Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel® MIC Architecture

Carsten Benthin Ingo Wald Sven Woop Manfred Ernst William R. Mark
Intel ® Corporation

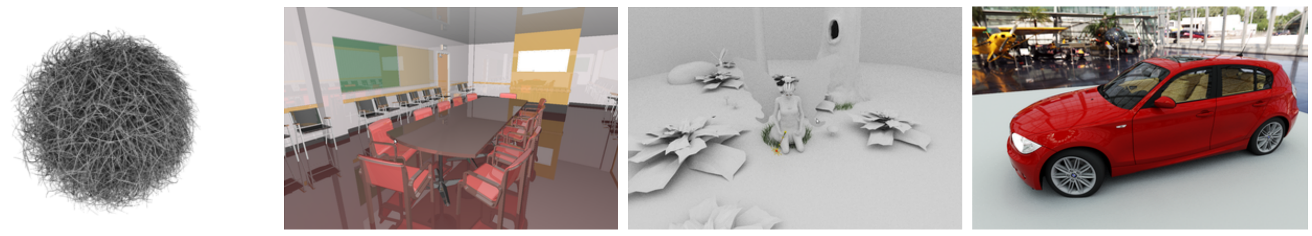


Fig. 1. Several examples rendered with our hybrid ray tracing scheme: *hairball* (2.8M triangles) with ambient occlusion, *conference* (218k triangles) with Whitted-style reflections, *fairy* (174k triangles) with 8-bounce path tracing, and a 4.3M triangle *car* model with path tracing including different BRDFs like Glass, Car Paint, Chrome, Plastic, and Lambertian, lit from an HDRI environment light source.

***Abstract*—Wide-SIMD hardware is power and area efficient, but it is challenging to efficiently map ray tracing algorithms to such hardware especially when the rays are incoherent. The two most commonly used schemes are either packet tracing, or relying on a separate traversal stack for each SIMD lane. Both work great for coherent rays, but suffer when rays are incoherent: The former experiences a dramatic loss of SIMD utilization once rays diverge; the latter requires a large local storage, and generates multiple incoherent streams of memory accesses that present challenges for the memory system.**

In this paper, we introduce a single-ray tracing scheme for incoherent rays that uses just one traversal stack on 16-wide SIMD hardware. It uses a bounding-volume hierarchy with a branching factor of four as the acceleration structure, exploits four-wide SIMD in each box and primitive intersection test, and uses 16-wide SIMD by always performing four such node or primitive tests in parallel. We then extend this scheme to a hybrid tracing scheme that automatically adapts to varying ray coherence by starting out with a 16-wide packet scheme and switching to the new single-ray scheme as soon as rays diverge. We show that on the Intel® Many Integrated Core architecture this hybrid scheme consistently, and over a wide range of scenes and ray distributions, outperforms both packet and single-ray tracing.

***Index Terms*—Ray tracing, SIMD processors**

I. Introduction

Ray tracing is a computationally intensive workload, so it is important to implement it efficiently. On modern programmable architectures such as CPUs and GPUs the key to reaching this goal is to efficiently use those architectures' SIMD units. SIMD units provide high performance and power efficiency by amortizing the hardware for caches and instruction decode over many arithmetic units. Many modern hardware architectures have wide SIMD units: 8-wide AVX on Intel® CPUs [1], 16-wide SIMD on the Intel® MIC architecture [2], and 16-wide (or greater) SIMD on GPUs [3].

Using wide-SIMD hardware *efficiently* can be challenging, since the algorithm must be organized such that all SIMD lanes perform the same arithmetic operation together most of the time. When tracing coherent rays, it is relatively straightforward to achieve high SIMD utilization. One common technique, known as packet tracing, shares one traversal stack and performs the node/triangle intersection test for all N rays [4]. All rays are forced to follow the same traversal sequence by always descending a subtree if any of the rays wants to traverse the subtree, using masks to track which rays are *active*. Packet tracing is particularly efficient on *explicit* SIMD architectures (where the SIMD length is exposed in the instruction set) because it does not require scatter/gather operations, and because the mix of scalar and vector operation utilizes both scalar and vector units. However, performance degrades badly once ray divergence becomes significant, eventually reaching a state where only very few of the *N* SIMD lanes are still active.

