Faster Incoherent Ray Traversal Using 8-Wide AVX Instructions

Attila T. Áfra[1,2]

[1]Budapest University of Technology and Economics, Hungary
[2]Babeş-Bolyai University, Cluj-Napoca, Romania
attila.afra@gmail.com

Abstract

Efficiently tracing randomly distributed rays is a highly challenging problem on wide-SIMD processors. The MBVH (multi bounding volume hierarchy) is an acceleration structure specifically designed for incoherent ray tracing on processors with explicit SIMD architectures like the CPU. Existing MBVH traversal methods for CPUs target 4-wide SIMD architectures using the SSE instruction set. Recently, a new 8-wide SIMD instruction set called AVX has been introduced as an extension to SSE. Adapting a data-parallel algorithm to AVX can lead to significant, albeit not necessarily linear, speed improvements, but this is often not straightforward. In this paper we present an improved MBVH ray traversal algorithm optimized for AVX, which outperforms the state-of-the-art SSE-based method by up to 25%.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Monte Carlo ray tracing methods such as path tracing and photon mapping shoot a very large number of random rays with little or no coherence. Primary rays are usually highly coherent, but most secondary rays are incoherent. Because of this, the ray traversal algorithm must be optimized for *incoherent* rays in order to achieve optimal rendering performance.

Regardless of the target processor architecture, either CPU or GPU, the traversal algorithm should take advantage of the SIMD facilities of the respective hardware as much as possible. SIMD can provide a substantial speedup, but efficiently leveraging it for tracing incoherent rays is a difficult problem. The reason for this is that rays with low coherence traverse very divergent paths in hierarchical acceleration structures.

There are two main vectorization approaches for tracing mostly incoherent rays. The first tries to extract hidden coherence by sorting the rays into coherent subsets and then processing the resulting groups using SIMD operations. The advantage of this method is that the SIMD utilization is high as long as the size of the ray group is greater than or equal to the SIMD width. Unfortunately, after multiple ray bounces, this condition is true only for tree nodes close to the root, thus the overall efficiency is low. On the other hand, the second approach ignores any possible hidden coherence and uses SIMD for individual rays instead of groups. This way, SIMD utilization is completely independent of ray coherence, but the performance does not scale linearly with the SIMD width because the algorithmic efficiency decreases.

The most commonly used vector instruction set on the x86 family of CPUs is SSE (Streaming SIMD Extensions), which has a SIMD width of 4 *lanes*, where the size of a lane is 32 bits. The successor of SSE, the AVX (Advanced Vector Extensions) instruction set provides a SIMD width of 8, potentially doubling the speed of SSE. AVX was introduced only recently, in 2011, with the Intel Sandy Bridge microarchitecture. Most ray tracing algorithms have been optimized for SSE or similar 4-wide SIMD instruction sets, and thus they may not scale well for wider SIMD.

In this paper, we propose an AVX-optimized single-ray traversal algorithm for the MBVH (multi bounding volume

(a) CONFERENCE ROOM (282K triangles) (b) FAIRY FOREST (174K triangles) (c) HAIRBALL (2.9M triangles)

(d) POWER PLANT (12.7M triangles) (e) SAN MIGUEL (10.5M triangles) (f) SIBENIK (80K triangles)

Figure 1: *Scenes with varying complexity used for the performance evaluation of our ray traversal algorithm. These images were rendered using simple diffuse path tracing.*

hierarchy) acceleration structure. The MBVH enables high SIMD utilization for single-ray traversal; therefore, it is a good choice for incoherent ray tracing. Our approach offers higher path tracing performance than previous SSE-based methods for a wide variety of test cases.

2. Previous Work

Ray packet algorithms using the BVH (bounding volume hierarchy) trace hundreds of rays (typically 256–1024) together in breadth-first fashion to exploit ray coherence. Such methods try to reduce the number of intersection operations, to minimize the memory bandwidth usage, and to improve the speed of the computations using SIMD.

