needs to be recovered with postprocessing.

In this paper, we describe how to augment LBVH and refitting hardware such as [VKJ*15, NKP*15] to emit CBVHs in a streaming manner. To our best knowledge, this is the first study on CBVH construction and refitting. The present work is focused on hardware implementation, but may also be applicable to GPGPU tree updates.

3. Background

We start from a formal description of quantized trees similar to the one given by Vaidyanathan et al. [VAMS16]. In a full precision BVH, each node is represented by an *Axis-Aligned Bounding Box* (AABB) consisting of a lower bound p and a upper bound q, where $(p,q) \in \mathbb{R}^3$. We write the components of vectors as, e.g., p_x, p_y, p_z. In CBVH, the bounds are quantized to a low resolution grid aligned with the parent AABB, represented as low-precision (e.g. 5–6 bit) integers, and decompressed during traversal. This results in a lossy compression where the AABB's memory footprint is sharply reduced, at the cost of some extra node tests due to enlarged bounds. If the quantized grids are zero-aligned, the local grid coordinates (r_i, s_i) for each axis i can be computed as

$$r_i = \left\lfloor \frac{p_i - u_i^{parent}}{2^{e_i^{\text{parent}}}} \right\rfloor; \quad s_i = \left\lfloor \frac{q_i - u_i^{parent}}{2^{e_i^{\text{parent}}}} \right\rfloor + 1, \tag{1}$$

where e is the local grid scale exponent, and u^{parent} is the quantized parent lower bound. The per-axis grid scale exponents e_i of a node are chosen to be the minimum where the node fits in 2^N grid intervals, so that its r, s can be expressed in N bits, i.e.,

$$e_i = \arg\min_k \left((v_i - u_i)/2^k \leq 2^N \right). \tag{2}$$

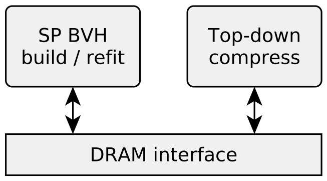
When traversing the tree, the decompressed bounds (u, v) can be computed as

$$u_i = u_i^{\text{parent}} + r_i 2^{e_i^{\text{parent}}}; \qquad v_i = u_i^{\text{parent}} + s_i 2^{e_i^{\text{parent}}}. \tag{3}$$

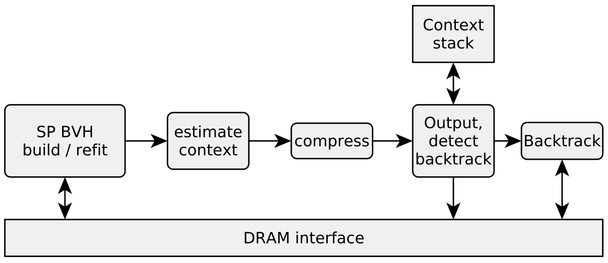
The original bounds (p,q) are enclosed in the decompressed bounds, such that $u \leq p \leq q \leq v$. Note that in order to encode or traverse a CBVH node with Equations 1 and 3, we need values for the parent scale and lower bound $e^{\text{parent}}, u^{\text{parent}}$. We refer to these values as a *traversal context*.

Methods have been proposed to traverse CBVHs at a low arithmetic precision instead of unpacking each node and using single-precision collision tests [Kee14, VAMS16]. In single-precision traversal, the input ray is tested against each AABB with the slabs test [KK86], which first computes parametric distances to each plane,

$$t_{u\_i} = (p_i - o_i)d_i^{-1}; \quad t_{v\_i} = (q_i - o_i)d_i^{-1}, \tag{4}$$



(a) Postprocessing, top-down compression HW



(b) Streaming, bottom-up compression HW (proposed)

Figure 2: *Hardware organizations for two CBVH compression strategies.*

where o, d are the ray origin and direction, and t_{u_i} and t_{v_i} are the parametric distances to the lower and upper bound planes on axis i, respectively. The distances are then combined with *min* and *max* operations to form parametric near and far distances (t_{near}, t_{far}) to the AABB. If $t_{far} < t_{near}$, the ray does not intersect the AABB. The most recent low-precision approach [VAMS16] instead computes plane distances incrementally during traversal:

$$t_{lb\_i} = t_{lb\_i}^{\text{parent}} + r2^{e_i^{\text{parent}}} d_i^{-1}; \quad t_{ub\_i} = t_{ub\_i}^{\text{parent}} - s' 2^{e_i^{\text{parent}}} d_i^{-1}. \tag{5}$$

As a result, single-precision multiplications can be replaced with cheaper 24×6 bit multiplications. In order to guarantee watertight traversal, the quantized offset s' is computed from the parent upper bound rather than the lower bound, and rounding modes are selected to maximize t_{far} and minimize t_{near}.

4. Algorithm

In this section, we describe the basic idea of streaming bottom-up CBVH update, followed by techniques to reduce the backtracking memory traffic inherent in the approach.

4.1. Bottom-up construction and backtracking

Top-down compression of a quantized tree is straightforward by evaluating Equation 1. In bottom-up compression, however, a traversal context is unavailable, and has to be estimated. A stream of BVH node pairs in depth-first, bottom-up order can then be compressed into a CBVH as in Algorithm 1. A traversal context is estimated for each processed BVH node and placed on a *traversal context stack*. When processing an inner node, the contexts of its children can be found at the top of the stack. By decoding the newly generated node and comparing the produced child traversal contexts to the stored contexts, we may detect whether a child was invalidated by the new node, i.e., whether the context estimated when encoding the child was incorrect. The algorithm then recursively backtracks to repair the invalidated children (function `Backtrack` in Algorithm 1).

Context estimation (function `EstimateContext`) is straightforward for e^{parent} by examining the exponent of $q-p$. For u^{parent} it is difficult with the formulation of [VAMS16] where the toplevel grid is aligned to the uncompressed lower bound of the scene, as this is unknown until the end of bottom-up traversal. We instead align all grids to zero, i.e.,

$$u_i = k2^{e_i^{parent}} \quad ; \quad v_i = l2^{e_i^{parent}} \; k,l \in \mathbb{Z}. \tag{6}$$

Each backtracking iteration decodes the target compressed node with its original context and recodes the single-precision bounds with a new context. Child contexts are then checked to determine whether to continue recursion further.

The approach so far is able to produce correct trees. However, backtracking may cause enough memory traffic to undo the savings from streaming compression. It should be noted that backtracking is more expensive than the original encoding, and makes random accesses to the memory, while the main streaming compression algorithm has a more efficient linear access pattern. In our initial experiments on bottom-up update, backtracking almost completely eliminates the memory traffic savings from streaming compression. Hence, it is interesting to minimize backtracking. We next investigate approaches to reduce backtracking.

