Compressed-Leaf Bounding Volume Hierarchies (originally submitted, un-shortened version)

Carsten Benthin Ingo Wald Sven Woop Attila T. Áfra
Intel Corporation

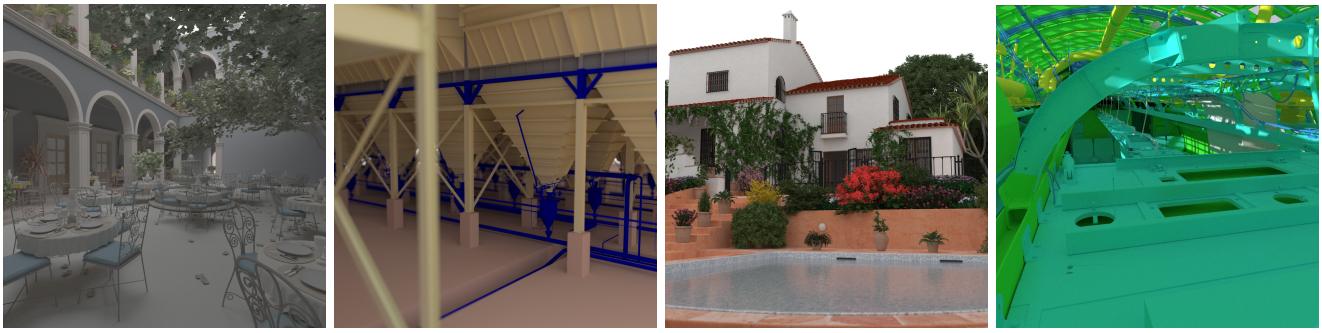

Figure 1: Four example scenes used for evaluating our method: San Miguel (10.5M triangles), Powerplant (12.7M triangles), Villa (38.3M triangles), and the Boeing model (350M triangles).

ABSTRACT

We propose and evaluate what we call *Compressed-Leaf Bounding Volume Hierarchies* (CLBVH), which strike a balance between compressed and non-compressed BVH layouts. Our CLBVH layout introduces dedicated compressed multi-leaf nodes where most effective at reducing memory use, and uses regular BVH nodes for inner nodes and small, isolated leaves. We show that when implemented within the Embree ray tracing framework, this approach achieves roughly the same memory savings as Embree's current compressed BVH layout, while maintaining almost the full performance of its fastest non-compressed BVH.

KEYWORDS

ray tracing, bounding volume hierarchy, compression

1 INTRODUCTION

The key to improving fast ray tracing is the use of acceleration data structures. Though indispensable for performance, such structures require both time and memory to be built and stored. In particular, the memory overhead of the acceleration structure can be a significant factor for large scenes.

One way to reduce this memory overhead is to compress the acceleration data structure. Compression works particularly well for BVHs, which naturally lead to conservative, incremental encoding. This can significantly reduce the size of the acceleration structure (often halving the size of BVH nodes), at the cost of introducing additional overheads.

This is the authors' version of a paper that has been accepted at (and thus, is to be published at) High Performance Graphics 2018 (HPG'18). Please note, however, that the version that is to appear at HPG is going to be a significantly shortened version of this author's copy (which is basically the paper as originally submitted). As such, this author's version here may contain additional version that may not appear in the final version; but the *newer* and *official* version of this work is the short paper that is to appear at HPG18.

In the case of BVHs, these overheads fall into three categories: First, there is the obvious cost of decompressing each BVH node during traversal; second, in particular for hierarchical encoding schemes, the need to track parent information slightly complicates the traversal; and third, conservatively quantizing each node's bounds results in less tight bounding boxes compared to uncompressed ones, resulting in a small increase in the number of nodes and primitives that have to be traversed and intersected, respectively.

On GPUs, the savings in node bandwidth may completely offset the additional ray traversal overhead; on CPUs some overhead typically remains. However, even with the increased traversal cost, for memory limited applications this trade-off between ray traversal performance and memory use can be appealing.

In this paper, we propose and evaluate an approach to BVH compression that improves upon fully-compressed wide BVHs by introducing a new dedicated node type for *compressed multi-leaf nodes* where applicable, while using fast, uncompressed BVH nodes for interior nodes and isolated individual leaf nodes.

