# Getting Rid of Packets
## – Efficient SIMD Single-Ray Traversal using Multi-branching BVHs –

Ingo Wald[◇] Carsten Benthin[◇] Solomon Boulos[†]
[◇] Intel Corporation [†]Stanford University

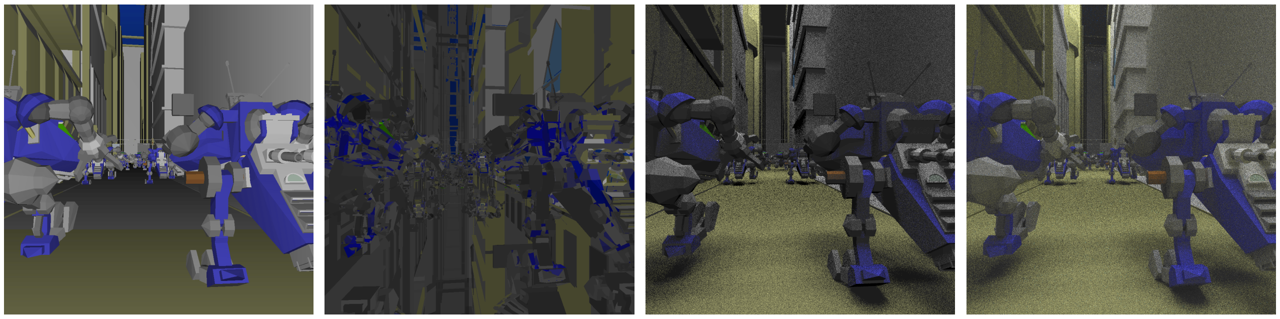
Figure 1: The BART robots scene (71.7K triangle, 1 quad light), rendered with primary rays only, forced 2-bounce reflections, soft shadows (16 light samples), and a 2-bounce path tracer (16 samples per pixel). Though slower than aggressive packet/frustum techniques for the primary ray case, our single-ray based method is more efficient for the less coherent soft shadows and path traced images.

## Abstract

While contemporary approaches to SIMD ray tracing typically rely on traversing packets of coherent rays through a binary data structure, we instead evaluate the alternative of traversing individual rays through a bounding volume hierarchy with a branching factor of 16. Though obviously less efficient than high-performance packet techniques for primary rays, we demonstrate that for less coherent secondary ray distributions this approach is at least competitive with (and often faster than) typical packet traversal techniques.

## 1 Introduction

Commodity hardware architectures continue to offer more compute performance every year, but increasingly rely on thread parallelism and ever wider SIMD units to deliver that performance. For ray tracing, exploiting thread parallelism is straightforward using screen-space parallelization. Exploiting SIMD units, however, is more complicated, and requires changing the core algorithms.

The standard technique for using SIMD in interactive ray tracing is to generate, trace, and shade *packets* of rays. For coherent rays (i.e., rays that want to traverse the same nodes and intersect the same primitives), packet tracing usually achieves both very high SIMD utilization and bandwidth reductions – both of which are crucial to exploiting the hardware's full potential. Packet techniques have been shown to work well for a wide variety of acceleration structures and primitive types; and have also been shown to not be restricted to primary rays, but to also work for (reasonably coherent) shadow and reflection rays.

On the flip-side, packet techniques become inefficient as the rays becomes less coherent. Even if there *is* coherence, the manner in which rays are grouped into packets has a crucial impact on performance. For example, consider computing soft shadows with 64 samples per light for a 64 ray input packet. To name only the most straightforward approaches, one could either shoot a separate 64-ray packet from each of the 64 surface points or connect all 64 surface points to one light sample each (and iterate 64 times to produce 64 light samples). Alternatively, one could trace a giant packet of $64 \times 64 = 4K$ rays. Other groupings make sense, too, but already the complications are manifest (we have not even discussed what to do when the incoming ray packet hit differing geometry or shaders).

In addition, packet tracing requires a large amount of regularity that complicates the actual shader/renderer code. For example, the importance sampling techniques used by many high-quality renderers choose a varying number of light samples for every surface sample, with some light sources receiving only a single sample (or none at all), while others receive lots of samples. For any technique that relies on "coherent" packets of exactly the "right" number of rays, supporting this kind of irregularity is, to say the least, an unsolved problem. It should be no surprise that packets remain unused in high-quality offline renderers.

Eventually, it *might* turn out that rendering algorithms like path tracing will simply not perform well on SIMD architectures regardless of modifications. On the other hand, it is also possible that packet tracing will, in practice, be "good enough" for the rendering algorithms that the average user will run on such hardware architectures. That notwithstanding we believe it is important to not simply hope for the best, but rather to reconsider alternatives to relying on packets—even if such techniques are likely to be slower for cases where packet tracing excels.

In this paper, we investigate the use of bounding volume hierarchies with branching factors equal to the architecture's SIMD width of a wide-SIMD hardware architecture (in this paper, we assume a SIMD with of 16). Instead of relying on packets, we trace every ray individually, and exploit SIMD parallelism by always testing every ray against 16 nodes or 16 triangles. We demonstrate that with a properly built bounding volume hierarchy (BVH) and a front-to-back traversal algorithm, this approach is somewhat slower than aggressive packet techniques for primary rays, but that it is at least competitive with packet techniques soon as packet coherence drops, while at the same time restoring a renderer's ability to trace individual and possibly incoherent rays.

## 2 Background

**SIMD Architectures.** Modern high-performance hardware architectures feature two distinct features: parallelism through many cores/execution units, and a SIMD-way of execution inside each core. The number of cores typically is in the few dozens (e.g., 16 cores on a 4-way 4-core Harpertown workstation, and 16 "cores" on a GTX8800. Current SIMD width for CPUs is 4 [9] (increasing to 8 in the near feature [10]) and recent GPUs provide an even higher SIMD width using their parallel floating point units. For the remainder of this paper, we will assume a SIMD width of 16, though the same approach would also fit other SIMD widths (a similar approach for a 4-wide SIMD architecture has been concurrently investigated in [4]).

**SIMD Packet Tracing.** Exploiting SIMD means performing the same basic operation on multiple data elements. In ray tracing, the most common SIMD operations are node traversal, primitive intersection, and shading. One of the most widely used techniques for SIMD ray tracing today is packet tracing, in which all these operations – traversal, intersection, and shading – are executed in parallel on a set of coherent rays called a "ray packet". Packet tracing was originally proposed by Wald et al. [28] for triangular scenes and kd-trees, but has since been applied to a wide variety of primitive types, acceleration structures, and hardware architectures.

For shading, this way of processing the hit points/fragments seems quite natural; it is exactly what current GPUs do, too, and in practice it seems to work rather well. For node traversal and triangle intersection, always performing the same operation on different rays is more tricky. As rays diverge they choose to follow differing paths down hierarchical data structures, especially if the rays are traversing different regions of the scene. This is similar to branch divergence in shaders, however, with a deep acceleration structure the number of branches compounds quickly. Nevertheless, current high-performance ray tracers assume that the amount of branch divergence is low enough that SIMD packet tracing will provide sufficient benefit.

**Larger Packet Techniques.** SIMD packet tracing can only provide as much benefit as the SIMD width of the machine. In addition to these hardware-related benefits, using packets larger than the SIMD width also allows for algorithmic benefits that actually avoid certain operations, typically by performing some conservative full-packet rejection tests based on bounding frusta, bounding rays, or interval arithmetic [5, 19, 20, 24, 26]. All of these techniques rely on high ray coherence to deliver benefits over SIMD packet tracing.