4.2. Modulo encoding

When applying the above compression scheme to test scenes, we note that child nodes are often invalidated even though their decompressed bounds are unchanged. We examine a typical case in Fig. 3, in which a node A is combined with a primitive to form a new node B. When encoding A, a scale $s=1$ is used. When encoding B, the bounds of A are snapped to a grid with scale $s=2$ and rounded. Though the child coordinates encoded by A still refer to the same coordinates, they are relative to the node bounding box derived from B, which has shifted. We find that in real scenes a majority of backtracking is due to this type of context mismatch. It is, then, interesting to find an encoding where the low-precision coordinates can represent the same points as the relative coordinates, but are robust to small changes in parent bounds.

We describe here a novel encoding which has this desired property, and follows from the global grids described earlier. Given that

$$u_i = k2^{e_i^{parent}} \quad ; \qquad v_i = l2^{e_i^{parent}} \; k,l \in \mathbb{Z}, \tag{7}$$

we can replace (r,s) used in Equations 1 and 3 with modulo coordinates $(\hat{r},\hat{s})$ such that

$$\hat{r} = k \mod 2^N; \qquad \hat{s} = l \mod 2^N, \tag{8}$$

where mod is the integer modulo operation, i.e., $a = b\lfloor a/b \rfloor + (a \mod b)$. To traverse a modulo encoded tree, given the parent's lower-bound modulo coordinate in the local grid $\hat{r}^{parent}$, we can

Algorithm 1: Bottom-up streaming CBVH compression algorithm

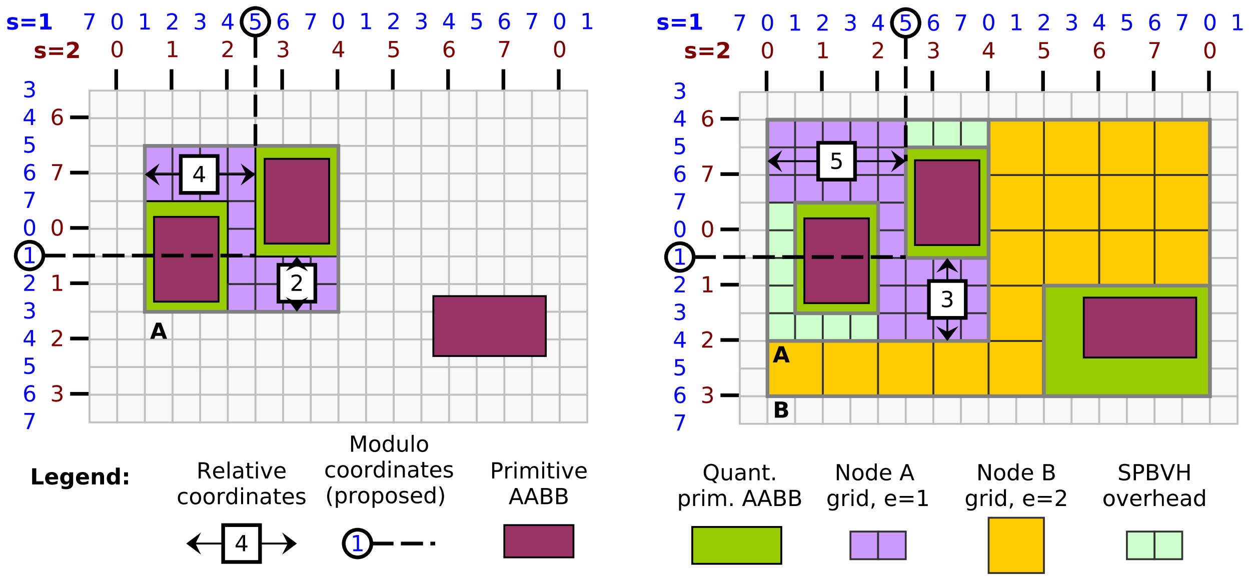
```
Data: nodes
Data: nodeCount
Data: contextStack
Function Backtrack (ptr, oldContext, newContext) is
    oldChildContexts[]
        ← Decode ( nodes[idx], oldContext ) ;
    nodes[idx], newChildContexts[]
        ← Encode ( oldChildContexts[0].aabb,
                   oldChildContexts[1].aabb,
                   newContext) ;
    foreach $i \in 1,2$ do
        if oldChildContexts[i].scale $\neq$
         newChildContexts[i].scale then
            Backtrack (nodes[idx].ptr[i],
            oldChildContexts[i], newChildContexts[i] ) ;
        end
    end
end
Function BUCompress(bvhNode) is
    context ← EstimateContext (bvhNode) ;
    nodes[nodeCount], childContexts[]
        ← Encode ( bvhNode.aabbs[0],
                   oldChildContexts[1].aabbs[1],
                   context) ;
    nodeCount ← nodeCount + 1 ;
    foreach child $\in 2,1$ do
        if child is a leaf then
            storedContext ← contextStack.Pop() ;
            if childContexts[child].scale $\neq$ storedContext.scale
             then
                Backtrack (bvhNode.ptr[child],
                 storedContext, childContexts[child]) ;
            end
        end
    end
    contextStack.Push(context) ;
end
```

recover relative coordinates as

$$r = (\hat{r} - \hat{r}^{parent}) \mod 2^N;$$
$$s = \begin{cases} (\hat{s} - \hat{s}^{parent}) \mod 2^N & \text{if } \hat{r} \neq \hat{s}, \\ (\hat{s} - \hat{s}^{parent}) \mod 2^N + 2^N & \text{if } \hat{r} = \hat{s}. \end{cases} \tag{9}$$

Note that parent lower bound modulo coordinates r^{parent} are needed for traversal, i.e., they are added to the traversal context. As with relative encoding, as long as the parent bounds limit a range of up to 2^N gridcells in the local scale, modulo coordinates can encode any child range. However, the modulo encoding *is robust to changes in parent bounds as long as the parent scale is unchanged.*

C pseudocode for traversal is shown in Fig. 4. Note that the parent modulo coordinate needs to be scaled to the local grid before use (line 1). In the code, floating-point coordinates are recovered



(a) Inputs of a bottom-up merge

(b) Result of merge

Figure 3: *Example of a bottom-up CBVH merge of a primitive with a 2-primitive node. 3 bit coordinates are used, i.e., a node can be up to 8 internal gridcells wide. After snapping the child node bounds to the new parent's grid, its relative coordinates are invalidated. The proposed modulo coordinates are more robust and only invalidated by a parent scale change. Note that unlike full-precision BVHs, plane reuse from the parent box changes the resulting bounds.*

(lines 7–8), but we could instead use the relative coordinates directly for ray tracing as in Vaidyanathan's [VAMS16] work. In this case, the upper bound coordinate relative to parent upper bound, s', is recovered as

$$s' = (\hat{s}^{parent} - \hat{s}) \mod 2^N, \qquad (10)$$

and the parent upper bound s_{parent} needs to be added to the traversal context. The grid scale is saturated close to the minimum representable floating point number (line 17).