The ray packet traversal algorithm proposed by Wald et al. [WBS07], sometimes called *ranged traversal*, works well for primary rays, but performs poorly for secondary rays because it naively assumes high coherence and is sensitive to the order of the rays in the packet. The packet that is traced contains *SIMD rays* instead of individual rays. A SIMD ray is a small 2×2 ray packet, which is processed using 4-wide SIMD operations. If all rays are active, the SIMD utilization is optimal, which means that 4 rays can be intersected at the cost of one. However, the rays are not redistributed by the traversal algorithm, and thus the SIMD utilization quickly drops for incoherent rays. In each traversal step, the method finds the first SIMD ray that intersects the current tree node and assumes that all rays after that also intersect it. This greatly reduces the number of ray/box intersections for primary rays. The number of intersections is further reduced by employing frustum culling.

A different approach is *ray stream tracing* [WGBK07, GR08], which also traces a group of rays breadth-first, but reorders the rays on-the-fly to maximize the SIMD efficiency. The inactive rays are filtered out in each traversal step using stream compaction. Because of this, all coherence is extracted from the ray stream. If there is very little coherence, the SIMD efficiency is low. A possible way to alleviate this problem is to increase the size of the ray group. However, tracing a ray group that is too large results in inferior performance because the group does not fit entirely into the cache. Another drawback of this algorithm is that it requires hardware gather/scatter support, which is not supported by current CPU architectures.

Partition traversal [ORM08] is an improved version of ranged traversal with exact ray filtering, similar to ray stream tracing. However, it does not require special stream processing instructions because it filters SIMD rays, and instead of reordering them directly, reorders an array of ray IDs. Since the SIMD rays are constant throughout the traversal, SIMD utilization is low for incoherent rays.

A more robust method is *adaptive ray packet reordering* [BWB08], which reorders individual rays, but only when

the SIMD efficiency drops below a certain threshold. The rays are not reordered in every traversal step because it is too costly. The algorithm extracts enough hidden coherence to be superior to traditional single-ray tracing even for diffuse path tracing with a few bounces.

Another category of ray traversal algorithms use *multi-branching hierarchical acceleration structures* like QBVH [DHK08] (Quad-BVH) and MBVH [WBB08,EG08,Ern11]. These methods completely ignore coherency by tracing single rays but intersecting them with multiple bounding boxes and triangles simultaneously. This has the advantage that SIMD efficiency can be high regardless of the distribution of rays, which makes MBVH traversal algorithms particularly suitable for incoherent ray tracing. The BVH usually has a branching factor equal to the SIMD width. To speed up the intersections with triangles in the leaves, the triangles are gathered into SIMD *multi-triangles* similar to the nodes.

Tracing individual rays has the disadvantage that memory bandwidth is wasted because nodes visited by multiple rays are loaded for each ray. The *MBVH RS* (ray stream) algorithm [Tsa09] addresses this problem by combining MBVH traversal with breadth-first ray stream traversal. It traces a group of rays together, filtering out the inactive rays in every traversal step. The rays in the stream are single rays and are intersected with multi-nodes the same way as in single-ray MBVH traversal approaches. However, the leaves of the tree contain single triangles, which are intersected with small SIMD ray packets constructed on-the-fly from the active rays.

All the CPU ray tracing algorithms mentioned above were designed before the introduction of the AVX instruction set. Áfra [Áfr11] adapted the ranged and partition traversal algorithms to AVX by doubling the size of the SIMD rays. According to the measurements on a Sandy Bridge CPU, AVX typically improves ray tracing speed for primary rays by about 50% compared to SSE.

Most recently, Benthin et al. [BWW*12] presented an MBVH-based hybrid single/packet ray traversal method for the Intel MIC (Many Integrated Core) architecture. Although MIC has an even higher SIMD width than AVX (16 lanes), the approach cannot be directly applied to AVX as there are major differences between the two vector instruction sets.

3. Acceleration Structure

We use the MBVH acceleration structure for our ray tracing method because of its high performance for arbitrarily distributed rays. In this section, we present the details of our implementation of the MBVH.