Typically, wide BVHs use a data layout where all of an individual node's N children are stored together in a consecutive block, typically in a SoA data layout. This allows for addressing all of a parent's N children with a single pointer and aids vectorization, but slightly confuses the terminology of what a "node" in a wide BVH actually is. Throughout the paper we will refer to each group of N siblings as a N-wide *multi*-node, with each sibling consisting of N *individual* nodes. We refer to the lowest level of individual nodes as *leaf nodes*, each logically containing one bounding box and a pointer to the actual *leaf data*.

The fundamental insight behind our compression approach is that almost all of the savings of BVH compression comes from compressing the leaf nodes, which in 8-wide BVHs make

up the majority of nodes, while most of the traversal overhead comes from traversing the interior nodes. Consequently, we focus on compressing just the leaf nodes, by introducing dedicated *compressed multi-leaf nodes*. Our approach achieves similar or better compression to fully-compressed BVHs, while having nearly the same traversal performance as uncompressed BVHs, as no decompression is required to traverse interior nodes.

2 RELATED WORK

Acceleration structures for ray tracing have a long history; a survey on the general concepts can be found in Havran's thesis [4]. Today, most ray tracers use some sort of BVH, typically with a branching factor of 4 or 8, and in some cases 16 [2, 3, 7, 9, 10]. While in the past each ray tracer implemented its own acceleration structures and traversal methods, the last few years have seen the emergence of commonly accepted ray tracing libraries such as Embree for CPUs [10], and OptiX for GPUs [7], both of which use wide BVHs.

BVHs combine a hierarchical data structure with conservative bounding volumes, and thus lend themselves naturally to hierarchical, incremental encoding. Expressing each child node's bounding box relative to the bounds of its parent allows to quantize these child boxes, thereby representing them with fewer bits. During traversal these boxes are then decompressed on-the-fly by dequantizing them relative to the previous traversal step's bounding box. This concept of incremental encoding was first proposed by Mahovski et al. [6], and has since been extended even to compressing the geometry itself [8].

The downside to hierarchical encoding is that tracking the respective decompressed parent boxes on the stack is costly, and complicates traversal. At least for wide branching BVHs—where such information can be amortized over multiple child nodes—this can be avoided by having each multi-node store not only the quantized child boxes, but also the single full precision parent box that the N quantized child boxes are relative to. This of course reduces the effectiveness of compression, but makes for much simpler and faster traversal. This approach has recently been adopted on both CPUs [10] and GPUs [11], and has even been proposed for dedicated ray tracing hardware [5]. In particular, while originally proposed mainly for reducing memory footprint, the reduced memory bandwidth requirements have been found to significantly increase performance on both GPUs [11] and dedicated hardware [5].

As proposed by Segovia et al. [8], the concept of encoding children's bounding boxes relative to their parents can be extended to encoding primitives relative to the child nodes they are located in. In this paper, we will only focus on compressing the BVH data, leaving the geometry itself untouched.

3 COMPRESSED-LEAF BVHS

Our method is driven by two key insights: First, that at least in the case of 4- or 8-wide BVHs, compressing only leaf nodes can yield most of the benefits of compressing all nodes, while minimizing the downsides; and second, that in wide BVHs it is often possible to find entire groups of leaf nodes that, when considered together, allow for even *more* memory savings than for interior nodes, by defining a special multi-leaf node type and compressing this. For the sake of brevity, throughout the rest of the paper we will only consider 8-wide BVHs (`BVH8`), which have recently seen wide adoption on both CPUs [10] and GPUs [11].

With regard to the first insight, we observe that in a binary BVH half of the nodes are leaf nodes, in a `BVH8` this ratio rises to over 80%. Furthermore, even in a shallow `BVH8` a ray will typically traverse multiple inner nodes per visited leaf node, so assuming the overhead due to decompression and less-tight bounding boxes is the same for inner and leaf nodes, most of the traversal overhead will be generated by the inner nodes. Consequently, compressing *only* the leaf nodes will provide most of the memory savings, yet come with little of the traversal overhead.