**Coherence Techniques.** As the distribution of rays becomes incoherent, SIMD packet tracing quickly suffers from low SIMD utilization, eventually using only a fraction of the hardware's potential. For large packet techniques, incoherent rays can lead to performing more computations than single ray code would have performed. The problem with packet techniques is that they do not answer the question where the coherent packets come from in the first place, and thus rely on the shading/rendering engine to produce good packets—which more often than not means they have to have exactly the right size (big enough to provide gains, yet small enough to avoid overhead), and a sufficient amount of coherence.

Generating packets of coherent rays is trivial for primary rays, and maintaining the primary rays' grouping for all future generations of these rays has been shown to work reasonably well at least for hard shadows and perfect reflections [22]. By carefully determining the order in which rays are cast, packet tracing can also be used to compute indirect diffuse illumination [27], as well as Whitted and Distribution Ray Tracing [1]. Reshetov [18] has shown that after several bounces of perfectly specular reflections packet utilization drops to essentially a single valid ray per packet, and for diffuse bounces or ambient occlusion rays, this frequently happens after even only one or two bounces [25].

Trying to increase the coherence of a given ray distribution was investigated as early as Pharr et al.'s Memory Coherent Ray Tracing [15], where rays from a path tracer were reordered into a more coherent sequence by stepping them through a coarse scheduling grid. While highly successful for reducing disk I/O in out-of-core ray tracing, it is not clear if the reordering overhead would pay off for extracting SIMD parallelism. In [25], it was shown that SIMD utilization can be improved by tracing large packets in breadth-first manner, and successively discarding rays that become inactive. Mansson et al. [13] proposed reordering rays via grouping them into larger batches than their packet size, and then shooting packet sized subsets of this batch in sequence. None of their proposed heuristics performed better than SIMD packet tracing in final rendering performance. Similar to Pharr's approach, Navratil et al. [14] also proposed reordering based on the acceleration structure. The focus was directed at geometry and ray bandwidth only, and no comparisons were made with respect to absolute performance or reducing computational costs such as ray-triangle tests. The drawback of all these reordering approaches is that they require to queue up large numbers of rays from which the coherence is to be extracted—which is likely to be problematic on high-throughput architectures with small cache sizes.

**Node-/Triangle-Parallel SIMD Ray Tracing.** The obvious alternative to tracing packets is to trace single rays, and always intersect this individual ray with *N* different nodes/triangles. In fact, the first approaches towards SIMD ray tracing that we are aware of [17] did exactly that, by building kd-trees whose cost function was skewed to favor large leaves with triangles close to the architecture's SIMD width, and then intersecting *N* triangles at once. This idea however was then abandoned in favor of packet techniques because of three reasons: first, kd-trees (the favorite data structure at that time) favor small leaves, and perform badly for large leaves; second, because kd-trees are intrinsically binary the same idea cannot be used for traversal; and third, the approach would not have given any benefit for shading and ray generation. For these reasons, Wald et al. [28] then argued that a more efficient way of using SIMD in a (kd-tree based) ray tracer is to trace, intersect, and shade packets of rays.

**Multi-branching BVHs and Single-Ray SIMD Intersection.** The arguments against node-parallel ray tracing may be true for kd-trees, but do not apply to BVHs. BVHs naturally support higher branching factors, and the first BVH based ray tracers did, in fact, use BVHs with a variable number of children per node [6, 21]. Even the idea of testing multiple BVH nodes in parallel has been proposed before [3], though not in a real-time context.

Concurrently to this paper, the idea of BVHs with branching factors higher than two has also been investigated by Dammertz et al. [4]; we will come back to their approach–and contrast it with ours–in Section 6.4.

## 3 Method Overview

In this paper, we investigate the use of BVHs with branching factors of up to 16 (the SIMD width of our target architecture). For inner nodes, we test 16 nodes in SIMD; for leaf nodes, we intersect triangles in batches of 16. Due to this SIMD processing, we use a SIMD-friendly data layout in which all 16 children of the same node are stored in a structure-of-array (SoA) "multi-node" layout. Note that this forces every multi-node to contain 16 node slots even if the parent node has less than 16 children; any unused nodes are flagged as invalid. The actual true hierarchy is governed by a surface area heuristic that takes the modified traversal and intersection cost into account. The resulting trees have an average branching factor and leaf size of 10–12 each (the most often traversed nodes high up in the tree usually are completely filled), and produce high SIMD utilization even for single-ray traversal.

We also introduce an efficient SIMD traversal technique for these BVHs that produces a strict front-to-back traversal with a minimum of scalar code to determine the traversal order. We acknowledge that this approach is slower for primary rays for which aggressive packet techniques excel, but demonstrate that it performs better than those techniques as soon as incoherent rays like soft shadows or path tracing are considered.

Though in principle also applicable to SIMD widths other than 16, our implementation is designed for a specific hardware architecture, and is currently hard-coded for a SIMD width of 16.

## 4 Building Good Multi-branching BVHs

Though an efficient traversal routine is key to performance, the actual way the data structure is built often can have a similar performance impact. To be able to evaluate the quality of a multi-BVH, we first have to derive a cost function, for which we can use a slightly modified form of the surface area heuristic (SAH). This heuristic in fact was originally introduced for BVHs with arbitrary branching factors, and thus works quite well for our purposes.

The only difference to the standard SAH is that the cost function for leaves and inner nodes is slightly different. Since we always perform 16 triangle tests respectively 16 axis-aligned box tests in parallel for the same ray, intersecting 16 triangles in a leaf now costs roughly as much as intersecting a single triangle; the same argument holds true for box tests in inner nodes. Thus, the cost for intersecting a leaf $L$ with $N(L)$ triangles is proportional to the number of "chunks" of 16 triangles,

$$C(L) = K_{tri}\lceil\frac{N(L)}{16}\rceil,$$

where $K_{tri}$ is a constant that models the cost of a SIMD triangle test, and which we will subsequently ignore. As our branching factor is equal to the SIMD with, the cost for an inner node is constant independent of the number of children.

Following the traditional assumptions for the surface area heuristic [6, 7], the probability of any node $n$ being traversed by a random ray is proportional to the node's surface area $SA(n)$. Thus, the aggregate cost in SIMD triangle intersections and SIMD box tests for a given multi-BVH can be estimated as

$$C_{tri} = \sum_{leaves\ L} SA(L)\lceil\frac{N(L)}{16}\rceil \quad \text{and} \quad C_{box} = \sum_{multi-nodes\ M} SA(M).$$

Based on this global cost estimate, we can evaluate the quality of any multi-BVH we produce.

### 4.1 Splitting vs Collapsing

One obvious choice for building multi-branching BVHs is to use the original Goldsmith-Salmon approach of successive merging of nodes. However, for binary BVHs this is known to produce rather poor BVHs, and we could not find any convincing argument that this should be different for multi-branching BVHs. We have therefore not investigated this technique at all.

For binary BVHs, the best known build techniques operate by recursively partitioning a node in top-down fashion until some set of criteria indicates that a leaf should be built. For our multi-BVH, the situation gets a bit more complicated, primarily because the cost of leaf and node intersection are now piecewise constant. Eventually, we see two promising ways of building multi-BVHs: First, one can build a binary BVH, and successively collapse it into a multi-way BVH; and second, one could use a top-down approach that successively splits in breadth-first order (i.e., always split the "biggest" node), and that stores a multi-node every time 16 nodes have been generated.