An alternative is to use the SIMD unit to trace *N independent rays*. That is, each SIMD lane has its own ray and its own traversal stack. This technique is particularly popular on today's GPUs, which have an implicit SIMD architecture that is well matched to it [3]. In a *single program multiple data* (SPMD) programming model such as CUDA or OpenCL, this even gives the appearance of each SIMD lane running its own scalar program. However, SIMD efficiency loss still occurs if different SIMD lanes execute different code paths. For example, if some rays want to descend further into the acceleration structure but others want to perform a ray/triangle intersection test, some of the SIMD lanes will be idle [3].

There are two more subtle performance challenges associated with the *N-independent-rays* algorithm. First, it needs a large working set for *N* independent stacks and temporary variables. This requires lots of (vector) registers and lots of local storage, which in turn lead to costly spilling to device memory, and/or to a reduction in the number of threads available for latency hiding. Second, every memory access translates to a

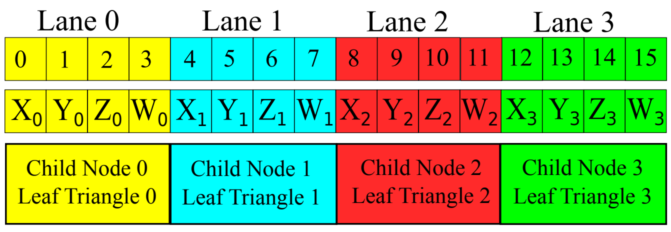

Lane 0
Lane 1
Lane 2
Lane 3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
X_0 Y_0 Z_0 W_0 X_1 Y_1 Z_1 W_1 X_2 Y_2 Z_2 W_2 X_3 Y_3 Z_3 W_3
Child Node 0 Leaf Triangle 0
Child Node 1 Leaf Triangle 1
Child Node 2 Leaf Triangle 2
Child Node 3 Leaf Triangle 3

Fig. 3. We view a 16-wide SIMD register as *4 lanes of 4 elements*, using four-wide SIMD (x,y,z,w) for four node respectively primitive intersection tests in parallel; shared data like ray and traversal stack have to be stored only once.

scatter/gather operation. When rays are coherent, the addresses generated by all lanes will be adjacent, but when not, they will be very different. Handling *N* totally divergent memory accesses in each iteration however is a challenge for any memory system, e.g. the number of concurrently open DRAM pages (and times to open/close such pages), number of TLB entries, etc. These factors explain why the performance of the *N-independent-rays* algorithm degrades significantly on modern GPUs as rays diverge [3].

These considerations lead to the question whether it is possible to use (wide-)SIMD hardware to efficiently trace a single ray at a time. One approach is to use a bounding volume hierarchy (BVH) with a branching factor and leaf size equal to the SIMD width (an *MBVH acceleration structure*) [5], [6], [7]. This approach uses N-wide SIMD to perform N node or triangle intersection tests in parallel for a single ray and does not rely on ray coherence at all. However, this approach quickly loses algorithmic efficiency for branching factors greater than four and with branching factors of 16 or greater is significantly worse than packet tracing if there is even a small amount of ray coherence [7].

A promising approach would be to use a branching factor of four, but use additional parallelism within each child-node test to better utilize a wider SIMD unit. We present such an approach in this paper. The key idea is to view the 16-wide SIMD hardware not as 16 independent lanes, but rather as *four lanes of four elements each*, and use this to process *four* nodes respectively four primitives in parallel, using 4-wide SIMD for each node/primitive intersection test (see Figure 3). This allows for effectively using wide SIMD for a single ray without having to rely on an inferior data structure, and with less strain on the memory system than the *N-independent-rays* would produce. We show that on the Intel® Knights Ferry PCI card [2] that we use for evaluation, this technique outperforms packet tracing for incoherent rays.