The caveat to this argument is that for a `BVH8` compressing only leaf nodes is not that easy, because every "node" in such a BVH is actually a *multi*-node of 8 individual nodes, each of which could be either a leaf or an inner node. However, our second insight is that the individual node types found in an 8-wide multi-node in practice are not random, and it is in fact relatively easy to build BVHs such that most (though certainly not all) leaf nodes end up in multi-nodes that contain *only* leaf nodes (Section 4.3).

Our method optimizes for this case by introducing a new *compressed multi-leaf* node type which replaces these all-leaf multi-nodes (see Listing 1), while leaving both inner and isolated leaf nodes unchanged. Moreover, as we know that all children of these compressed multi-leaf nodes are leaves we can do *additional* optimizations. In particular, we can store the data the child leaves would have pointed to right after the node itself in memory, eliminating the (otherwise hard to compress) child pointers from those nodes, and providing even more potential for compressing the resulting treelet-like structures (see Section 4.5).

In summary, our resulting BVH—which we call a Compressed-Leaf BVH (`CLBVH`)—has two multi-node types: regular `BVH8` multi-nodes containing 8 individual nodes, each of which could be a regular inner node, or a regular individual leaf node, just as in Embree's original `BVH8` multi-nodes; and our new compressed multi-leaf node which stores (up to) 8 individual leaves, in compressed form (also see Listing 1).

Since we will use a surface area heuristic (SAH) [**?**] BVH builder it is theoretically possible to contrive degenerate inputs where most leaves would end up in mixed multi-nodes, and result in very few of our compressed multi-leaf nodes being created. In practice, it is relatively easy to get the vast majority of leaves to end up in compressed multi-leaf nodes (see Section 4.3), and since we can actually compress these nodes better than a traditional compressed 8-wide BVH (`QBVH8`), we typically end up with the same and sometimes even slightly better compression ratios.

4 IMPLEMENTATION

We implement and evaluate the previously discussed strategy within a modified version of Embree 3.0.

4.1 Node Compression and Decompression

In Embree's `BVH8` data layout, each multi-node contains 8 bounding boxes and 8 (64-bit) child pointers (see Listing 1). For each individual node in such a multi-node, the `childRef` value encodes whether the node is an inner or leaf node; as well as the pointer. For inner nodes the pointer refers to another `BVH8` multi-node; for leaves it points to the leaf's *leaf data*, the list of primitive data belonging to the leaf. Exactly what primitive data is stored in a leaf depends on the BVH type: for triangles, it is either a list of `triangle4` structures, fully pre-gathered vertices of four triangles in SoA layout; or a list of `triangle4i` with four triangles' worth of vertex indices.

Embree's fully compressed `QBVH8` structure employs a similar approach to Ylitie et al. [11]. In the `QBVH8` layout, the 8 child bounding boxes are expressed relative to the parent's bounding box, and quantized to 8-bit fixed point values. Each `QBVH8` multi-node stores the parent bounding box in the form of its *start* and *extent*, stored as two 3-dimensional single precision vectors (2×12 bytes). Each child's bounding box is stored as 2×3 bytes, for the box's *lower* and *upper* bounds, requiring 48 bytes for all 8 children. Including the 8 child pointers, this sums to a total of 136 bytes, slightly more than half an uncompressed `BVH8` multi-node. Note that this layout differs from [11], as the *extent* is stored in full precision; this provides tighter bounds but requires more space.

Listing 1: Illustration of the three BVH node types. Top: Embree's regular BVH8 nodes contain 8 pointers and float boxes (256 bytes). Middle: Embree's quantized QBVH8 nodes contain 8 pointers, 8 quantized bounding boxes, and 6 floats to specify the dequantization domain (136 bytes). Bottom: At the leaf level, our method introduces an even smaller compressed BVH node type (72 bytes)—knowing it will only contain leaf nodes—and omits the pointers by storing the primitive data right after the node itself.

```
// BVH8 multi-node: 192+64=256 bytes
struct BVH8MultiNode {
  // one float box per child
  box3f childBounds[8];
  // child pointers (to nodes _or_ primitives)
  uint64 childRef[8];
};
// QBVH8 multi-node: 24+48+64=136 bytes
struct QBVH8MultiNode {
  // shared de-quantization start and extent
  vec3f start, extent;
  // 8 child boxes, in 8-bit fixed-point
  box3ui8 childBounds[8];
  // child pointers (to nodes _or_ primitives)
  uint64 childRef[8];
};
// Compressed multi-leaf BVH node: 24+48=72 bytes
struct CLBVHMultiNode {
  // shared de-quantization start and extent
  vec3f start, extent;
  // 8 child boxes, in 8-bit fixed-point
  box3ui8 childBounds[8];
  // implicit pointer
  // primitive data stored right behind this node
  LeafPrimData childPrims[0];
};
```