We originally believed collapsing to be the simpler approach, so we focus on that first. To do so, we take an existing high-quality BVH builder (based on SAH-based binning), build a binary BVH, and copy this binary BVH into our multi-node layout (i.e., every multi-node has exactly 2 valid and 14 invalid children), which we can then collapse.

When collapsing a binary BVH, the simplest method is to collapse from the leaves up, always merging every pair of leaves as long as they together have less than triangles. Though trivial to implement, this turns out to be a bad idea. For example, if two nodes $A$ and $B$ have less than 16 triangles when combined (say, $N_A = 6$ and $N_B = 7$) then all $N_A + N_B = 13$ triangles can be intersected for the same cost as a single triangle. While this simple comparison would suggest merging the two nodes, the *probability* of intersecting the combined node could actually be much higher than before (in the case that $SA(A \cup B) > SA(A) + SA(B)$).

### 4.2 SAH-based Collapsing

Instead of collapsing naïvely we opted for a more sophisticated approach: we determine a set of three tree transformations (merging a child node into the parent node, merging two leaf-node children, and merging two inner-node children), and greedily execute these techniques based on the technique's impact on expected tree cost.

**Merging a child node into its parent.** We iterate over all the children $c_i$ of a node $n$, and determine if we could adopt that node's children as our own (i.e., whether $N(n) - 1 + N(c_i) <= 16$). If this merge operation is possible, the change in SAH is exactly $SA(c_i)$, since neither $c_i$'s children nor $SA(n)$ change at all, while $c_i$ disappears. We determine the child with highest possible gain, and, if found, collapse it. As merging a child always produces a positive gain, we perform this technique as often as possible.

**Merging two leaf-node children.** Once no child can be merged into the current node, we examine every pair $(c_i, c_j)$ of leaf-node children, and determine the impact of potentially merging them. The change in tree cost is the cost of the previous configuration ($SA(c_i)\lceil\frac{N(c_i)}{16}\rceil + SA(c_j)\lceil\frac{N(c_j)}{16}\rceil$) less the cost of the merged node ($SA(c_i \cup c_j)\lceil\frac{N(c_i)+N(c_j)}{16}\rceil$). We determine the pair with the highest (positive) gain, and, if found, merge it.

**Merging two inner-node children.** In some cases, the previous two techniques can get stuck in a situation where neither can execute. In fact, we found cases where a certain node has 16 children (i.e., merging a child is not possible), each of which had only two leaves that the above heuristic suggested should not be merged. In this case, it would still make sense to have two 16-leaf children, thus potentially allowing the parent node to be merged upwards.

To help that to happen, we have added a technique that looks at every pair $(c_i, c_j)$ of inner-node children, determines if merging those would be beneficial, and, determines the pair with maximum (positive) cost reduction ($SA(c_i) + SA(c_j) - SA(c_i \cup c_j)$). Since we perform the merge only if this gain is positive, it is still possible to get stuck in local minima, but this seems to be quite tolerable.

**Taking it all together.** Initially, we had assumed it would be beneficial to perform collapsing bottom-up (i.e., recurse first, then apply the just described collapsing operations). Once again, however, intuition turned out to be wrong, and produces relatively bad trees. Instead, the algorithm that produces the best trees is to start at the root, to first maximize the respective node's branching factor by pulling up children as long as possible; then to recurse into the children; and only at the very end trying to merge pairs of leaf-node or inner-node children. This corresponds to collapsing inner nodes top-down and leaf nodes bottom up (which is algorithmically similar to standard top-down greedy splitting).

Though our method attempts to produce trees with an optimal cost, a purely SAH based collapsing tends to produce trees with relatively low branching factors (5–6 on average). While the cost metric indicates that trees with higher branching factors would *not* be faster even though they produce higher SIMD utilization, forcing higher branching factors has the added benefit of reducing memory consumption. The low branching factors turned out to be from nodes near the leaves, so we typically perform an *a-priori* leaf collapsing before SAH collapse that collapses every sub-tree with at

| | erw6 (804) | office (34K) | robots (72K) | soda (141K) | conference (282K) | cruiser (3.6M) |
|---|---|---|---|---|---|---|
| original binary BVH w/ SAH build | | | | | | |
| num leaf nodes | 335 | 16K | 34K | 63K | 132K | 1.75M |
| num inner nodes | 335 | 16K | 34K | 63K | 132K | 1.75M |
| SAH box cost estimate | 445 | 482 | 13.6M | 1.5K | 3806 | 13.9K |
| SAH tri cost estimate | 190 | 141 | 2.0M | 368 | 546 | 1.6K |
| avg num active children / node | 2 | 2 | 2 | 2 | 2 | 2 |
| avg num tris / leaf | 2.4 | 2.13 | 2.26 | 2.26 | 2.1 | 2.08 |
| after SAH-based collapse | | | | | | |
| num leaf nodes | 74 (-4.5x) | 2833 (-5.6x) | 6345 (-5.4x) | 11.3K (-5.6x) | 25K (-3.2x) | 318K (-5.5x) |
| num inner nodes | 7 (-48x) | 296 (-54x) | 759 (-44.8x) | 1147 (-55x) | 2.8K (-47x) | 32.5K (-54x) |
| SAH box cost estimate box/tri | 7.3 (-61x) | 21.8 (-22x) | 1.37M (-9.9x) | 125 (-12x) | 449 (-8.5x) | 2.06K (-6.7x) |
| SAH tri cost estimate box/tri | 117 (-38%) | 110 (-22%) | 1.86M (-7%) | 278 (-24%) | 471 (-15%) | 1.49K (-9%) |
| avg num active children / node | 11.4 | 10.6 | 9.4 | 10.9 | 9.7 | 10.8 |
| avg num tris / leaf | 10.9 | 12.0 | 11.3 | 12.5 | 11.3 | 11.5 |

Table 1: BVH statistics for original binary BVH before collapsing, as well as for the BVH produced through SAH collapsing. The collapsed BVH consistently has a branching factor of 9.4–11.4 (nodes that do not contain any leaves are always filled), and have significantly lower expected SAH cost. The number of nodes drops by 45–55x for inner nodes and 4–5x for leaf nodes, indicating a 3–4x reduction in overall memory footprint.

most 16 triangles into a leaf node. The resulting trees can have a higher expected cost but achieve average branching factors of 10–12 (rather than 5–6), and consequently require less memory.

Cost-wise, collapsing is rather cheap: though not optimized at all, on a 2.2GHz Core2 laptop CPU collapsing an existing BVH requires roughly 12 seconds for the cruiser (3.6M triangles), and less than half a second for the conference scene (280K tris).

### 4.3 Top-down recursive splitting

While collapsing produces fairly good trees with reasonable build times, it still consumes quite a bit of memory and would never be faster than the initial binary BVH build. Therefore, we also investigate a top-down greedy splitting technique, which is both surprisingly simple and produces nearly the same tree quality.