This *per-lane* approach also has some overhead, and for coherent computations is not as efficient as processing 16 different rays. Thus, we also extend our technique to a hybrid scheme in which we generate and shade rays in packets, trace them as packets as long as they are coherent, and then, on-the-fly, switch to the single-ray scheme when the rays diverge. On our platform this hybrid scheme is superior to both packet and single ray tracing, and achieves consistently higher performance than either of those techniques.

II. Previous Work

Ray Tracing goes back to the seminal work by Appel et al. [8] and Whitted et al. [9], and is often used in the context of advanced rendering algorithms as pioneered by Cook [10], Kajiya [11], etc. For an introduction to ray tracing see for instance [12]. Making ray tracing fast requires the use of a good *acceleration data structure* such as grids, kd-trees, or bounding volume hierarchies (BVHs) (e.g., [13], [14]). Building good acceleration structures is usually done by applying the *surface area heuristic (SAH)* as originally introduced by Goldsmith and Salmon [15], and later refined by lots of others (see, e.g., [13], [16]).

With both CPUs and GPUs using ever wider SIMD designs, researchers had to increasingly focus on *SIMD ray tracing*. For this purpose, Wald et al. [4] proposed the concept of *packet tracing*, which works by traversing *N* different rays in parallel through the acceleration data structure by conservatively descending into subtrees and de-activating those rays that would usually not have traversed a subtree. This method maps well to current CPU designs, but for incoherent rays suffers from low SIMD utilization when too many rays get inactive.

To improve SIMD utilization different researchers have looked into using packets much larger than the SIMD width, and *compacting* the still-active rays at different points in time [17], [18], [19]. A hybrid approach that uses compaction of large ray sets and uses a similar idea of a fall back to single ray tracing in case no coherence is detected was presented by Tsakok et al. [20]. However, compaction usually comes at a not inconsiderable cost (in particular on architectures with relatively small caches), and for really incoherent rays and complex scenes produces only modest benefits.

SIMD single-ray tracing (SSRT) was first applied by Hurley et al. [21] to intersect multiple primitives in SIMD in an otherwise scalar kd-tree traverser, and by Christensen et al. [22]. Applying this same concept to BVHs—where it can be applied to node traversal steps—has been independently proposed by Dammertz et al. [5], Ernst et al. [6] and Wald et al. [7].

The Intel® Many Integrated Core (Intel® MIC) Architecture is a many-core x86 architecture for high performance and throughput computing. It is designed for highly parallel applications which have the highest demands for compute power and memory bandwidth, and the *Knights Ferry* (KNF) Software Development Platform is its first incarnation [2]. The first generally available Intel® MIC product called *Knights Corner* (KNC) has already been announced [2]. Besides significant performance improvements, KNC shares many architectural properties with the KNF, and the algorithmic improvements proposed here will carry over to it.

KNF's many-core processor (called *Aubrey Isle*) consists of 32 x86 cores, each having fully coherent L1/L2 caches (32K/256K), and each running 4 threads for latency hiding. Each x86 core is *superscalar*, having both a scalar and a 16-wide SIMD unit that can both issue in the same cycle (a process called *pairing*). This is favorable for our approach in that shared data can be kept in scalar registers, and their associated scalar operations can often be issued in parallel to other vector instructions.

Of particular interest to this paper, the 16-wide SIMD unit is organized into four *lanes* of four *elements* each, and is programmed using a three operand 16-wide SIMD instruction set that also supports *masking* (for predication); the third operand can be either a register, or a memory address. In case it is a register, the instruction set allows certain *free swizzle*s to be applied to the four elements of each lane (e.g., $xyzw \rightarrow yyyy$, or $xyzw \rightarrow yxwz$); if it is a memory address, it allows for a *free load* from this address (including a 1-to-16 or 4-to-16 broadcast, if desired). Those modifiers are called free because they are an integral part of the given vector instruction, and do not require any additional issue cycles.

III. Algorithm Overview

Intel® MIC's 16-wide SIMD can also be viewed as being organized into four *lanes* of four *elements* each. The core idea of our approach is to exploit this organization to realize an efficient way of traversing single rays through a BVH with a branching factor of *four* (rather than 16, like the SIMD width). This aims at two separate goals:

First, reducing the branching factor from 16 to 4 significantly increases the efficiency of SSRT itself: a sixteen-wide BVH is significantly less effective at culling than a two- or four-wide BVH [7]; even with the best-known build algorithms the average node and leaf utilization is rather low, and having to perform 16-wide reductions after every node test and triangle intersection carries a significant performance overhead (measured to be up to 50% vs. a four-wide BVH).