The multi-nodes of an N-wide MBVH have up to N children, and store information (e.g., bounding box) about their children instead of themselves. We leverage the larger SIMD width of AVX by using 8-wide MBVHs. Most MBVH implementations have leaves referring to a single multi-triangle. Our version does not have this restriction: the leaves point to a *list* of multi-triangles.

3.1. Memory layout

There are two fundamental types of nodes in any BVH: inner nodes and leaf nodes. The inner nodes refer to their child nodes, whereas the leaf nodes refer to triangles. Both have axis-aligned bounding boxes. An inner node of an MBVH must store for each of its children a bounding box and a node index. If a child is a leaf node, a reference to the first triangle and the number of triangles are required in addition to the bounding box. Instead of defining a separate leaf node type, we keep all leaf information in the inner nodes in a compact format. Therefore, there is only a single MBVH node type.

We encode the child node information with a 32-bit integer value called *node ID*. For inner nodes, this is simply the node index, which is a non-negative integer. If a node is a leaf, the node ID encodes the index of the first multi-triangle and the multi-triangle count. The sign bit is used to distinguish between an inner and a leaf node ID. Leaf node IDs have their sign bit set, thus they are negative values. The lower 29 bits of the leaf node ID contain the triangle index, and the remaining 2 bits contain the count. This representation is compact, can be quickly decoded, and simplifies the traversal algorithm.

The multi-node first contains N bounding boxes in structure-of-arrays (SoA) form (i.e., a *multi-box*), which is necessary for fast loading into SIMD registers. If there are less than N children, the empty lanes are filled with invalid boxes. Next, N node IDs are stored. Finally, the data structure is padded to be aligned on a cache line boundary (i.e., 64 bytes).

```
template <int N>
struct Node {
  float minBound[3][N];
  float maxBound[3][N];
  int ids[N];
  int padding[N];
};
```

The node size for MBVH4 is 128 bytes, and for MBVH8 it is 256 bytes.

The multi-triangles, similar to the bounding boxes in the nodes, have a fixed-size SoA layout to facilitate SIMD processing. The triangle data is pregathered to maximize intersection performance at the cost of higher memory usage.

3.2. Construction

For the construction of the MBVH, we use the top-down greedy splitting technique by Wald et al. [WBB08]. This is a simple algorithm producing trees with high average branching factors. We split the nodes with the high-quality SBVH

(Split Bounding Volume Hierarchy) algorithm [SFD09], which uses spatial splits in addition to object splits. The splitting costs are computed using the surface area heuristic (SAH) [Hav00].

Initially, the root multi-node contains a single child node. We build the multi-nodes by repeatedly splitting its child nodes using simple binary splits. In each step, we first select the child node with the largest surface area. Then, we split this node and replace it with the resulting two nodes. We stop splitting either if the number of children in the multi-node is equal to the maximum branching factor N, or if there are no more splitable nodes according to the SAH. We call this type of splitting *horizontal splitting*. For this we use a modified SAH cost function:

$$C_H = \frac{SA(B_L)}{SA(B)} \left\lceil \frac{T_L}{N} \right\rceil + \frac{SA(B_R)}{SA(B)} \left\lceil \frac{T_R}{N} \right\rceil, \qquad (1)$$

where N is the SIMD width, $SA(B)$ is the surface area of a bounding box, T_L and T_R are the number triangles in the left and right child, and B_L, B_R, and B are the bounding boxes of the left child, right child, and parent node. Note that we have omitted the traversal and triangle intersection costs because horizontal splitting does not increase the number of traversal steps.

After building a multi-node with the algorithm above, we split its children to create child multi-nodes, which we call *vertical splitting*. After splitting a node, we create a new multi-node containing the two children. Then, we recursively build the resulting multi-nodes, starting again with horizontal splitting. The SAH cost function for vertical splitting is the following:

$$C_V = K_T + K_I \left(\frac{SA(B_L)}{SA(B)} \left\lceil \frac{T_L}{N} \right\rceil + \frac{SA(B_R)}{SA(B)} \left\lceil \frac{T_R}{N} \right\rceil \right), \qquad (2)$$

where K_T is the cost of a traversal step and K_I is the cost of a multi-triangle intersection. In our implementation, we set both costs to 1.