The basic algorithm is to always keep a set of "potential" nodes, each of which corresponds to a sub-tree that is not split, yet. For each such potential node, we can–using standard binning techniques in the spirit of [8, 16]–determine if there is any good split for this node. Starting with a single node that contains all triangles, we can then greedily split the node with the biggest surface area until either no more split-able nodes are available, or until 16 nodes have been generated. At this stage, the (up to 16) nodes are stored in a multi-node; those children without splits are marked as leaves, and those with valid splits are processed recursively, eventually producing new multi-nodes. The final algorithm (see Algorithm 1) has slightly different branching factors than SAH-based collapsing, but roughly the same SAH cost. Though producing comparable trees and faster to build, for the remainder of this paper we will use the SAH-collapsing method described above.

### 4.4 Resulting Build Quality

In Table 1, we compare several performance influencing factors (expected SAH cost, average branching factor, etc) for both the binary BVH before collapse, as well as for the resulting BVH after the collapsing has been performed. As can be seen, the collapsing produces branching factors of 9.4–11.4 (nodes with high SAH close to the root will typically be full) which we believe is reasonably close to the optimum. The multi-BVH has consistently 4–5$x$ less leaf nodes and consistently produces around 11 triangles per node when the a-priori collapsing of sub-trees with 16 triangles is used. Note that the branching factor and the number of triangles per leaf will be a first-order approximation to SIMD utilization during traversal. This indicates around 70% utilization, which is actually quite good for incoherent rays (compare, e.g., to the data in [25]).

The number of inner nodes drops even further, by consistently around 50$x$. Though every node now is 16$x$ as large as a traditional node, the net savings is still significant (we had, in fact, expected a

**Algorithm 1** Pseudo-code for top-down splitting. "wants to get split" involves an SAH binning step

```
NodeState rootState = {all triangles }
make-multi-node(rootState);

Function makemultinode(NodeState: node to split)
multi-node = { node to split }
while |multi-node| < 16 do
    pick non-leaf node with largest SAH that wants to get split
    if no node left that wants to get split then
        break;
    else
        split and remove node, insert children into multi-node
    end if
end while
for all nodes of multi-node do
    if node is non-leaf then
        child = make-multi-node(child); {recurse}
    end if
end for
```

*higher* memory consumption). This significantly lower node count and memory consumption indicates potential for fast build algorithms, which we have not investigated, yet.

The expected SIMD triangle tests cost goes down (9–38%), but not significantly; since a binary BVH is very good at reducing triangle tests, this is hardly surprising. The expected cost for SIMD box tests however drops significantly, from 6.7$x$ for the cruiser to up to 61$x$ for erw6; for conference and nrcoff the savings in expected SIMD box tests is around one order of magnitude.

## 5 SIMD Traversal and Intersection

Our Multi-BVH is implemented in an existing ray tracer assuming a 16-wide SIMD target. This codebase already supports packet traversal (for SIMD-sized packets), a prototypical SIMD stream tracing in the spirit of [25], and a version of the aggressive DynBVH traversal algorithm as described in [24]. Since both DynBVH and SIMD stream tracing rely on packets that are larger than 16 rays, the codebase is designed around large packets that have a fixed maximum size, but which can be partially and sparsely filled (rays can be flagged as inactive). On average, 64–256 rays per packet are being used, though some shaders may trace shadow rays in batches of 16 per packet (part of which can be inactive). We use the existing shading framework, but traverse each of the packet's rays individually, one after another.

## 5.1 Traversal Setup

Given an input ray packet, we simply loop over all (active) rays in that packet and traverse them individually. Each such ray gets pulled out of the packet, and is immediately replicated into a 16-ray SoA packet format similar to that used by packet techniques (this allows the compiler to keep the ray data in registers). This ray is then fed to the traversal loop, which starts at the root (multi-)node.

## 5.2 SIMD Node Test

Whenever a ray visits a multi-node, it is tested against that node's 16 AABBs. This is done using the same SIMD slabs test as used in the traditional packet techniques: the test performs 16 ray/box tests in parallel, and doesn't care whether any of the rays and/or boxes are the same. The test returns a mask indicating which of the 16 children are hit by the ray.

## 5.3 Strict Front-to-Back Traversal Order

Based on this valid mask, we now have up to 16 children that may need to be traversed. Binary BVHs are traditionally traversed depth-first with an "ordered traversal" that determines the traversal order of both two children based on the given ray's direction sign for the node's chosen axis.

For our multi-BVH, this method cannot easily be applied, as picking an *a-priori* ordering for 16 different boxes is not trivial. Alternatively, we could traverse the nodes in the order of their distance to the ray origin [11]. This distance is already calculated during the box test, but determining the order of the intersected boxes would require a horizontal sort of two SIMD registers, which is not trivial.

Instead of using a stack-based depth-first traversal, we can use a strict front-to-back traversal based on keeping a *unsorted list* of which nodes are currently "active", as well as the distance to each node. After every (multi-)node test, the node's active children are appended to the list using a fast vector compact operation that compacts the node's child IDs based on the box test's overlap mask.

Once a node has been traversed, the next node to be traversed is selected by scanning the active list for the element with the smallest distance. Since only a few children for a given node are active and since the multi-BVH is fairly shallow the list stays very short (it is rarely longer than 16 elements). Thus, scanning the list for the shortest element is very fast when scanned in 16-wide blocks at a time. After the closest node has been selected, we replace it with the element at the end of the list (again, we do not sort the list).

## 5.4 Leaf Intersection and Active List Pruning

So far, we have only considered inner node traversal steps. Once traversal reaches a leaf node, we obviously intersect the triangles in that leaf (see below), which is relatively straightforward. In addition, since every triangle intersection step has the potential to shorten the valid ray distance (if a hit is found, we do not have to traverse beyond that ray), there is also a good chance that a (successful) ray-triangle intersection would result in some of the nodes in the active list to now be behind the newly found hit distance.

To exploit that fact, we prune such newly deactivated nodes from the active list by performing a stream compaction on the list. Since the list can only get shorter, there is no need for temporary memory. Again, this operation is very fast, as vector compaction is cheap, and the whole list is typically processed in a single iteration (since it is usually short).

## 5.5 Traversal order – Taking it all together

Taken together, using a strict front-to-back traversal order based on an (unsorted) active list is quite simple, and very efficient. In particular, though we had feared the list might become prohibitively long, this is not the case: it is indeed possible to construct a degenerate case where the list contains all leaf nodes (i.e., the list length is only bounded by the number of nodes), but this is a worst-case assumption and in practice does not happen[1]. On average the list length is 3–5 elements, and rarely ever gets longer than 16. The likely reason for this is that even if a node has 16 children (often less), only few (≈3) of those will actually be active, and only those will be appended to the list. At the same time, every traversal step takes one node off that list, and every leaf traversal will perform an additional pruning. With the trees being as shallow as they are (see Table 1), the list never grows very large. The operations to manipulate this list (compact during prune, and selecting the closest among 16) are very cheap, too, at least on our target architecture. On our target architecture, we also believe this is less expensive than a full priority queue approach [11] and performs the same function. The complete algorithm is depicted in Algorithm 2.