Decompressing a `QBVH8` node is done for each traversal step by computing the lower bound (and similarly, the upper bound) as $QBVH8.start_i + (float)QBVH8.lower_i * QBVH8.extent_i$. On a CPU this requires 8 logical instructions per dimension and box: 2 loads (*start*, *extent*), 2 byte-to-int load and up-conversion (for lower and upper bounds), 2 int-to-float conversions, and 2 multiply-adds. The decompression can be done for all 8 quantized child bounding boxes in parallel using SIMD instructions, resulting in an overhead of around 24 instructions for a single ray-node intersection test, making it at least more than twice as expensive as for an uncompressed `BVH8` node.

4.2 Introducing Compressed Multi-Leaf Nodes

The main limitation of the existing `QBVH8` multi-nodes is that the eight 64-bit pointers cannot easily be compressed, limiting the level of compression that can be achieved. We address this by explicitly targeting only multi-nodes that contain only leaves, in which case we know that the eight pointers all refer to leaf data. We exploit this by storing the referenced primitive data directly after the `QBVH8` node itself (see Listing 1), allowing us to replace 64-bit pointers with 8-bit offsets. In fact, in cases where the builder can guarantee to always produce exactly one `triangle4` per leaf we can remove these 8-bit offsets, and compute each leaf's child pointer implicitly.

We use the same layout and compression/decompression scheme as the `QBVH8` nodes for storing the compressed multi-leaf node bounding boxes. Inner nodes are almost completely unchanged, the only modification required is that the child pointers that point to a leaf need one more bit (e.g. the LSB of the pointer) to encode what kind of leaf it points to, an isolated leaf node, or our compressed multi-leaf node.

By eliminating the primitive pointers for the leaf nodes our compressed multi-leaf node requires only 72 bytes. Compared to an uncompressed `BVH8` multi-node (256 bytes) this yields a compression factor of over 3×.

4.3 Builder Modifications

Since our scheme will only compress multi-nodes made up of *only* leaf nodes, its effectiveness will depend significantly on how common such nodes are. Fortunately, for wide BVHs such nodes turn out to be the more common leaf node type. While it is possible to get isolated leaves when small groups of triangles are far away from the bulk of the scene's triangles, in which case a good SAH builder will most certainly separate them. However, the far more common case is that at some point the number of triangles in a sub-tree contains less

Table 1: Detailed memory usage (in MB), traversal statistics, and render performance (in MRays/s) for our CLBVH relative to Embree's regular uncompressed BVH8 and fully compressed QBVH8 variants; for two typical configurations: highest performance (SBVH+pre-gathered triangle data), and lowest memory consumption (BVH+triangle indices).