For both splitting strategies, we stop splitting a node if the SAH cost of splitting is greater than making the node a leaf. We also stop if the number of triangles in the node is 1. An alternative termination criterion would be to stop when there are N or less triangles (the intersection cost is the same), but this produces higher cost trees.

We limit the leaf size to 2N triangles. Thus, we always split a node if the number of triangles exceeds this value.

4. Ray Traversal Algorithm

In our algorithm, we trace rays in batches (e.g., 256 rays), but we trace them individually. This way, hidden coherence cannot be exploited, but it is possible to implement superior ordered traversal, which improves performance independently from coherence. The purpose of ordered traversal is to find the closest intersection in less steps [SKHBS02]. Ordered BVH traversal cannot stop at the first valid intersection like kd-tree traversal, but it still skips many subtrees.

Ray type	1	2	3	4	5	6	7	8
primary	47.0	31.5	14.6	5.8	0.9	0.1	0.0	0.0
AO	56.5	25.5	12.6	4.0	1.1	0.2	0.0	0.0
1-bounce	54.7	25.4	13.4	4.7	1.4	0.3	0.0	0.0
2-bounce	54.5	25.7	13.4	4.8	1.4	0.3	0.0	0.0
8-bounce	53.2	26.1	13.9	5.0	1.4	0.3	0.1	0.0

Table 1: *The distribution (in %) of the number of nodes hit by rays in MBVH8 multi-nodes for primary, ambient occlusion (AO), 1-bounce, 2-bounce, and 8-bounce diffuse rays. Only multi-nodes with at least one child hit were considered. The statistics were collected for the* SAN MIGUEL *scene, but the results are very similar for other scenes. Note that for more than 90% of the ray/multi-node intersections, only 1–3 nodes are hit.*

When a ray encounters a multi-node during the traversal loop, it is intersected with the N bounding boxes stored in the node. This is done using the SIMD version of the standard slabs test [KK86], which computes a hit mask and the intersection distances.

4.1. Ordered traversal

Similarly to the binary BVH traversal algorithm [WBS07], we traverse the MBVH with depth-first ordered traversal, which needs a traversal stack. We use the intersection distances provided by the box test algorithm to determine the traversal order of the intersected child nodes. A straightforward way to achieve this is to do horizontal SIMD sorting [FAN07], which unfortunately is quite costly and has suboptimal SIMD efficiency. Because of its high cost, some previous single-ray MBVH traversal approaches employed faster but less accurate ordering methods instead [EG08, DHK08].

If there are less than N intersected nodes, SIMD sorting is even less efficient because it always sorts N values. We have measured the distribution of the number of nodes hit for different types of rays, which can be seen in Table 1. The results show that for about 90% of the valid multi-node intersections, only 1–3 children are hit. A single hit is the most likely (40%–60%). Therefore, SIMD sorting almost always works at a very low efficiency, especially for wide branching factors.

We sort the nodes with a scalar method optimized for a low number of hits, which was introduced in the Intel Embree ray tracer for 4-wide MBVH ray traversal [Ern11]. This is significantly faster than SIMD sorting. The main idea of the method is to use specialized implementations of sorting for the most frequent numbers of hits. We extend the original algorithm to handle wider trees by implementing the following cases: 1, 2, 3, 4, and 5–N hits.