We next discuss a hardware-friendly way to compress a given floating point node pair to modulo encoding. C pseudocode for compression, along one axis, is shown in Fig. 5. First, the input floating point upper and lower bounds are broken into sign, mantissa and exponent (lines 1–5). They are then aligned to local grid scale, rounding the lower bound down and upper bound up (lines 7–8). Next the mantissas are converted from sign-magnitude representation to two's complement (lines 10–12). The low mantissa bits now correspond to the quantized modulo coordinates $\hat{r}, \hat{s}$ and are extracted (lines 16–18). Note that the input scale must be selected such that at this point `ub_m` $-$ `lb_m` $\leq 2^N$. We now have enough information to store the compressed node in memory. In a bottom-up build, we next need to decide whether to backtrack, and compute a traversal context for backtracking. Backtracking detection is based on the scale, while the context additionally has a quantized child AABB (in single precision), and a lower-bound coordinate in the child grid, i.e., lb_mod scaled to the child grid. These are computed next (lines 20–39).

4.2.1. Hardware complexity

The overhead of modulo encoding in traversal consists of low-precision arithmetic. Lines 3, 4, 10 in Figure 4 represent N-bit adders, while lines 13–17 can be implemented with a priority encoder and a low-precision subtractor. The child axis computation logic of lines 10–17 is also present in relative coordinate traversal. In total, the overhead is equal to ca. 2 N-bit adders and 1 shifter per AABB axis, or 12 adders and 6 shifters for a node pair. Using the component costs in [VAMS16], the overhead of the adders is 5% in addition to a *traversal point update* traversal unit, or 13% against a shared-plane traversal unit.

A hardware implementation of a single-axis compressor shares much of its structure with a floating-point adder. Figure 6 contrasts a compressor to the significand datapath of an adder. The main components of an adder are input alignment, significand addition/subtraction, normalization and rounding. In Fig. 5, we align two inputs like the adder equivalent, apply rounding, and normalize back to IEEE-754 single-precision. The cost of conversions to and from two's complement representation is similar to that of rounding. In summary, a single-axis compressor has roughly 2x the input alignment, 2x the normalization, and 6x the rounding logic of a single-precision adder. We can approximate it as the equivalent of two adders, as significand addition is more expensive than rounding. A BVH node pair compressor has six single-axis compressors and, in a bottom-up build, scale estimators for each axis, which are the equivalent of single-precision subtractors, for a total cost of 15 single-precision adders. This appears a reasonable cost considering that, e.g., the tree builder in [DFM13] has over 500 FPUs.

```
parent_lb_mod <<= (parent_scale - scale);

int rel_lb = (lb_mod - parent_lb_mod) &
    ((1<<QUANT_BITS)-1);
int rel_ub = (ub_mod - parent_lb_mod) &
    ((1<<QUANT_BITS)-1);
if(rel_ub==0) rel_ub += (1<<QUANT_BITS);

clb_out = parent_clb + glm::ldexp(float(
    rel_lb), scale);
cub_out = parent_clb + glm::ldexp(float(
    rel_ub), scale);

int gridsize = rel_ub - rel_lb;
child_scale_out = scale;
while(gridsize < (1<<(QUANT_BITS-1))) {
    gridsize <<= 1;
    child_scale_out--;
}
if(child_scale_out < -126)
    child_scale_out = -126;
```

Figure 4: *Modulo-encoded CBVH traversal along one axis.* `clb_out`, `cub_out`*: Decompressed lower and upper bound* (u_i,v_i). `lb_mod`, `ub_mod`*: Modulo coordinates of lower and upper bound* $(\hat{r}_i,\hat{s}_i)$. `parent_lb_mod`*: Node lower bound modulo coordinate, in parent scale* $(\hat{r}_i^{parent})$. `parent_scale`, `scale`, `child_scale_out`*: scale* (e_i) *of parent, current and child node.*

4.3. Treelet-based compression

We find from experiments with real scenes that backtracking typically only descends one or two hierarchy levels before terminating. Fig. 7 shows the depth histogram of backtracking iterations in a test scene: it is visible that the distribution is top-heavy. Since the working set for possible, e.g., 1- or 2-level backtracks is easily stored on chip, there are many possible ways to avoid the memory traffic from these short backtracks. We describe one approach here which eliminates backtracking down to a fixed depth while always retaining fully pipelined throughput.

In the proposed approach, we place logic after the compression and update unit to collect BVH nodes into small treelets with a fixed maximum depth M. Each treelet is then compressed top-down. Context estimation is done only at the treelet root node, while only the nodes at level M are output. This is almost equivalent to preemptively backtracking all treelets, except there is no need to decompress nodes, simplifying the hardware. A hardware architecture with a depth of 3 is shown in Figure 8. An M-level treelet has up to M^2-1 nodes, so the cost of this approach grows quickly, but it is evident from Figure 7 that removing backtracks of even 1–2 levels gives most of the benefits available. Treelet collection is performed by similar stack-based hardware as backtrack detection: each stack element has space for a treelet of depth $M-1$. Plane sharing [FD09] may be used to compress the treelets and reduce chip area, as the compression and decompression is very cheap in fixed-function hardware.

```
//inputs -> sign, exponent, mantissa
int lb_s, lb_e, lb_m;
int ub_s, ub_e, ub_m;
break_float(lb, lb_s, lb_e, lb_m);
break_float(ub, ub_s, ub_e, ub_m);

alignf(lb_e, lb_m, lb_s, scale, RND_DOWN);
alignf(ub_e, ub_m, ub_s, scale, RND_DOWN);

//sign-magnitude -> 2's complement
if(lb_s) lb_m = -lb_m;
if(ub_s) ub_m = -ub_m;

ub_m++;

// Output modulo coordinates
lb_mod_out = lb_m & ((1 << QUANT_BITS)-1);
ub_mod_out = ub_m & ((1 << QUANT_BITS)-1);

// Output child scale
int gridsize = ub_m - lb_m;
child_scale_out = scale;
while(gridsize < (1<<(QUANT_BITS-1))) {
    gridsize <<= 1;
    child_scale_out--;
}
if(child_scale_out < -126)
    child_scale_out = -126;

// Decode bounds
lb_s = (lb_m < 0) ? 1 : 0;
if(lb_s < 0)
    lb_m = -lb_m;
ub_s = (ub_m < 0) ? 1 : 0;
if(ub_s)
    ub_m = -ub_m;

clb_out = pack_float(lb_m, scale, lb_s);
cub_out = pack_float(lb_m, scale, ub_s);
```

Figure 5: *Hardware-oriented algorithm for modulo-encoded CBVH compression.* `lb`,`ub`*: Input uncompressed lower and upper bound* (p_i,q_i). `lb_mod_out`, `ub_mod_out`*: Output modulo coordinates* $(\hat{r}_i,\hat{s}_i)$.