	Embree "best speed" (SBVH, triangles only) (leaves store full pre-gathered primitives)					Embree "least memory" (BVH, triangles only) (leaves store only vertex indices)						
	BVH8 uncompressed	QBVH8 fully compressed		CLBVH (ours) (fast)		BVH8 uncompressed	QBVH8 fully compressed		CLBVH (ours) (fast)		(compact)	
San Miguel												
BVH/QBVH memory	152/-	-/85.3	-43.7%	53.6/30.5	-44.5%	141/-	-/79.5	-43.7%	50.3/28.4	-44.3%	50.3/28.4	
leaf data memory	559	559		558		239	239		239		179	-25.1%
total memory	711	645	-9.3%	643	-9.4%	380	318	-16.2%	317	-16.2%	257	-32.3%
travs/isecs per ray	16.8/5.0	17.2/5.2		16.8/5.1		20.4/5.26	20.8/5.58		20.4/5.36		20.4/5.36	
MRays/s	113	100	-11.5%	110	-2.5%	88.8	79.7	-11.3%	88.4	-0.4%	79.9	-10.0%
Powerplant												
BVH/QBVH memory	178/-	-/103	-41.8%	68.5/33.9	-42.3%	151/-	-/84.7	-43.7%	56.9/29.2	-42.8%	56.9/29.2	
leaf data memory	735	735		733		280	280		280		214	-23%
total memory	913	868	-4.9%	835	-8.5%	481	365	-24.1%	366	-23.8%	301	-37.4%
travs/isecs per ray	11.8/3.7	12.2/4.3		11.7/4.1		16.8/5.0	17.2/5.2		16.8/5.13		16.8/5.13	
MRays/s	188	153	-19.0%	181	-4.0%	90.2	76.8	-14.8%	86.8	-3.7%	80.0	-11.2%
Villa												
BVH/QBVH memory	625/-	-/351	-43.7%	212/133	-45.8%	542/-	-/305	-43.7%	185/116	-45.8%	185/116	
leaf data memory	2273	2274		2273		899	899		899		762	-15.2%
total memory	2898	2625	-9.4%	2608	-10.1%	1440	1204	-16.4%	1192	-17.3%	1063	-26.2
travs/isecs per ray	15.24/4.82	15.5/5.0		15.2/4.95		17.5/5.56	17.9/5.77		17.5/5.67		17.5/5.67	
MRays/s	118	105	-10.7%	115	-2.2%	87.9	81.5	-7.3%	88.1	+.01%	82.2	-6.4%
Boeing												
BVH/QBVH memory	5132/-	-/2887	-43.7%	1879/1013	-43.6%	4598/-	-/2586	-43.8%	1661/915	-44.0%	1661/915	
leaf data memory	18942	18942		18941		7640	7640		7640		6206	-18.7%
total memory	24074	21829	-9.3%	21833	-9.3%	12238	10226	-16.4%	10216	-16.5%	8782	-28.2%
travs/isecs per ray	16.0/5.6	16.3/5.9		16.0 / 5.7		25.3/7.96	26.1/8.27		25.3/8.05		25.3/8.1	
MRays/s	111.4	98.5	-11.5%	108	-3.3%	68.2	60.4	-11.4%	66.7	-2.2%	63.0	-7.5%

triangles than eight times the number of targeted triangles per leaf, at which point the builder will almost invariably generate all leaf nodes.

As a result we require only a simple modification to the BVH build algorithm to produce our compressed multi-leaf nodes. We take the existing top-down BVH8 builder and, whenever it starts building a new multi-node, check whether the current number of primitives is below a certain threshold. In our implementation we use $N \times M$ as this threshold, where N refers to the BVH branch factor, and M to the number of target primitives within a leaf node. For example, Embree by default targets four triangles per leaf in its `BVH8` builder, so our threshold would be 32. Any time the builder is asked to build a multi-node with less than this number of triangles it enters a special code path, in which it will continue the SAH-based splitting process for as long as the SAH decides this is useful. It then takes the resulting (at most) 8 leaves and builds one of our compressed multi-leaf nodes, which also includes copying that node's leaf data right behind the generated multi-node.

In all other cases (i.e., when building a multi-node with more than 32 triangles) the builder proceeds exactly as before, first using the SAH to split the group of triangles into at most 8 sub-sets, and builds a regular `BVH8` multi-node from this. Note that it is perfectly valid for some of the nodes of this multi-node to become individual leaves if they have less than the target of four triangles in them.

4.4 Traversal

The BVH traversal method is only marginally affected by the introduction of our `CLBVH` multi-nodes. For the purpose of the tree traversal these multi-nodes are simply treated as leaves, and the top-down-traversal loop traverses nodes exactly as before, until it reaches a leaf.

Only in this leaf intersection code does our method require some slight modifications: If the leaf node reached is a regular isolated leaf node it is intersected in the same way as before; if it is a `CLBVH` node we first decompress it, test the ray against the 8 children boxes, and then intersect the primitives in the nodes intersected by the ray. Interestingly, ordering the compressed multi-leaf node's intersected children based on the intersection distance did not provide any benefit. In the majority of cases only a single node is intersected, so the cost incurred to sort the nodes does not pay off.