**Algorithm 2** Pseudo-code for the front-to-back SIMD traversal.

```
nodeID = root; activeList = empty;
while true do
  node = rootNode; activeList = empty;
  if node is leaf then
    Perform SIMD triangle tests for 16 tris at a time
    compact activelist w/ new hit distance
  else
    (overlapmask,distances) = SIMD box test(node.boxes);
    vector compact (node.childIDs,distances) w/ overlapmask
    append compacted (node.childIDs,distances) to activelist
  end if
  if activelist is empty then
    return;
  else
    scan activelist for node w/ minimum distance
    replace closest node with last node; shrink list
  end if
end while
```

## 5.6 SIMD Triangle Intersection

Though we have so far neglected this issue, when we reach a leaf we also have to intersect the triangles stored in this leaf. We iterate over the leaf's item list in SIMD chunks of 16, fetch 16 triangle IDs, gather the resulting triangles and vertices, and perform 16 parallel triangle tests. The resulting code again is nearly identical to the 16-rays-one-triangle code, except that after all triangle tests have been performed, an additional reduction has to be performed (since the ray may have hit several of the triangles and we have to determine the closest one). This reduction however is rather simple.

## 5.7 Pre-Gathering

One drawback of the 16-wide SIMD triangle test is that the intersection code has to gather triangles from up to 16 different memory locations. This can be avoided by pre-gathering each leaf's triangles during building, and then having each leaf store its triangle's vertices in compact SoA form. This is trivial to implement and greatly simplifies memory access patterns, but it also increases overall memory consumption, so whether it pays off probably depends on the application. With leaves now having 10 and more triangles (see Table 1), it may also make sense to combine our technique with other techniques like storing self-sufficient meshlets or triangle strips inside each leaf; but this, too, requires further investigation.

## 6 Results

As mentioned before, our approach is designed for the Larrabee architecture; since this is not yet physically available, we will focus

[1]For a similarly artificial case of a completely uniform geometry, the list length is bounded by $O(klogN)$), where $logN$ is the depth of the tree.

on statistical traversal results (albeit only for 16-wide SIMD), as well as on emulator/simulator data. All code can be compiled and executed on the actual simulators, and simulator runs have been performed on selected experiments to ensure correctness. All images and statistics have been computed by recompiling the same code with an emulation library written in standard C++.

Test Scenes. As test scenes, we have chosen the freely available MGF scenes erw6, office, NRC office, conference, soda shoppe, and cruiser, as well as the robots scene from the BART benchmark [12]. These scenes (see Figure 3) span a wide range of complexity, from 800 triangles in erw6 to over 3.4 million triangles in cruiser. For erw6, office, NRC office, and soda shoppe we have used the original area light sources (spheres and quads) from the MGF files; for cruiser, conference, and robots the original light sources have been replaced with a single area light.

Shaders / Ray Distributions. In terms of ray distributions, we have chosen an EyeLight shader ($N \cdot V$) that traces only primary rays; a shader that computes two bounces of perfect specular reflections irrespective of actual surface parameters; a shader that computes soft shadows; and a path tracer with 2 diffuse bounces and separate light source sampling (see Figures 1 and 2). The path tracer setup takes 16 samples per pixel, and combines $4 \times 4$ pixels into the same packet (i.e., 256 rays per packet total). All other shaders use a single sample per pixel, and combine $8 \times 8$ pixels (i.e., 64 rays per packet). Soft shadows are computed with 16 samples per light source; as there are different ways of doing this we have added two separate implementations – one that traces a 16-sample packet for every surface point, and one that traces 16 successive packet (of 64 rays each) in which every surface point takes one random sample per light. The path tracer computes one sample per light per surface point, and groups shadow rays into one packet per light source. Each incoming ray is then bounced by taking a random direction on the hemisphere, and is bounced exactly two times (i.e., a maximum path length of three rays). To generate random samples, we use a simple 48-bit linear congruence generator.

Secondary ray packets can be partially filled (e.g., through shadow rays facing away from the surface normal); unused ray slots get masked out, but partially filled packets never get re-ordered, compressed, or optimized in any other form.

Traversal Methods. In terms of traversal methods, we compare our multi-BVH traversal against both DynBVH-style traversal and SIMD packet traversal ("packet$_{16}$" traversal). All traversal methods get fed with the same packets (with up to 256 rays per packet for the path tracer), but traverse them differently: multi-BVH traversal traverses each packet ray by ray; packet$_{16}$ traces larger packets in batches of 16 rays (without any frustum or interval optimizations), and DynBVH traces the entire packet with first-hit tracking and interval arithmetic culling (see [24]). Both packet$_{16}$ and multi-BVH traversal should be able to keep all ray data in registers, DynBVH will not. Also note that secondary ray packets are not always "full", which particularly hampers the DynBVH traversal.

## 6.1 Statistical Results per Ray Distribution

The first obvious value to quantify is the number of SIMD box tests and SIMD triangle tests performed by each traversal method. Though our single-ray traversal intersects one ray with 16 triangles/nodes rather than 16 rays with the same node/triangle, the operations performed by both variants are almost identical, and are roughly comparable in cost. There may, of course, be a difference in memory access cost, as the 1-ray 16-triangle version will collect 16 different triangles, which is not the case for the packet versions. On the other hand, the multi-BVH traversal will always read 16 BVH nodes "en bloc", while packet traversal and DynBVH traversal read from multiple cache lines in a less predictable pattern. As we believe that these two factors will roughly cancel each other out, we will assume roughly comparable cost for both variants.

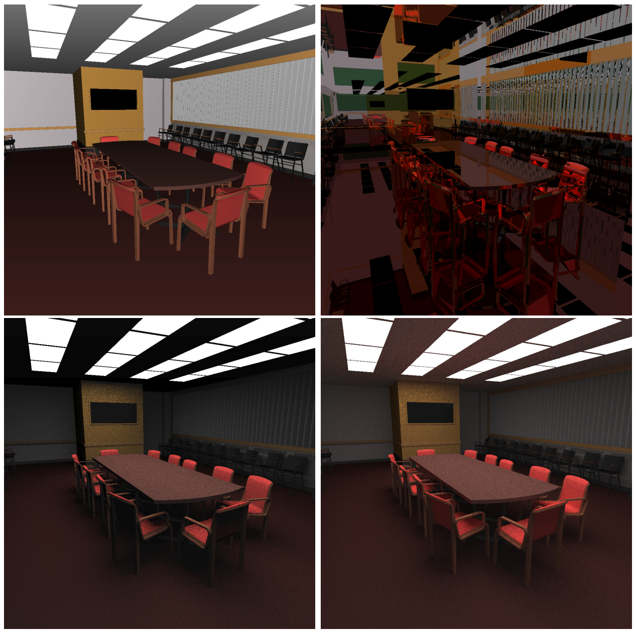
Figure 2: Eyelight shader (primary rays), perfectly specular reflections (2 bounces), soft shadows (16 samples per light), and path tracing (2 bounces, 1 sample per light) for the conference scene.

The actual numbers for all our scenes, ray distributions, and traversal algorithms can be found in Tables 2–5. Not unexpectedly, DynBVH is the most volatile among these techniques, performing significantly less operations for primary rays and (to a lesser degree) for the forced specular reflections, but performing up to 6$x$ more triangle tests even than packet traversal for the path tracing case. Our multi-BVH technique behaves exactly opposite to that, performing up to 3–4x more operations for primary rays, but for incoherent rays performs up to 3x less triangle tests than packet traversal, and up to 19x less than DynBVH traversal. While the large number of triangle tests for DynBVH is not unexpected (due to its speculative nature), it turns out that single ray multi-BVH traversal even performs less box tests (3–4x), which we found somewhat surprising.