We first convert the SIMD hit mask produced by the hit test to an integer bit mask, where a 1 bit indicates a hit, and

Scene	Tris	BVH	Nodes	Node util	Tri util
CONFERENCE ROOM	282K	4-way	32K	92%	89%
		8-way	11K	80%	72%
FAIRY FOREST	174K	4-way	18K	94%	85%
		8-way	6K	82%	69%
HAIRBALL	2.9M	4-way	671K	93%	94%
		8-way	152K	87%	77%
POWER PLANT	12.7M	4-way	1.3M	93%	89%
		8-way	425K	79%	77%
SAN MIGUEL	10.5M	4-way	999K	94%	86%
		8-way	341K	78%	71%
SIBENIK	80K	4-way	10K	96%	82%
		8-way	3K	86%	65%

Table 2: *MBVH tree statistics for the test scenes: number of triangles, branching factor of the MBVH, number of multi-nodes, SIMD utilization of multi-nodes, and SIMD utilization of multi-triangles.*

its position corresponds to the index of the node. If this mask is equal to zero, no nodes were hit. An important part of the algorithm is to quickly get the indices of the hit nodes. This can be done with two fast CPU instructions: *bit scan forward* (BSF) and *bit test and complement* (BTC). BSF returns the index of the first hit node (i.e., the position of the least significant set bit). In order to compute the next index, we first have to remove the previously found hit from the mask by clearing the corresponding bit using BTC. We repeat these two steps until the mask becomes zero.

The algorithm consists of 5 main steps, each corresponding to the different sorting cases. In each step, we extract a hit index from the mask; except the last step, where we may extract a variable number of hits. If the mask becomes zero, we sort all the identified nodes so far with a fast specialized sorting technique. The ID of the node with the closest intersection is put into a variable indicating the next node to traverse. The remaining node IDs are pushed onto the traversal stack. This avoids the overhead of pushing and almost immediately popping the closest child. The traversal stack also contains the intersection distances, which are used to skip nodes beyond the closest hit.

The sorting cases are the following:

- **1 hit**: The traversal continues with the hit node.
- **2 hits**: The two intersection distances are compared. The far node is pushed on the stack, and the traversal continues with the near node.
- **3 hits**: For 3 or more hits, the 3 hits found so far are pushed on the stack without sorting. If there are exactly 3 hits, the nodes are sorted on the stack using a *sorting network* [Bat68] consisting of 3 comparators. Next, the top node is popped from the stack, and the traversal continues with that node.
- **4 hits**: The next node is pushed on the stack, and the 4 nodes are sorted using a sorting network with 5 comparators.
- **5–N hits**: All remaining nodes are pushed on the stack in a loop. Then, the nodes are sorted using *insertion sort*, which is efficient for very small data sets. This part of the algorithm is the slowest, but it is rarely executed.

The following pseudocode implements the main part of the ordered traversal algorithm:

```
// 1
int i = bsf(mask); mask = btc(mask, i);
if (mask == 0) {
  nodeID = node->ids[i]; goto traverse;
}
// 2
int i2 = bsf(mask); mask = btc(mask, i2);
if (mask == 0) {
  if (dist[i] < dist[i2]) {
    stack.push(node->ids[i2], dist[i2]);
    nodeID = node->ids[i];
  } else {
    stack.push(node->ids[i], dist[i]);
    nodeID = node->ids[i2];
  }
  goto traverse;
}
// 3
stack.push(node->ids[i], dist[i]);
stack.push(node->ids[i2], dist[i2]);
i = bsf(mask); mask = btc(mask, i);
stack.push(node->ids[i], dist[i]);
if (mask == 0) {
  stack.sortTop3();
  nodeID = stack.pop(); goto traverse;
}
// 4
i = bsf(mask); mask = btc(mask, i);
stack.push(node->ids[i], dist[i]);
if (mask == 0) {
  stack.sortTop4();
  nodeID = stack.pop(); goto traverse;
}
// 5-N
int oldSize = stack.size();
do {
  i = bsf(mask); mask = btc(mask, i);
  stack.push(node->ids[i], dist[i]);
} while (mask != 0);
stack.sortTopN(stack.size() - oldSize + 4);
nodeID = stack.pop(); goto traverse;
```

In this code, `nodeID` is the ID of the current multi-node that is being traversed, `node` is a pointer to this multi-node in the memory, and `dist` is a SIMD array containing the intersection distances for each child node.