4.4. Minimum scale adjustment

The techniques proposed so far nearly eliminate backtracking in many practical scenes. However, they have difficulties in the special case where the input BVH has large subtrees where all nodes share a degenerate axis, i.e., they have zero width along an axis. In practice, this arises when the scene has axis-aligned, planar, finely tesselated geometry. In this case, the entire subtree is first encoded bottom-up at the minimum scale, e.g. 2^{-126} - close to the minimum representable floating point number - in Figure 5. On encountering non-degenerate geometry with nonzero width along that scale, the

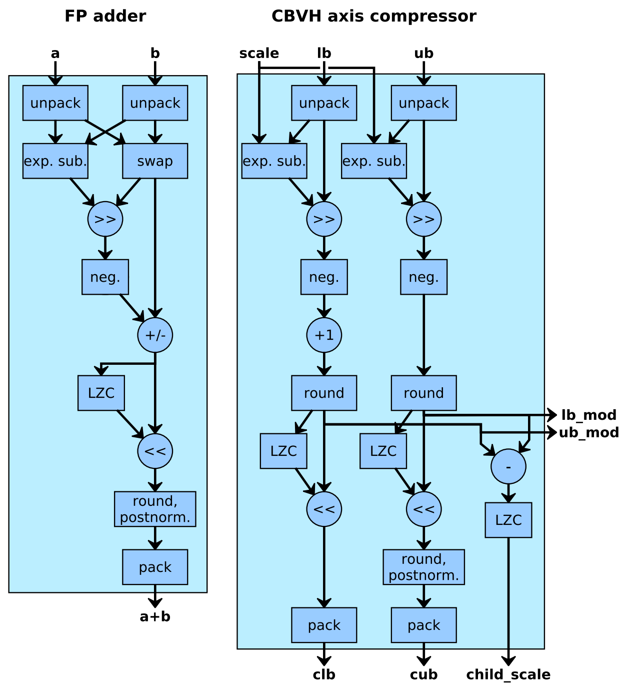


Figure 6: *Hardware implementation of a CBVH compressor (as in Figure 5 based on modulo coordinates, contrasted to a floating-point adder. LZC: leading zero counter. For clarity, only the significand datapath of the FP adder, which comprises most of the logic, is shown. The single-axis compressor has similar computational resources as two FP adders.*

subtree is backtracked, such that the scale gradually approaches the minimum throughout the subtree. Since each backtracking iteration can only decrease scale by a factor of 2^N, the entire subtree is backtracked. Modulo encoding is unhelpful in this case since the scale difference is large, and the distribution of backtracking depth is unfavorable for treeletwise compression.

An example from the **cloth** scene is shown in Fig. 9. This type of geometry is rare in typical scenes, but may easily occur in, e.g., synthetic animated scenes, such as the example. In this work, we approach this type of scene by selecting a minimum scale that is coarse enough to avoid the above issue, but fine enough that tree quality is unaffected. As a drawback, a parameter is added to the construction and traversal algorithms which may need adjustment based on scene size. However, we find later in evaluation that there is a wide margin for the value in practical scenes.

Another approach we experimented with was to snap upper bound coordinates up to numbers which are higher than the lower bound and exactly representable in single-precision floating point. For example, the degenerate geometry in the **cloth** scene lies on the plane $z = -0.4$. The corresponding primitive bounding boxes would, then, be $\text{ulp}(-0.4) = 2^{-25}$ wide, where ulp() is the *unit in the last place* function. This has similar effects as a manageable minimum scale. This approach avoids adding a parameter, however,

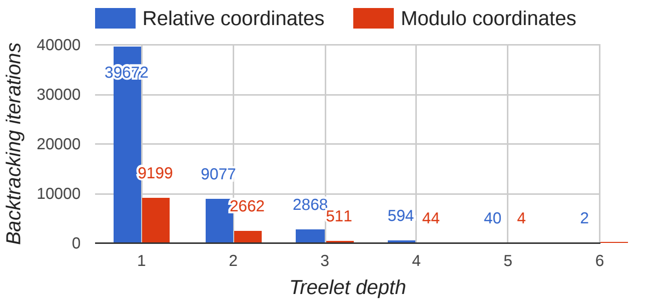


Figure 7: *Depth histogram of backtracking iterations in the Elephant scene with relative and modulo coordinates.*

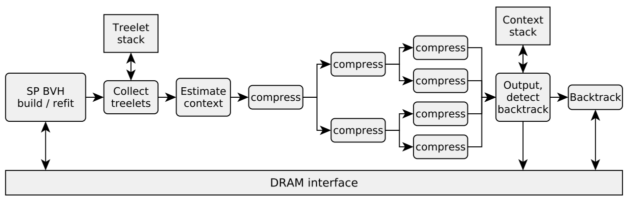


Figure 8: *Hardware architecture for treelet compression*

it does not work with degenerate geometry where the corresponding coordinate is zero.

4.5. Plane sharing

The methods described so far assume that 6 coordinates are stored per bounding box. It can be beneficial in terms of memory footprint to further compress the tree by reusing coordinates from the parent node, i.e., plane sharing [FD09]. This technique can also be used in CBVHs to reduce the arithmetic cost of traversal [VAMS16]. In this scheme, 6 coordinates are stored per a pair of bounding boxes, and the remaining 6 are reused from the parent AABB. Extra bits are included in the node datastructure to denote which coordinates are reused. However, unlike full-precision BVH, in CBVH this is a lossy process: a coordinate quantized to a coarse scale at a high hierarchy level often differs from the same coordinate quantized to a finer scale. This causes only modest quality loss, but greatly complicates bottom-up construction, as it often invalidates nodes far down in the hierarchy. So far, we do not have a satisfactory solution for plane sharing, and it is left as an open problem.