Note that these numbers do not yet capture other effects such as the additional interval tests performed by DynBVH traversal (which we completely ignore here), the fact that DynBVH cannot hold all rays in registers, or the fact that too large packets may have a negative impact on first-level cache (256 rays already consume 12 kilobytes!). Also, it does not capture the fact that with the ability to trace arbitrarily sized packets the renderer might use the same number of rays more economically (e.g., taking only five samples on a light rather than 16).

## 6.2 Memory Bandwidth

Though the number of primitive operations is a first-order approximation of compute requirements, it does not model the second important factor that influences performance: bandwidth. Packet tracing reduces memory bandwidth by amortizing each memory access over all rays in the packet. Though our traversal statistics indicate that we perform less node traversals and triangle intersections, each node read from memory is now significantly larger (16x in bytes, 8x in cache lines), and triangle intersection now requires gathering up to 16 different triangles.

Since this might potentially become a performance problem, we quantified the memory bandwidth by taking a functional simulator and running both packet and multi-BVH traversal through

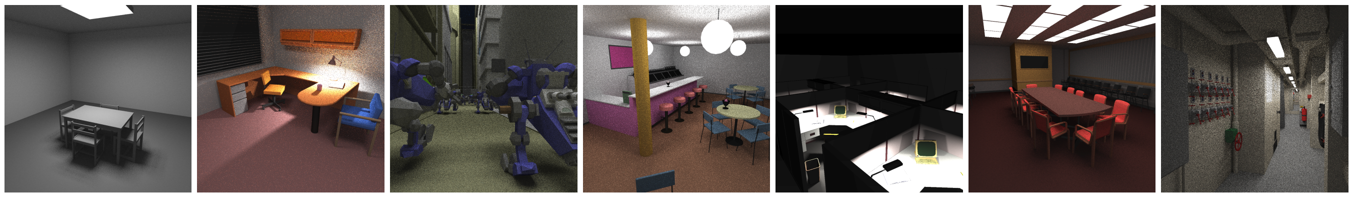
Figure 3: Test scenes used in our experiments. From left to right: erw6 (804 triangles, 1 quad light); office (34K, 2 quad lights, 1 sphere light); robots (72K, 1 quad light); soda shoppe (141K, 6 sphere lights); nrc-office (222K, 12 quad lights); conference (280K, 1 quad light); and cruiser (3.6M triangles, 1 quad light). All images are rendered with a two-bounce diffuse path tracer with 1 sample per light and 64 samples per pixel.

cachegrind (a freely available cache simulator [2]) in order to determine the number of L2 misses.

Based on cachegrind, the L2 bandwidth for path tracing the conference scene is indeed higher for the single-ray multi-BVH traversal than for the packet based binary traversal, but not significantly so: packet traversal reported 48 million L2 misses (29m read; 19m write), multi-BVH traversal reported 54m (34m read; 19m write). As the write misses indicate, there are some constant factors (build, frame buffer writes) that somewhat hide the true impact on traversal bandwidth, but at least compared to overall per-frame bandwidth a 20% impact on bandwidth seems tolerable.

### 6.3 Performance Comparison

As our target architecture–Larrabee [23]–is not yet physically available, all numbers reported so far are based on simulator data. Our whole system is written in C/C++ using SIMD intrinsics, is then compiled using a special version of the Intel compiler, and run on a cycle-accurate simulator. Due to the preliminary state of the tools we used, however, all results have to be taken with a grain of salt: In particular, the newer pieces of the code (i.e., multi-BVH and shaders) are much less mature than the previously existing pieces of code. This becomes aggravated by the fact that the system has almost exclusively been developed in a functional simulator, and has not yet been duly optimized for the actual hardware/cycle-accurate simulator.

Since the Larrabee architecture is not yet publicly available, any actual performance data on it is of a sensitive nature. We therefore report only relative performance data, with all performance normalized to the multi-BVH traversal performance. Since a simulator run for full-screen path tracing with multiple rays per pixel can take a long time in cycle-accurate mode, we have restricted ourselves to only the conference scene (we believe this to be the most representative one), to only a single core (performance usually scales linearly in number of cores), and to the Eyelight and 2-bounce path trace shaders. Even then, for path tracing we have not rendered the full image, but only a fixed number of randomly selected screen tiles (of course, all traversals render the same tiles).

| | mBVH | $\text{packet}_{16}$ | dynbvh |
|---|---|---|---|
| primary rays | 1.0 | 3.0× | 3.6× |
| two-bounce path tracer | 1.0 | -6% | -46% |

Table 6: Relative performance for packet tracing, dynbvh traversal, and our multi-BVH for both primary rays and two-bounce path tracing in the conference scene. Performance is relative to the multi-BVH, and is measured on a cycle-accurate simulator (single core).

As can be seen from Table 6, the simulated data does indeed correlate to the statistical performance data: As expected, for primary rays both packet and dynbvh traversal outperform the multi-BVH by $3-4\times$. For path tracing, the situation is different, with the multi-BVH outperforming DynBVH traversal by almost $2\times$. Compared to the plain packet tracer, the performance advantage is slimmer, however, with our multi-BVH outperforming the chunk traversal by only a few percent. This relatively low performance is due to several different effects: First, the multi-BVH is the newest and least optimized of the three traversals, leaving some room for improvement. Second, the simulator reports a significantly lower core utilization than for both packet and dynbvh traversal, indicating that the additional memory accesses (likely for the pre-processed triangle data) are hurting (only parts of the image have been rendered, and the simulator starts with cold caches). Third, as already indicated before the multi-BVH traversal and intersection have to perform some horizontal operations that a packet traverser does not, and those are costly. Fourth, and most importantly, the impact of the shading cost masks the true impact on traversal efficiency: the path trace shader is quite complex (including random number generation, sampling, trigonometric functions, etc) and is not optimized at all, eventually making the shader cost more than half the total per-frame time and dwarfing the time spent in ray-triangle and ray-box tests. Finally, as argued before we believe that freeing shaders from the burden to produce coherent packets of rays would allow to simplify and optimize the shaders, which is not factored into these results (the high cost for the shaders in fact supports this view).

### 6.4 Comparison to QBVHs

Concurrently to our work, the use of shallow BVHs has also been investigated by Dammertz et al [4]. In fact, both approaches are surprisingly similar in the data structure and algorithms they propose as well as in the conclusions they draw. One of the few differences relates to the traversal order: Dammertz stores the traversal order per node, and then uses this in a depth-first traversal; which is essentially the same as the "ordered traversal" used for binary BVHs (see, e.g., [24]). We instead use a strict front-to-back traversal.

The much bigger difference between Dammertz' approach and ours is the branching factor: while Dammertz argued for a branching factor of 4, we use 16. This however is mainly a side effect of the architecture that the algorithms run on, and 4 is likely to be the better choice for 4-wide SIMD architecture. Whether 16 is the right branching factor even on a 16-wide SIMD architecture will require further investigation: While there are some indications that branching factors higher than two can be beneficial by themselves, a branching factor of 16 would arguably be too wide for a SIMD width of less than 16. In fact, the cost metric we have used suggests that the optimal branching factor is somewhere between 4 and 8.

Consequently, if there were a method that would use the 16-wide SIMD to intersect less than 16 triangles (say, four), and that would do that *faster* than we currently perform 16 triangle intersections, then a BVH with a less extreme branching factor might be even faster. Some obvious approaches would be to use 16-wide SIMD to test the three dimensions of a 4-wide BVH, or to compute the three edge tests for four triangles. Such triangles are already being investigated, but so far are not fast enough; in their absence, we believe 16 to be the right choice for our architecture.