If we do not need the closest intersection of the ray with the geometry (e.g., for shadow and ambient occlusion rays), we stop the traversal at the first valid intersection with a tri-

Scene	Ray type	Single traversal			Stream traversal		
		MBVH4 Mray/s	MBVH8 Mray/s	Speedup ×	MBVH4 Mray/s	MBVH8 Mray/s	Speedup ×
CONFERENCE ROOM	primary	32.3	39.5	1.22	36.0	31.0	0.86
	AO	46.7	52.4	1.12	42.9	37.6	0.88
	1-bounce	26.2	28.8	1.10	24.2	21.0	0.87
	2-bounce	25.4	27.6	1.09	21.9	19.4	0.89
	8-bounce	24.6	26.2	1.07	18.9	17.1	0.90
FAIRY FOREST	primary	27.1	32.8	1.21	29.9	26.8	0.90
	AO	29.7	34.8	1.17	27.9	26.3	0.94
	1-bounce	19.1	22.3	1.17	17.8	16.6	0.93
	2-bounce	19.6	22.4	1.14	16.7	15.9	0.95
	8-bounce	19.3	21.9	1.13	15.1	14.4	0.96
HAIRBALL	primary	16.8	20.6	1.23	19.9	18.1	0.91
	AO	9.2	11.0	1.20	7.9	8.2	1.04
	1-bounce	7.2	8.6	1.20	6.1	6.1	1.00
	2-bounce	6.6	7.8	1.19	5.2	5.0	0.98
	8-bounce	6.0	7.1	1.19	3.8	3.5	0.92
POWER PLANT	primary	40.0	44.0	1.10	40.4	33.2	0.82
	AO	24.4	29.5	1.21	20.9	20.5	0.98
	1-bounce	21.6	25.0	1.15	18.7	17.6	0.94
	2-bounce	19.2	22.4	1.16	15.3	14.4	0.94
	8-bounce	17.6	19.4	1.11	10.7	10.6	0.99
SAN MIGUEL	primary	13.6	16.7	1.23	15.3	14.2	0.93
	AO	10.3	12.0	1.17	8.5	8.5	1.00
	1-bounce	7.6	9.1	1.19	6.5	6.2	0.95
	2-bounce	7.2	8.6	1.20	5.2	5.2	1.00
	8-bounce	6.5	8.1	1.25	4.0	3.9	0.97
SIBENIK	primary	32.7	36.3	1.11	34.5	28.1	0.81
	AO	36.6	41.1	1.12	33.6	31.3	0.93
	1-bounce	19.7	21.6	1.10	16.5	15.2	0.92
	2-bounce	18.8	20.4	1.09	14.0	13.1	0.94
	8-bounce	17.8	18.1	1.02	11.1	10.6	0.96

Table 3: *Ray tracing performance in million rays per second (Mray/s) for primary, ambient occlusion (AO), and diffuse rays with up to 1, 2, and 8 bounces. The AO rays were long, and the traversal stopped at the first intersection. The diffuse rays were generated using path tracing without Russian roulette, and the number of bounces indicate the maximum ray recursion depth. We have compared two main algorithms: single-ray traversal and stream traversal (MBVH RS). The MBVH4 versions use SSE, whereas the MBVH8 versions use AVX. Our single-ray MBVH8 traversal method delivers the highest performance for all test cases. The timings do* not *include ray generation and shading. The speedups from AVX over SSE are also listed. The scenes were rendered from the viewpoints shown in Figure 1 at* 1024×768 *resolution. The CPU used was an Intel Core i7-3770.*

angle. Ordered traversal is not necessary in this case. Instead, we push the nodes on the stack without any sorting.

5. Results

We compared our method with MBVH4 single-ray traversal (the fastest algorithm used in Embree) and also with MBVH RS traversal, which, unlike the other methods, is able to extract hidden coherence from rays. Our implementation of MBVH RS uses multi-triangles instead of single triangles in order to maximize SIMD utilization for highly incoherent rays and wide SIMD units. We tested this method with both 4-way and 8-way branching MBVHs.