4.5 Compressing Leaf Data

Though our method initially aimed at only compressing multi-nodes, once we have found these nodes with 8 leaves we can do more with them. In addition to eliminating the child pointers we can also realize that a multi-node acts as a mini "treelet", with a small set of triangles that in the common case share common features. For example, triangles within a `CLBVH` node are very likely to share vertices or vertex indices, and properties (e.g., object ID, etc.). This in turn means we can apply additional (lossless) compression of the primitive leaf data, by extracting these common features, allowing for additional compression which was not previously possible in the original `QBVH8`.

Though a complete investigation of the range of available compression options are beyond the scope of this paper, we have implemented a simple proof of concept where we store every vertex index used in the 8-wide compressed multi-leaf node only once, and in each leaf use only 8-bit indices into this list of 32-bit vertex indices. Further compression is likely possible, but we find even this approach already yields

additional memory savings, (see Section 5), albeit at some additional decompression cost.

5 RESULTS

To evaluate the performance and memory impact of our method we use a modified version of Embree 3.0 and the publicly available *protoray* path tracer [1]. The renderer was configured for purely-diffuse path tracing with up to eight bounces. This was chosen to ensure that the performance impact of shading would not be unduly masked by shading cost (in our setup shading costs only ˜15-20% of the total run-time), while at the same time ensuring realistic ray distributions.

In terms of hardware, we used a dual-socket Xeon Platinum 8180 workstation with 2×28 cores and 96 GB of memory. We used four different models as benchmark scenes, with complexity ranging from 10M to 350M triangles (Figure 1).

Due to the many possible variations Embree's data structures can be configured in, testing all of these variants would be prohibitive. As such, we settled on two configurations which we believe to be representative of common use cases. In the "best performance" case we assume the user is not limited by memory, and chooses all options for best performance; in particular, in this setup we enable spatial splits and use the pre-gathered `triangle4` layout (that even pre-gathers vertex data into the leaf data). In the polar opposite "least memory" case we assume a user that is memory constrained, and therefore uses the `triangle4i` primitive layout (which only stores vertex indices), does not enable spatial splits, etc.

For both of these configurations we report data for: Embree's default uncompressed `BVH8`; its existing `QBVH8` in which both inner and leaf nodes are equally compressed; our `CLBVH` with regular inner nodes and compressed multi-leaf nodes, but no additional compression of the primitive data (`CLBVH-fast`); and at least for the "least memory" case, our `CLBVH` with the additional lossless compression of the indices stored in the compressed leaves (`CLBVH-compact`). The final configuration currently only supports the `triangle4i` layout, so is included only for this configuration.

5.1 Comparison to BVH8 and QBVH8

In Table 1, we have compiled a comprehensive list of different traversal, memory usage, and performance statistics for the seven most important configurations (see Table 2 for a distilled summary).

"Best Performance" Configuration. Looking first at only the "best performance configuration" we see that the `QBVH8` and our `CLBVH` are roughly on par in terms of memory savings. Our `CLBVH` does not compress inner nodes, and thus eventually spends about two thirds of node memory on these uncompressed inner nodes; however, our leaf nodes are nearly half the size of the `QBVH8` leaf nodes, and as such the total node memory use is similar in all cases. At this equal memory footprint, our `CLBVH` generally performs much better, reducing the performance impact from 10-20% down to 2-4%. However, in terms of *total* memory savings neither method can save more than about 10% of total memory, as in this configuration the lion's share goes into pre-gathered primitive data ("leaf data" in Table 1), which neither of the two methods can change.

Table 2: Distilled summary of Table 1, directly contrasting memory savings and performance impact for those methods. When compared to the existing QBVH8 our method either achieves better compression at same performance, or better performance at same compression.

"best speed" BVH (spatial splits + pre-gathered vertices)		
	QBVH8	CLBVH (fast)
mem	-4.9% – -9.4%	**-9.3% – -10.1%**
perf	**-10.7% – -19%**	-2.2% – -4.0%
"least memory" BVH (no spatial splits, indices only)			
	QBVH8	CLBVH (fast)	CLBVH(compact)
mem	-16% – -24%	-16% – -24%	**-26% – -37%**
perf	**-7.3% – -15%**	-3.7% – +.01%	-6.4% – -11%

"Least Memory" Configuration. In the "least memory" configuration total performance is already significantly lower, even for the regular uncompressed `BVH8`, in part because the loss of spatial splits triggers significantly more traversal steps and primitive intersections ("trav/isecs" in Table 1), and because without the pre-gathered vertices the primitive intersection tests become more costly. It does, however, reduce memory by roughly 2×, even for the uncompressed `BVH8`, primarily due to the significant reduction in leaf primitive data (though also due to a roughly 10% reduction in total nodes from not doing spatial splits).