## 7 Summary and Conclusion

We have investigated the use of BVHs with branching factors of 16 to efficiently trace rays on a 16-wide SIMD architecture without having to rely on packets. Compared to the speculative BVH packet traversal, our technique performs significantly worse for primary rays and coherent ray distributions, but performs better for less coherent distributions like soft shadows and path tracing. Compared to SIMD packet tracing, the difference is smaller but the new technique still at least competitive for incoherent rays. Because our approach does not rely on ray coherence, the single-ray multi-BVH traversal even performs less SIMD box/triangle tests than the breadth-first SIMD stream tracing approach proposed in [25].

In addition, our method offers several interesting points for further improvements that we have not yet investigated. With significantly shallower trees, less nodes, and more triangles per leaf, the multi-bvh could be built very efficiently, which is promising for dynamic scenes. For coherent rays, it might also be possible to combine our approach with existing BVH packet traversal.

Outside of traversal, a renderer that is no longer required to produce ray packets with specific properties allows to re-establish a degree of flexibility that was not available for packet-based renderers. Eventually, this should allow to apply our technique in any ray-based renderer, and with any global illumination algorithm technique–including techniques such as bidirectional path tracing, metropolis light transport, or photon mapping. In particular, supporting arbitrarily sized packets should also make it easier to use adaptive sampling and filtering techniques. For shading, working on SIMD granularity rather than on larger packets is a big advantage: it allows to keep more data in registers, it simplifies the shading loop, and it allows rays to be traced as soon as they are generated rather than having to queue them up into large enough packets.

On the downside, it is at least possible that practical ray distributions will be significantly more coherent than path tracing or random light source sampling. Similarly, it could turn out that packet techniques can be made simpler and more stable than we have assumed in this paper. In either of those two cases, packet techniques would work much better than we have assumed in this paper, and would remain the way to go, in which case our technique would make sense only for rather extreme niche applications.

Though the existing numbers are promising, a lot still remains to be done. In particular, it is not clear whether 16 is a good choice for either branching factor or leaf size. Also, packets would still make sense for coherent ray distributions, but how to do this exactly remains to be investigated. It also remains to be seen how exactly the different techniques will perform when fully optimized for the actual hardware, with all effects (number of registers, caches, bandwidth) taken into account. The potential to trace individual rays nearly as fast as packets also suggests that it may be appropriate to re-think the overall system design of a real-time ray tracer. Finally, if it is indeed true that even our relatively simple path tracer is already most costly than tracing the rays (irrespective of traversal method), then it may also be time to stop trying to further optimize ray traversal, and to rather concentrate on shading.

## References

[1] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007*, May 2007.

[2] Valgrind Tool Suite. http://valgrind.org.

[3] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali. Ray tracing for the movie 'Cars'. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[4] H. Dammertz, J. Hanika, and A. Keller. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Rendering Techniques 2008 (Proc. 19th Eurographics Symposium on Rendering)*, 2008.

[5] K. Dmitriev, V. Havran, and H.-P. Seidel. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[6] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. pages 14–20, 1987.

[7] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[8] W. Hunt, G. Stoll, and W. Mark. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006.

[9] Intel Corp. Intel Pentium III Streaming SIMD Extensions. http://-developer.intel.com/vtune/cbts/simd.htm, 2002.

[10] Intel Corp. Intel AVX. http://softwareprojects.intel.com/avx, 2008.

[11] T. Kay and J. Kajiya. Ray tracing complex scenes. *Proceeding of SIGGRAPH*, pages 269–278, 1986.

[12] J. Lext, U. Assarsson, and T. Möller. BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, 2000.

[13] E. Mansson, J. Munkberg, and T. Akenine-Moller. Deep Coherent Ray Tracing. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 79–85, 2007.

[14] P. A. Navrátil, D. S. Fussell, C. Lin, and W. R. Mark. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. 2007.

[15] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH*, pages 101–108, 1997.

[16] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006.

[17] A. Reshetov. Personal communication.

[18] A. Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 57–60, 2006.

[19] A. Reshetov. Faster Ray Packets-Triangle Intersection through Vertex Culling. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 105–112, 2007.

[20] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. In *Proceedings of SIGGRAPH*, pages 1176–1185, 2005.

[21] S. Rubin and T. Whitted. A 3D representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH*, pages 110–116, 1980.

[22] J. Schmittler. *SaarCOR - A Hardware-Architecture for Realtime Ray Tracing*. PhD thesis, Saarland University, 2006.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3), 2008.

[24] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):6:1–6:18, Jan. 2007.

[25] I. Wald, C. P. Gribble, S. Boulos, and A. Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007.

[26] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. In *Proceedings of SIGGRAPH*, pages 485–493, 2006.

[27] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).

[28] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. In *Proceedings of EUROGRAPHICS*, pages 153–164, 2001.

| scene | #tris | #lights | num SIMD box-tests | | | | num SIMD tri tests | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream |
| erw6 | 806 | 1 | 1.22M | 1.01M (83%) | 282K (23%) | 1.00M (82%) | 1.17M | 390K (33%) | 501K (43%) | 384K (33%) |
| nrcoff | 222K | 12 | 5.73M | 4.95M (86%) | 1.45M (25%) | 4.88M (85%) | 2.00M | 650K (32%) | 908K (45%) | 608K (30%) |
| conf | 280K | 1 | 3.77M | 3.39M (90%) | 1.05M (28%) | 3.25M (86%) | 1.86M | 518K (28%) | 894K (48%) | 421K (23%) |
| office | 34K | 3 | 2.07M | 1.75M (85%) | 522K (25%) | 1.70M (82%) | 1.62M | 832K (51%) | 1.13M (70%) | 794K (49%) |
| soda | 141K | 6 | 4.12M | 2.48M (60%) | 735K (18%) | 2.42M (59%) | 2.08M | 966K (46%) | 1.40M (67%) | 886K (43%) |
| robots | 72K | 1 | 7.81M | 5.08M (65%) | 1.59M (20%) | 4.97M (64%) | 1.84M | 443K (24%) | 648K (35%) | 402K (22%) |
| cruiser | 3.6M | 1 | 7.08M | 5.37M (76%) | 1.70M (24%) | 5.26M (74%) | 2.26M | 703K (31%) | 1.06M (47%) | 652K (29%) |

Table 2: Traversal results for primary rays. As expected, our single-ray traversal cannot compete with packet techniques for almost perfect coherent rays. Though surprisingly close to naïve packet tracing, we perform 2–3 times as many triangle/box tests as DynBVH.