We measured the traversal performances for different types of rays and scenes. We generated random diffuse rays with 1-bounce, 2-bounce, and 8-bounce path tracing. Russian roulette was not used to terminate paths. We also tested the methods with primary and ambient occlusion rays. The ambient occlusion rays were quite long (one-eighth of the scene size). The rays were traced in tiles of 16×16 pixels, and the rendering resolution was 1024×768. The test scenes were CONFERENCE ROOM, FAIRY FOREST, HAIRBALL, POWER PLANT, SAN MIGUEL, and SIBENIK (see

Scene	BVH	Mray/s	N_T/ray	N_I/ray	KB/ray
CONFERENCE ROOM	4-way	24.6	9.6	2.6	1.68
	8-way	26.2	5.9	2.4	2.35
FAIRY FOREST	4-way	19.3	13.2	3.7	2.34
	8-way	21.9	7.8	3.0	3.09
HAIRBALL	4-way	6.0	30.1	7.8	5.22
	8-way	7.1	17.3	7.4	7.12
POWER PLANT	4-way	17.6	13.9	2.2	2.16
	8-way	19.4	8.8	1.8	2.89
SAN MIGUEL	4-way	6.5	24.2	5.0	3.97
	8-way	8.1	14.7	4.2	5.24
SIBENIK	4-way	17.8	13.0	2.6	2.12
	8-way	18.1	8.1	2.4	2.91

Table 4: *Ray traversal statistics of the single-ray MBVH4 and MBVH8 algorithms for 8-bounce diffuse rays: million rays per second (Mray/s), number of traversal steps (N_T/ray), number of multi-triangle intersections (N_I/ray), and amount of accessed scene data in kilobytes per ray (KB/ray). Note that MBVH8 requires less traversal steps and intersections, but more memory traffic.*

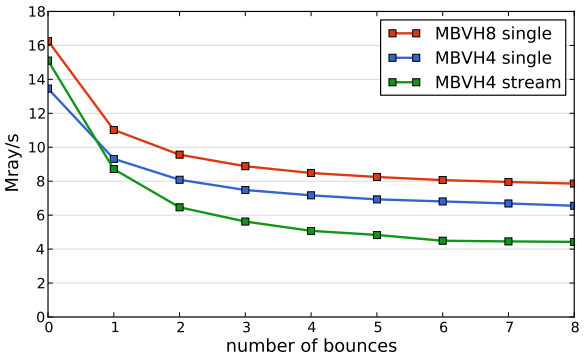

Figure 2: *N-bounce diffuse path tracing performance (including primary rays and shading) for the* SAN MIGUEL *scene for the fastest three methods. MBVH8 single traversal (our proposed method) outperforms both MBVH4 single traversal (the Embree method) and MBVH4 stream traversal (MBVH RS).*

Figure 1). The statistics for the constructed MBVH trees are shown in Table 2.

Our benchmark system had an Intel Core i7-3770 processor (Ivy Bridge, 4 cores, 8 threads, 3.4 GHz, 8 MB L3 cache) and 16 GB RAM (DDR3-1600, dual channel). The algorithms were implemented in C++ with SIMD intrinsics and were compiled for 64 bits with Intel C++ Compiler XE 13.0. We used SSE for the MBVH4 traversal methods and AVX for the MBVH8 ones.

The performance results in million rays per second are listed in Table 3. Our method, MBVH8 single traversal implemented with AVX, yields a speedup of 2–25% relative to MBVH4 for the tested ray types and scenes. To understand where this improvement comes from (and why it is not higher), we have collected some ray traversal statistics for both algorithms in Table 4. We can observe that MBVH8 decreases the number of traversal steps and triangle intersections by a factor of 1.52× on average. However, the actual speedup is lower than this value for two main reasons. First, the traversal steps and intersections are slightly more expensive than for MBVH4 because of the wider sorts and reductions. Second, the amount of accessed scene data is *increased* by about 1.35×, which results in more cache misses.