5. Evaluation

In order to evaluate the proposed methods, we implemented the proposed streaming bottom-up compression algorithms, as well as baseline top-down compression, in software. The algorithms are evaluated for LBVH construction and refitting, assuming hardware units like [VKJ*15, NKP*15]. A set of 16 test scenes is used for evaluation, as listed in Figure 11. Common assumptions about input and output data layouts are as follows. Input primitive data is stored as an array-of-structures of triangles with three vertices (á 36B) and a primitive ID (4B) per triangle. We assume a CBVH format with a memory footprint of 16B per node pair, which is the same

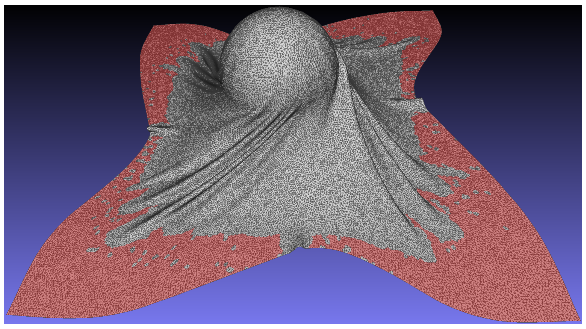


Figure 9: *Example scene (**cloth**) with a large amount of geometry with a degenerate axis (red).*

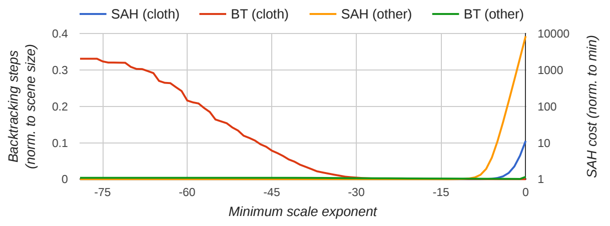


Figure 10: *Effect of minimum scale on tree quality and backtracking: **cloth** scene and average of other scenes. Tree quality is represented by SAH cost normalized to minimum value. The number of backtracking steps is normalized to scene size. The planar geometry in **cloth** begins to cause significant backtracking with a minimum scale of less than 2^{-30}. An overly coarse minimum scale, ca. 2^{-10}, harms tree quality.*

density as in [Kee14]. Further space could be saved by optimizing the child pointer representation as in, e.g., [SE10, LV16], but this is outside the scope of this work. Each node in a pair has six 6-bit relative or modulo coordinates and a leaf bit, which leaves space for a 27-bit child pointer. Leaf pointers in the CBVH index a primitive ID array as in, e.g., [Wal07], which references the input primitive data. LBVH needs to output the ID array, while refitting leaves it unchanged.

5.1. Minimum scale

We first calibrate the minimum scale parameter to be used in later benchmarks. As discussed in Subsection 4.4, the minimum scale should be coarse enough to avoid issues with axis-aligned, planar geometry. However, an overly coarse scale has an adverse effect on tree quality. The measured tradeoff is illustrated in Figure 10. In our set of benchmarks, only **cloth** has a significant amount of thin geometry, so we examine **cloth** and the other scenes separately. With a fine minimum scale, **cloth** exhibits a large amount of backtracking, but this is mostly eliminated at a minimum scale of 2^{-30}. Conversely, up to a minimum scale of 2^{-15}, there was no effect on scene quality. Therefore, there appears to be a wide margin for the minimum scale parameter. In the following benchmarks, we use a value of 2^{-30}. It should be noted that though axis-aligned geometry is often found in indoor and architectural scenes, several such scenes were included in the benchmark (**crytec**, **conference**, **italian**, **babylonian**), but did not cause significant backtracking even without minimum scale adjustment. Axis-aligned geometry, therefore, only seems to become an issue in synthetic worst-case scenes.

5.2. Backtracking and memory traffic

Memory traffic is computed based on the numbers of primitives, nodes and backtracking iterations recorded from the software builder. For LBVH, we assume the hardware builder [VKJ*15] is modified to output large leafs when multiple primitives share the same Morton code. The builder reads input primitives (40B traffic per primitive), sorts their AABBs (64B per node), and outputs a CBVH hierarchy (16B per node) and primitive ID array (4B per primitive). For refitting, we update LBVH trees to match the original geometry (as the exact geometry has no bearing on memory traffic). Refitting is assumed to proceed by Wald's algorithm [WBS07], i.e., it needs to read the CBVH hierarchy (16B per node), primitive ID array (4B per primitive) and primitives (40B per primitive), and output an updated CBVH hierarchy (16B per node). In the case of one primitive per leaf, this gives the memory traffic in Fig. 1. Finally, each backtracking iteration with the bottom-up builds requires a minimum-size DRAM read and write. We assume a DRAM interface with a 64B access granularity, e.g., a LPDDR4 interface with a 64-bit channel.

Table 1 shows results in aggregate and Table 3 per scene. We see that modulo encoding gives roughly the same benefits as one level of treeletwise compression: they reduce backtracking by 4.4x and 6.8x, respectively. The gains from modulo encoding are orthogonal with treeletwise compression: together, they reduce backtracking by 15x. With two-level treelet compression or one-level compression combined with modulo encoding, the amount of backtracking is so low that the memory traffic results are close to ideal.

The main scene parameter affecting memory traffic savings is the number of primitives per leaf: the worst-case **conference** scene has 8.7 triangles per leaf, while most scenes have 1–2. It should be noted that some LBVH builders store one primitive per leaf, e.g. [Kar12]. For this type of tree, streaming construction would consistently give best-case savings.

In LBVH, plain streaming bottom-up construction saves 16% memory traffic compared to the baseline of BVH output and post-processing compression. Modulo coding improves savings to 37%, a single level of treelet compression to 39%, and a combination of both techniques to 41%. At this point backtracking traffic is small, so adding two more levels of treelet compression improves savings only to 42%, even as backtracking falls 10x. In refitting the savings are larger: on average 21% without either backtracking optimization, 54% with both optimizations, and 56% with 4-level treelet compression.

Since modulo encoding adds some overhead to traversal, and similar memory traffic savings can be recovered by adding a level of treelet compression which only complicates the update, it may be advantageous to store the CBVH in relative coordinates and rely on treelet compression. In this case, modulo encoding is used to enable inexpensive compression hardware as in Figure 6, though

Table 1: *Normalized backtracking and memory traffic results with (baseline) postprocessing top-down compression and streaming bottom-up compression, with combinations of proposed modulo coordinates and treelet-based compression (with different treelet depths). Backtracking results are normalized to R0 and memory traffic to Base.*

Compression dir.		Top-down	Bottom-up (proposed)							
Coordinates		Relative					Modulo (proposed)			
Treelet depth		-	-	2	3	4	-	2	3	4
Backtrack iterations	Min	-	1.00	0.06	0.01	0.00	0.10	0.02	0.00	0.00
	Max	-	1.00	0.25	0.09	0.03	0.44	0.15	0.06	0.03
	Avg	-	1.00	0.15	0.04	0.01	0.22	0.06	0.02	0.01
Memory traffic (LBVH)	Min	1.00	0.75	0.53	0.51	0.50	0.54	0.51	0.50	0.50
	Max	1.00	0.95	0.86	0.84	0.83	0.87	0.84	0.83	0.83
	Avg	1.00	0.84	0.61	0.59	0.58	0.63	0.59	0.58	0.58
Memory traffic (Refit)	Min	1.00	0.67	0.41	0.39	0.38	0.42	0.39	0.38	0.38
	Max	1.00	0.90	0.74	0.70	0.69	0.76	0.71	0.69	0.68
	Avg	1.00	0.79	0.49	0.45	0.44	0.52	0.46	0.44	0.44