Second, operating on a four-wide BVH (also called *Quad-BVH* or *QBVH*) allows for performing efficient packet traversal on *exactly* the same data structure used for single-ray traversal (packet traversal is roughly equally efficient for both binary and QBVHs, but significantly less efficient for a 16-wide BVH). This then allows for a hybrid traversal scheme (see Section V) in which we use packet tracing as long as rays are coherent, and switch to single-ray traversal as soon as rays are detected to diverge.

IV. Single-Ray Traversal

For our QBVH-based single-ray traversal approach we view the 16-wide registers as four 4-wide registers, and process four nodes respectively four triangles in parallel using per-lane operations. For example, given a vector A and 4 vectors B_i in `float4` format, we can compute the four dot products $<A,B_i>$ via

```
vA   = load(&A,BROADCAST_4X16);
dot  = mul(vA,free_load(&B,LOAD16));
dot  = add(dot,free_swizzle{yxwz}(dot));
dot  = add(dot,free_swizzle{zwxy}(dot));
```

Since both loads and swizzles are "free" in this code, assuming that `vA` had been in registers already (as a ray would usually be) we would be able to compute these four dot products with only three instructions. This is an up to 4× lower throughput than computing 16 separate dot products in the "packet" (structure-of-arrays layout) approach (which could be done with the same number of instructions)—but is an up to 4× win if most of these 16 elements were inactive.

These free loads and swizzles do come with some restrictions, though: swizzles are free only for intra-lane swizzles, and free loads require data to be aligned to 64 bytes for vector loads, and to 16 bytes for 4-to-16 load-broadcasts, respectively.

A. Data Organization

The key lesson to be learnt from this example is how crucial proper data organization and alignment are to our approach: to reach the 3 instructions we have assumed that A was already in a register, and that all four B_i vectors can be loaded together with a single free load; if this was not the case the cycle count would be significantly higher: For example, if the four B_i came from different memory regions we would require four explicit loads that together would more than double the instruction count.

1) Ray Data: Though we eventually operate on individual rays, rays are generated and shaded in packets, using a structure-of-arrays (SoA) format. For the single-ray box test and triangle test kernels, however, we require a layout in which the respective ray's origin (`org`), direction (`dir`), and pre-computed 1/direction (`rcp`) are pre-loaded into one register each (*xyz* coordinates replicated into all four lanes).

Upon entering traversal we first create an array-of-structures (AoS) copy of the given packet's `org`, `dir`, and `rcp` values, from which we can then load-and-broadcast each ray's data whenever so required. All of the ray's hit data can stay in the original packet format, and modified there whenever an intersection occurs.

2) (Quad-) BVH Nodes: During traversal, we frequently have to perform four ray-box intersection tests with a node's four children (stored as AABB). We perform one node test in each lane, using each lane's elements to process the x,y, and z coordinates in parallel (see Section IV-C.1). Given such a 'lane-based' kernel, a good data layout for a quad-node is to store all four nodes in one compact memory region, with the first cache line dedicated to the four min coordinates, the second to the max coordinates.

```
struct QuadNode {
 struct { float3 min3;uint32 data;   } min[4];
 struct { float3 max3;uint32 unused;} max[4];
};
```

In this layout the four min (and max) coordinates are aligned to 64-byte boundaries, and can be loaded into the lanes of a vector through a single free load. The otherwise unused fourth element in each of the nodes' min lanes stores additional information: Depending on whether the given child node is an internal or a leaf node, which is indicated by the most significant bit in *data*, an offset to either the child or to the first triangle of the respective leaf is stored in the remaining bits. Additionally, the two least significant bits hold the number of triangles in case of a leaf node.