In contrast with single traversal, the AVX implementation of MBVH RS is typically slightly *slower* than the SSE one. This is primarily caused by the high cost of 8-wide SIMD sorting.

Comparing the two MBVH4 traversal algorithms, we see that MBVH RS is faster than the optimized single-ray approach for primary visibility, by at most 18%. Nevertheless, single traversal is the most efficient for all other ray types, including ambient occlusion rays, which are somewhat coherent. The biggest difference is for rendering the POWER PLANT scene (one of our most complex test scenes) with 8-bounce path tracing, where single traversal is 64% faster for diffuse rays.

Our MBVH8 traversal method is faster than MBVH RS in all our tests, including primary ray benchmarks (see Figure 2). For 8-bounce diffuse rays, its performance is 38–100% higher. Thus, according to these results, wide-SIMD single-ray traversal is significantly more efficient for incoherent ray tracing than ray stream traversal on the latest CPU architectures.

6. Conclusions and Future Work

In this paper we have presented an improved MBVH ray traversal algorithm optimized for incoherent rays and the AVX instruction set. Our method outperforms previous incoherent ray traversal algorithms on the CPU for all tested scenes. Also, we have demonstrated that AVX can provide a notable performance boost even for highly random rays.

As future work, it would be interesting to combine our approach with packet/stream traversal to attain optimal performance for both coherent and incoherent ray distributions, similarly to [BWW*12]. We also plan to investigate incoherent ray tracing on Intel Xeon Phi, a coprocessor based on the MIC architecture.

Acknowledgements

This work was possible with the financial support of the Sectoral Operational Programme for Human Resources Development 2007-2013, co-financed by the European Social Fund, under the project number POSDRU/107/1.5/S/76841

with the title *Modern Doctoral Studies: Internationalization and Interdisciplinarity*.

The test scenes are courtesy of Anat Grynberg and Greg Ward (CONFERENCE ROOM), Ingo Wald (FAIRY FOREST), Samuli Laine and Tero Karras (HAIRBALL), University of North Carolina at Chapel Hill (POWER PLANT), Guillermo M. Leal Llaguno (SAN MIGUEL), and Marko Dabrovic (SIBENIK).

References

[Áfr11] ÁFRA A. T.: Improving BVH ray tracing speed using the AVX instruction set. In *Eurographics 2011 - Posters* (Llandudno, UK, 2011), Eurographics Association, pp. 27–28. 3

[Bat68] BATCHER K. E.: Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 307–314. 5

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive ray packet reordering. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008* (Los Alamitos, CA, USA, 2008), IEEE Computer Society, pp. 131–138. 2

[BWW*12] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W.: Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (September 2012), 1438–1448. 3, 7

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2008), 1225–1233. 3, 4

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008* (2008), pp. 35–40. 3, 4

[Ern11] ERNST M.: Embree: Photo-realistic ray tracing kernels. *SIGGRAPH 2011 Talk* (2011). 3, 4

[FAN07] FURTAK T., AMARAL J. N., NIEWIADOMSKI R.: Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2007), SPAA '07, ACM, pp. 348–357. 4

[GR08] GRIBBLE C. P., RAMANI K.: Coherent ray tracing via stream filtering. In *2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (August 2008), pp. 59–66. 2

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, November 2000. 4

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 269–278. 4

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time whitted ray tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008* (2008), pp. 41–48. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of High-Performance Graphics 2009* (2009), Eurographics Association, pp. 7–14. 4

[SKHBS02] SZIRMAY-KALOS L., HAVRAN V., BENEDEK B., SZÉCSI L.: On the efficiency of ray-shooting acceleration schemes. In *Proceedings of Spring Conference on Computer Graphics (SCCG)* (2002), pp. 97–106. 4

[Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-BVH ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 151–158. 3

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets – efficient SIMD single-ray traversal using multi-branching BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008* (2008), pp. 49–57. 3

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007). 2, 4

[WGBK07] WALD I., GRIBBLE C. P., BOULOS S., KENSLER A.: *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. Rep. UUSCI-2007-012, SCI Institute, University of Utah, 2007. 2