Table 2: *LBVH runtime results (ms) for single-precision build, postprocessing compression, and streaming compression with and without proposed optimizations.*

Output	BVH	CBVH			
Compression dir.	-	TD	BU (proposed)		
Coordinates	Float	Rel.	Rel.	Mod.	Leaf
Treelet depth	-	-	-	4	size
Toasters	0.2	0.2	0.3	**0.1**	1.1
Bunny	1.1	1.6	1.9	**0.8**	1.1
Elephant	1.3	1.9	2.2	**1.0**	1.1
Cloth	1.5	2.1	2.4	**1.1**	1.0
Fairy	2.1	2.5	2.6	**1.9**	3.8
Armadillo	3.4	4.8	5.7	**2.6**	1.2
Crytek	3.6	4.8	5.4	**3.0**	2.0
Conference	3.2	3.5	3.4	**2.9**	8.7
Sportscar	4.3	5.8	6.4	**3.5**	1.9
Italian	5.0	6.6	7.7	**4.2**	2.8
Babylonian	6.7	8.8	10.2	**5.6**	2.4
Hand	9.6	13.3	15.2	**7.6**	1.5
Dragon	13.4	19.0	21.5	**10.3**	1.3
Buddha	16.7	23.5	26.0	**13.0**	1.5
Lion	23.9	33.6	37.7	**18.7**	1.3
Hairball	40.9	56.5	52.2	**32.2**	1.4
Average	-	+36%	+52%	-20%	2.1

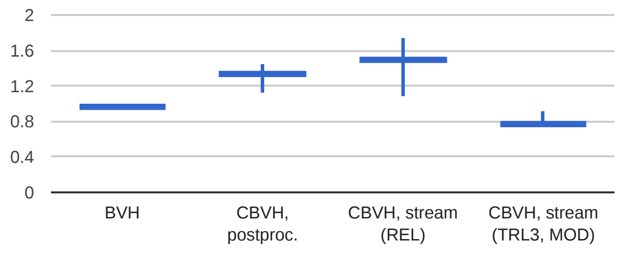


Figure 11: *LBVH runtime results, normalized to single-precision LBVH. Average values and ranges are shown.*

the resulting nodes are immediately converted to relative representation.

5.3. Runtime

We further examine the runtime effects of the proposed techniques on LBVH construction by means of architectural simulations, based on cycle-level simulation of the LBVH builder. Four alternatives are compared: single-precision BVH build, postprocessing top-down compression, streaming compression, and streaming compression with the proposed optimizations. In the last alternative, modulo encoding is used and treelet depth is set at 4. For a conservative comparison, we use optimistic assumptions for the postprocessing compression hardware used as baseline, and pessimistic assumptions for the backtracking unit required by the proposed method. The backtracking unit performs backtracking steps sequentially, while the compression unit starts each memory operation immediately after its dependencies are available, corresponding to a highly parallel, multi-threaded implementation. The LBVH builder is configured with a 256KB scratchpad memory and a buffer size of 8 AABBs, such that it can handle scenes of up to 4M triangles. The operating frequency is set at 1GHz. We assume a dual-channel, 32-bit LPDDR3-1600 memory interface with a maximum bandwidth of 12.8GB/s, which is simulated with the cycle-accurate Ramulator model [KYM15].

Runtime results are shown in Table 2 and Figure 11. As expected from the memory traffic results, the proposed method is consistently faster than single-precision BVH construction, while postprocessing compression is slower. On average, the proposed methods are 20% faster than single-precision LBVH, while baseline post-processing top-down compression is 36% slower.

5.4. Tree quality

In order to verify the quality of the constructed trees, we ray traced each test scene and counted intersection tests. CBVHs required, on average, 8% more box tests and 13% more triangle tests than BVH, in line with previous work [Kee14, VAMS16]. The proposed techniques had $< 1\%$ effect on quality compared to a top-down build.

5.5. Watertightness

In addition to performance and tree quality, it is interesting whether a ray tracing system is watertight, i.e., whether it is guaranteed to avoid false misses. Vaidyanathan's traversal algorithm [VAMS16] is proven watertight based on the criterion that exact parametric distances t_{max}, t_{min} to each AABB are enclosed in the distances computed during traversal. We do not have a formal proof that our trees satisfy the criterion, but verified that this criterion held over every box test when path tracing all test scenes. Moreover, decompressing our bounds to single precision always gave AABBs that enclose the uncompressed AABB.

6. Limitations and Future Work

A main limitation of the present work is that shared-plane CBVHs [VAMS16] cannot yet be updated, and are left as an open problem. It may also be interesting to use shallow hierarchies as in [VKJT16] instead of shared-plane encoding, as this gives at least some of the memory footprint advantage of shared-plane encoding while making bottom-up updates more tractable. Moreover, since the node size of shared-plane CBVHs is small compared to cache lines, only a small fraction of the memory footprint advantage translates into memory bandwidth savings [VAMS16], at least in a straightforward implementation. From this perspective, it may be advantageous to use a shallow hierarchy and traverse fewer, larger nodes.

In this work, full-precision child pointers were used for simplicity, but it seems possible to integrate, e.g., the techniques of Liktor et al. [LV16] or Segovia et al. [SE10] to compress pointer fields. Moreover, using compact primitive storage formats such as triangle strips [LYM07] may be a low-hanging fruit to speed up tree update. Although this article focused on fixed-function hardware, the proposed encoding may also be interesting for CBVH builds on a programmable GPU, where bottom-up algorithms are preferred in order to take advantage of the GPU's parallel resources.

7. Conclusion

This article showed that the construction and refitting of compressed BVH trees can be significantly optimized by means of streaming, bottom-up compression. Two novel techniques were proposed to enable the rapid hardware-accelerated update of compressed BVHs in animated scenes: modulo encoding and treelet-based compression. Together, the proposed techniques allow streaming compression in bottom-up order, reducing memory traffic from LBVH construction by 38% and from refitting by 54% on average, compared to a postprocessing compression step. Since both LBVH and refitting are memory-bound algorithms, these savings translate directly into improvements in performance and energy efficiency, and are especially significant on mobile devices with limited power budget and memory bandwidth. Runtime was evaluated for LBVH builds, and was in line with the memory traffic reductions. Notably, it is faster to update CBVHs than uncompressed BVHs with the proposed method. The described work represents a step toward real-time, mobile ray tracing of large-scale animated scenes.