| scene | #tris | #lights | num SIMD box-tests | | | | num SIMD tri tests | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream |
| erw6 | 806 | 1 | 3.78M | 3.20M (85%) | 1.14M (30%) | 3.16M (84%) | 3.60M | 1.18M (33%) | 1.53M (42%) | 1.14M (32%) |
| nrcoff | 222K | 12 | 19.0M | 17.1M (90%) | 6.67M (35%) | 16.7M (88%) | 6.99M | 2.74M (39%) | 4.12M (59%) | 2.42M (35%) |
| conf | 280K | 1 | 14.8M | 14.6M (99%) | 7.65M (52%) | 13.4M (91%) | 7.09M | 3.43M (48%) | 6.35M (90%) | 2.47M (35%) |
| office | 34K | 3 | 7.12M | 6.49M (91%) | 2.68M (38%) | 6.39M (90%) | 5.76M | 3.72M (65%) | 5.09M (88%) | 3.45M (60%) |
| soda | 141K | 6 | 11.7M | 9.05M (77%) | 4.08M (35%) | 8.85M (76%) | 7.72M | 4.81M (62%) | 7.63M (99%) | 4.22M (55%) |
| robots | 72K | 1 | 23.6M | 18.3M (78%) | 8.65M (37%) | 17.3M (73%) | 6.64M | 2.25M (34%) | 3.52M (53%) | 1.73M (26%) |
| cruiser | 3.6M | 1 | 23.5M | 20.4M (87%) | 10.1M (43%) | 19.5M (83%) | 7.57M | 3.37M (45%) | 5.72M (76%) | 2.69M (36%) |

Table 3: Traversal results for (forced) specular reflections (two bounces). Though somewhat more costly than primary rays, specular reflections still perform well with both packet traversal-16 and DynBVH traversal (numbers are aggregate, and include primary and first-generation bounce rays).

| scene | #tris | #lights | num SIMD box-tests | | | | num SIMD tri tests | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream |
| **Packet-grouping: 16 successive packets (per light) of 64 rays each (1 light sample per hitpoint)** | | | | | | | | | | |
| erw6 | 806 | 1 | 17.4M | 17.1M (98%) | 7.76M (45%) | 16.4M (94%) | 16.6M | 5.90M (36%) | 7.34M (44%) | 5.40M (33%) |
| nrcoff | 222K | 12 | 368M | 341M (93%) | 129M (35%) | 318M (86%) | 156M | 64.4M (41%) | 98.2M (63%) | 53.4M (34%) |
| conf | 280K | 1 | 58.1M | 103M (1.8x) | 71.7M (1.2x) | 79.4M (1.4x) | 26.1M | 36.5M (1.4x) | 61.5M (2.4x) | 21.8M (84%) |
| office | 34K | 3 | 111M | 134M (1.2x) | 64.6M (58%) | 95.6M (86%) | 45.1M | 59.4M (1.3x) | 92.1M (2.0x) | 38.8M (86%) |
| soda | 141K | 6 | 228M | 472M (2.1x) | 307M (1.3x) | 261M (1.1x) | 114M | 191M (1.7x) | 398M (3.5x) | 123M (1.1x) |
| robots | 72K | 1 | 103M | 228M (2.2x) | 188M (1.8x) | 166M (1.6x) | 23.2M | 46.0M (2.0x) | 84.7M (3.7x) | 30.1M (1.3x) |
| cruiser | 3.6M | 1 | 83.2M | 128M (1.5x) | 112M (1.3x) | 76.4M (92%) | 18.8M | 20.3M (1.1x) | 49.2M (2.6x) | 8.77M (47%) |
| **Surface sample based grouping: 64 Packets of 16 light samples per surface each** | | | | | | | | | | |
| erw6 | 806 | 1 | 17.1M | 16.6M (97%) | 15.9M (93%) | 24.2M (1.4x) | 16.6M | 5.74M (35%) | 8.49M (51%) | 7.73M (47%) |
| nrcoff | 222K | 12 | 361M | 307M (85%) | 304M (84%) | 325M (90%) | 156M | 52.4M (34%) | 101M (65%) | 55.5M (36%) |
| conf | 280K | 1 | 57.9M | 98.7M (1.7x) | 96.3M (1.7x) | 138M (2.4x) | 26.1M | 34.2M (1.3x) | 37.0M (1.4x) | 34.9M (1.3x) |
| office | 34K | 3 | 110M | 129M (1.2x) | 128M (1.2x) | 138M (1.3x) | 45.1M | 56.1M (1.2x) | 109M (2.4x) | 57.5M (1.3x) |
| soda | 141K | 6 | 225M | 461M (2.0x) | 459M (2.0x) | 491M (2.2x) | 114M | 180M (1.6x) | 337M (3.0x) | 188M (1.6x) |
| robots | 72K | 1 | 102M | 223M (2.2x) | 219M (2.1x) | 336M (3.3x) | 23.2M | 44.7M (1.9x) | 46.1M (2.0x) | 45.4M (2.0x) |
| cruiser | 3.6M | 1 | 81.8M | 122M (1.5x) | 118M (1.4x) | 104M (1.3x) | 18.8M | 19.3M (1.0x) | 27.1M (1.4x) | 11.8M (63%) |

Table 4: random samples per area light. For both tested grouping strategies, our single-ray based method performs better than both $\mathrm{packet}_{16}$ and DynBVH traversal except for the two smallest scenes. Note that the multi-BVH traversal shows variations due to the random nature of the samples, but is otherwise agnostic to the order or grouped of the rays. SIMD stream tracing might actually perform better if all rays were grouped into one single packet, but would require to re-write the shaders, and to generate prohibitively large packets.

| scene | #tris | #lights | num SIMD box-tests | | | | num SIMD tri tests | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream | $\mathrm{mBVH}_{16}$ | $\mathrm{packet}_{16}$ | dynbvh | stream |
| erw6 | 806 | 1 | 112M | 191M (1.7x) | 198M (1.8x) | 136M (1.2x) | 96.5M | 63.4M (66%) | 165M (1.7x) | 44.0M (46%) |
| nrcoff | 222K | 12 | 1.93B | 6.85B (3.5x) | | 2.81B (1.5x) | 673M | 1.43B (2.1x) | | 725M (1.1x) |
| conf | 280K | 1 | 374M | 1.21B (3.2x) | 1.88B (5.0x) | 736M (2.0x) | 169M | 354M (2.1x) | 2.09B (12.4x) | 225M (1.3x) |
| office | 34K | 3 | 387M | 963M (2.5x) | 1.01B (2.6x) | 490M (1.3x) | 194M | 618M (3.2x) | 1.97B (10.2x) | 278M (1.4x) |
| soda | 141K | 6 | 879M | 2.64B (3.0x) | 3.13B (3.6x) | 1.38B (1.6x) | 446M | 1.42B (3.2x) | 8.65B (19.4x) | 980M (2.2x) |
| robots | 72K | 1 | 616M | 1.78B (2.9x) | 2.19B (3.6x) | 951M (1.5x) | 155M | 370M (2.4x) | 1.65B (10.6x) | 181M (1.2x) |
| cruiser | 3.6M | 1 | 634M | 1.65B (2.6x) | 3.02B (4.8x) | 809M (1.3x) | 183M | 340M (1.9x) | 2.05B (11.2x) | 139M (76%) |

Table 5: Traversal results for a two-bounce diffuse path tracer. For diffusely bounced rays, single-ray multi-BVH traversal is superior to both $\mathrm{packet}_{16}$ and DynBVH traversal except in the trivially simple erw6 scene. Due to its speculative nature, DynBVH traversal actually behaves far worse than even $\mathrm{packet}_{16}$ traversal, and consistently performs 10–20x more triangle intersections than the single-ray multi-BVH traversal. (nrcoff results for DynBVH are missing because due to its many light sources the results did not complete in time). In fact, multi-BVH traversal actually performs less SIMD triangle and box tests than the SIMD stream tracing approach proposed in [25].