Since leaves in this configuration store only indices, we can also apply the additional primitive data compression described in Section 4.5, leading to two different variants of our `CLBVH`: `CLBVH-fast` uses our `CLBVH` nodes with regular, uncompressed leaf data, while `CLBVH-compact` performs the leaf data compression described in Section 4.5.

When looking at only the `QBVH8` and `CLBVH-fast` variants, the relative memory savings of both are higher, since they are no longer as dominated by the leaf data cost. Apart from this, we see the same outcome as for the best-performance case: Generally speaking, our `CLBVH` achieves nearly exactly the same memory savings as the `QBVH8`, but at a much lower performance impact (in fact, in one case we are even slightly *faster* than the uncompressed BVH).

When comparing the `QBVH8`-compact and `CLBV-compact` methods the picture is slightly different, but still very positive. Generally speaking, `CLBVH-compact` achieves even higher compression than `QBVH8`, at still (slightly) higher performance.

We conclude that for both the best performance memory and least memory configurations, our new `CLBVH` approach is consistently better than Embree's current `QBVH8`, achieving either better performance at the same compression, or better compression at the same performance.

"Best Performance" vs "Least Memory". When comparing the fastest uncompressed `BVH8` with the least-memory compressed one we find that `CLBVH-fast` can save about a factor of 2.2× in total memory, for a roughly 30-50% drop in performance, with nearly all the performance drop due to the

lower-quality BVH, and hardly any additional cost from our method (see `CLBVH-fast` configured for best performance). For an additional roughly 5% of performance, `CLBVH-compact` can save even more memory, reaching, on average, a nearly 3× reduction in total memory.

6 SUMMARY AND DISCUSSION

In this paper, we have proposed a new variant for compressing BVHs. As with existing compressed BVH types, our `CLBVH` allows for trading performance for memory savings; however, in general our technique is either better-compressing at the same performance, or better-performing at the same compression, than Embree's `QBVH8`.

On the performance side, in particular the leaf data decompression, has not yet received as much attention as other parts of Embree, suggesting some headroom for further reducing its performance impact.

Many of the ideas outlined in this paper also have potential beyond the triangle meshes and AABB BVH nodes described in this paper. For example, oriented bounding boxes (OBBs) and hair primitives often require a large amount of memory, and often share common directions with their parents or siblings; suggesting that having multi-leaf nodes with multiple of such siblings should allow for some significant compression.

Finally, it would be interesting to evaluate how some of our memory-performance trade-offs change in a GPU or custom hardware setting.

REFERENCES

[1] Attila T. Áfra, Carsten Benthin, Ingo Wald, and Jacob Munkberg. 2016. Local Shading Coherence Extraction for SIMD-Efficient Path Tracing on CPUs. In *Proceedings of High Performance Graphics*. Eurographics Association, 119–128.

[2] Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)*. 1225–1234.

[3] Manfred Ernst and Gunter Greiner. 2008. Multi Bounding Volume Hierarchies. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*. 35–40.

[4] Vlastimil Havran. 2001. *Heuristic Ray Shooting Algorithms*. Ph.D. Dissertation. Faculty of Electrical Engineering, Czech TU in Prague.

[5] Sean Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *Proceedings of the Conference on High Performance Graphics 2014*.

[6] Jeffrey Mahovsky and Brian Wyvill. 2006. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum* 25, 2 (June 2006).

[7] S.G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and others. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)* 29, 4 (2010).

[8] Benjamin Segovia and Manfred Ernst. 2010. Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. In *Graphics Interface 2010*. 153–160.

[9] Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *Proc. of the IEEE/EG Symposium on Interactive Ray Tracing*. 49–57.

[10] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33, 4, Article 143 (2014), 8 pages.

[11] Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. ACM.