Acknowledgement

The 3D models used are courtesy of Ingo Wald (Fairy), Andrew Kensler (Toasters), Yasutoshi Mori (Sportscar), Frank Meinl (Crytek Sponza), Jonathan Good (Italian, Babylonian), Anat Grynberg and Greg Ward (Conference), Naga Govindaraju, Ilknur Kabul and Stephane Redon (Cloth), the Stanford Computer Graphics Laboratory (Bunny, Buddha, Dragon), and Samuli Laine (Hairball). Crytek Sponza and Dragon have modifications courtesy of Morgan McGuire [McG11].

References

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)* (2014). 2

[DFM13] DOYLE M., FOWLER C., MANZKE M.: A hardware unit for fast SAH-optimized BVH construction. *ACM Transactions on Graphics 32*, 4 (2013), 139:1–10. 1, 3, 5

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proc. High-Performance Graphics* (2015), pp. 13–20. 2

[FD09] FABIANOWSKI B., DINGLIANA J.: Compact bvh storage for ray tracing and photon mapping. In *Proc. Eurographics Ireland Workshop* (2009), pp. 1–8. 2, 6, 7

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proc. High-Performance Graphics* (2011), pp. 59–64. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *Computer Graphics and Applications 7*, 5 (1987), 14–20. 2

[IWP07] IZE T., WALD I., PARKER S. G.: Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proc. Eurographics Conf. Parallel Graphics and Visualization* (2007), pp. 101–108. 2

[Kar12] KARRAS T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proc. High-Performance Graphics* (2012), pp. 33–37. 2, 8

[Kee14] KEELY S.: Reduced precision hardware for ray tracing. In *Proc. High-Performance Graphics* (2014), pp. 29–40. 1, 2, 3, 8, 9

[KIS*12] KOPTA D., IZE T., SPJUT J., BRUNVAND E., DAVIS A., KENSLER A.: Fast, effective BVH updates for animated scenes. In *Proc. Symp. Interactive 3D Graphics and Games* (2012), pp. 197–204. 1, 2

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. In *ACM SIGGRAPH Computer Graphics* (1986), vol. 20, ACM, pp. 269–278. 1, 3

[KYM15] KIM Y., YANG W., MUTLU O.: Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters PP*, 99 (2015), 1–1. 9

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 1, 2

[LSL*13] LEE W., SHIN Y., LEE J., KIM J., NAH J., JUNG S., LEE S., PARK H., HAN T.: SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. High-Performance Graphics* (2013), pp. 109–119. 1

[LV16] LIKTOR G., VAIDYANATHAN K.: Bandwidth-efficient BVH layout for incremental hardware traversal. In *Proc. High Performance Graphics* (2016), pp. 51–61. 1, 8, 10

[LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2007), IEEE, pp. 19–26. 10

[McG11] McGuire M.: Computer graphics archive, 2011. http://graphics.cs.williams.edu/data/meshes.xml. 10

[MW06] Mahovsky J., Wyvill B.: Memory-conserving bounding volume hierarchies with coherent raytracing. In *Computer Graphics Forum* (2006), vol. 25, pp. 173–182. 2

[NKK*14] Nah J., Kwon H., Kim D., Jeong C., Park J., Han T., Manocha D., Park W.: RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics 33*, 5 (2014), 162:1–15. 1

[NKP*15] Nah J., Kim J., Park J., Lee W., Park J., Jung S., Park W., Manocha D., Han T.: HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3 (2015), 389–401. 1, 2, 3, 7

[PL10] Pantaleoni J., Luebke D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. High-Performance Graphics* (2010), pp. 87–95. 2

[Pow15] PowerVR: PowerVR ray tracing. Accessed Feb 20, 2017, 2015. http://imgtec.com/powervr/ray-tracing/. 1

[SE10] Segovia B., Ernst M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proc. Graphics Interface* (2010), pp. 153–160. 2, 8, 10

[SKBD12] Spjut J., Kopta D., Brunvand E., Davis A.: A mobile accelerator architecture for ray tracing. In *Proc. Workshop on SoCs, Heterogeneous Architectures and Workloads* (2012). 1

[SKKB09] Spjut J., Kensler A., Kopta D., Brunvand E.: TRaX: a multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28*, 12 (2009), 1802–1815. 1

[VAMS16] Vaidyanathan K., Akenine-Möller T., Salvi M.: Watertight ray traversal with reduced precision. *Proc. High-Performance Graph* (2016). 2, 3, 4, 5, 7, 9, 10

[VKJ*15] Viitanen T., Koskela M., Jääskeläinen P., Kultala H., Takala J.: MergeTree: a HLBVH constructor for mobile systems. In *SIGGRAPH Asia Technical Briefs* (2015), p. 12. 1, 2, 3, 7, 8

[VKJT16] Viitanen T., Koskela M., Jääskeläinen P., Takala J.: Multi bounding volume hierarchies for ray tracing pipelines. In *SIGGRAPH Asia Technical Briefs* (2016), p. 8. 10

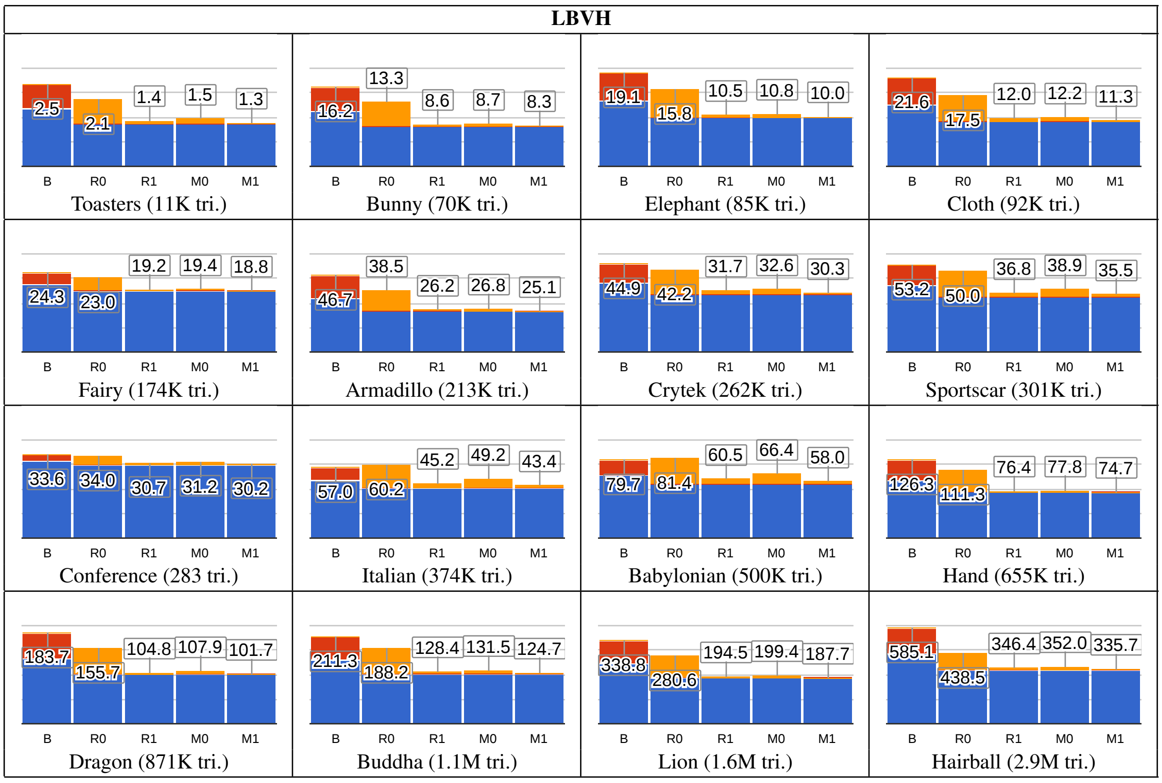
[Wal07] Wald I.: On fast construction of SAH-based bounding volume hierarchies. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2007), pp. 33–40. 1, 2, 8

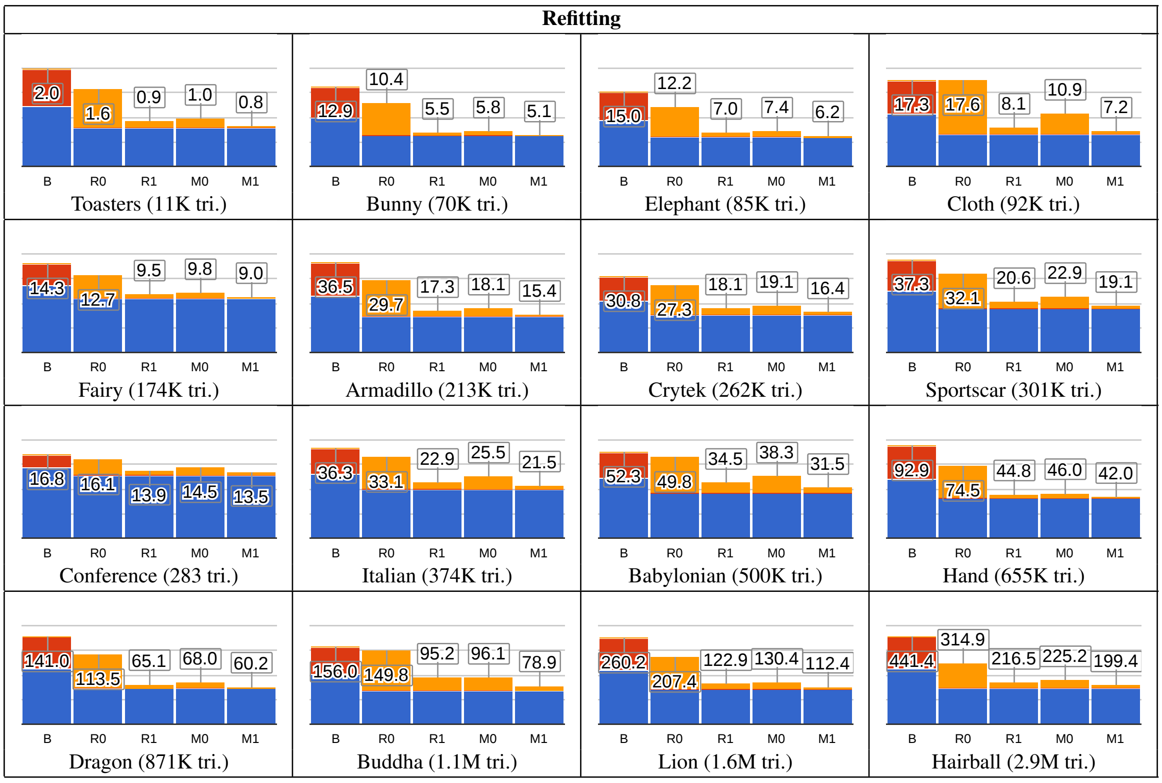
[WBB08] Wald I., Benthin C., Boulos S.: Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *Proc. IEEE Int. Symp. Interactive Ray Tracing* (2008), IEEE, pp. 49–57. 1

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 6. 1, 2, 8

[WSS05] Woop S., Schmittler J., Slusallek P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444. 1

Table 3: *Memory traffic with LBVH and refitting. B: Baseline, separate top-down compression. R0: Relative encoding. R1: Relative encoding, 1 layer of treelet-based compression. M0: Modulo encoding. M1: Modulo encoding, 1 layer of treelet-based compression. Red: Separate compression, blue: construction/refit, orange: backtracking.*


LBVH
B R0 R1 M0 M1
Toasters (11K tri.)
B R0 R1 M0 M1
Bunny (70K tri.)
B R0 R1 M0 M1
Elephant (85K tri.)
B R0 R1 M0 M1
Cloth (92K tri.)
B R0 R1 M0 M1
Fairy (174K tri.)
B R0 R1 M0 M1
Armadillo (213K tri.)
B R0 R1 M0 M1
Crytek (262K tri.)
B R0 R1 M0 M1
Sportscar (301K tri.)
B R0 R1 M0 M1
Conference (283 tri.)
B R0 R1 M0 M1
Italian (374K tri.)
B R0 R1 M0 M1
Babylonian (500K tri.)
B R0 R1 M0 M1
Hand (655K tri.)
B R0 R1 M0 M1
Dragon (871K tri.)
B R0 R1 M0 M1
Buddha (1.1M tri.)
B R0 R1 M0 M1
Lion (1.6M tri.)
B R0 R1 M0 M1
Hairball (2.9M tri.)


Refitting
B R0 R1 M0 M1
Toasters (11K tri.)
B R0 R1 M0 M1
Bunny (70K tri.)
B R0 R1 M0 M1
Elephant (85K tri.)
B R0 R1 M0 M1
Cloth (92K tri.)
B R0 R1 M0 M1
Fairy (174K tri.)
B R0 R1 M0 M1
Armadillo (213K tri.)
B R0 R1 M0 M1
Crytek (262K tri.)
B R0 R1 M0 M1
Sportscar (301K tri.)
B R0 R1 M0 M1
Conference (283 tri.)
B R0 R1 M0 M1
Italian (374K tri.)
B R0 R1 M0 M1
Babylonian (500K tri.)
B R0 R1 M0 M1
Hand (655K tri.)
B R0 R1 M0 M1
Dragon (871K tri.)
B R0 R1 M0 M1
Buddha (1.1M tri.)
B R0 R1 M0 M1
Lion (1.6M tri.)
B R0 R1 M0 M1
Hairball (2.9M tri.)