Since this layout forces all inner nodes to have *exactly* four children those nodes with less children have to be padded with empty nodes. Padding introduces a memory overhead of roughly 25% for node memory (see Table I); however, since a QBVH has significantly fewer nodes than a binary BVH we still spend less on node storage than a corresponding binary BVH would have.

3) Triangle Data: The triangle intersection kernel also operates on four different triangles—one per lane—in parallel. To avoid having to gather the four triangles' vertices from different memory locations we use a separate array that stores for each triangle all required data in a single cache line-sized data block. This is similar to Wald et al [4] and Aila et al [3].

```
struct Triangle {
 struct { float3 pos; uint32 quantN; } vtx[3];
 float3 gNormal; uint32 shaderID;
};
```

This triangle record holds the triangle's three vertices (one per lane), leaving the lanes' fourth components of each vertex to store additional data such as vertex index or discretized vertex normal. In the rest of the cache line-sized data block we additionally store the geometry normal and shader ID. Each vertex within the data block is aligned to a 16-byte boundary and can be loaded into a register lane by a single instruction.

Triangles are stored in the same order as they appear in the QBVH's leaves, thus each leaf can address all of its triangles with a single pointer and count value.

Since we store triangles individually we do not have to perform any padding to multiples of four (saving memory). The triangle intersector will always process triangles in groups of four; instead, for partially filled leaves we simply include triangles from a neighboring leaf.

B. QBVH Construction and Quality

Building BVHs with branching factors of more than two has been addressed previously [5], [6], [7], and in fact, any such construction method can be used for our approach. In our implementation, we use a top-down construction: As long as a quad-node has less than four children we take the child with the biggest surface area (thus greedily minimizing the SAH early up in the tree), try splitting it into two, and iterate. Once a quad-node is fully built, we descend into each of its non-leaf children, and build those recursively.

For the splitting process we use a standard SAH binning process [23], [24], [25], with the only exception that we create a leaf as soon as the number of triangles drops to or below 4. In fact, we decided to actually *enforce* a leaf size of four or less by chopping larger leaves into chunks of four (using simply the input order). This case is extremely rare and does not introduce any measurable degradation in QBVH quality. On the other hand, knowing that *all* leaves have four or less triangles, the single-ray traversal kernel can immediately perform a single four-triangle intersection when reaching a leaf, without any additional looping or branching code at all.
The average node and leaf utilization we get from this build is rather high, at $\geq$75% for inner quad-nodes, and over 90% for leaves (see Table I). These numbers compare well to those reported by [7] for a 16-wide BVH (around 65% and 70-75%, respectively) despite their more sophisticated build process.

The expected SAH cost for our QBVH also compares well to a traditional binary BVH: Since we do not perform any additional merge/collapse steps that trade quad-node utilization for BVH quality the only difference to a traditional SAH builder is that we use a leaf threshold of four rather than always splitting until the SAH itself indicates a leaf. Even when applied to a binary BVH this modification degrades SAH cost by only about 5%; while at the same time reducing the number of nodes by roughly half (see Table II). Our QBVH (QVH_{LT4}) has exactly the same number of leaves as the binary BVH with leaf threshold of 4 (BVH_{LT4}), but in addition has 33-40% fewer internal nodes, resulting in an even lower SAH cost and node count (see Table II). These statistics indicate that our data structure does not have any significant overhead compared to a traditional binary BVH; on the contrary, it should be at least competitive even *without* applying our SIMD single-ray tracing scheme.

Scene	tris	quad-nodes w/ #children			avg util. of	
		2	3	4	quad-node	leaf
fairy	174k	8k	4k	11k	78%	90%
conference	282k	13k	6k	16k	78%	92%
hairball	122k	6k	3k	7k	77%	90%
car	4355k	200k	95k	270k	78%	90%

TABLE I. Average quad-node/leaf utilization for our constructed QBVHs and distribution of partly filled quad-nodes/leaves. Average quad-node utilization is 78% and leaf utilization over 90%.

Scene	BVH_{LT1}		BVH_{LT4}		$QBVH_{LT4}$	
	nodes	SAH	nodes	SAH	nodes	SAH
fairy	213k	33.2	108k	33.3	70.9k	23.5
conference	309k	49.4	178k	51.0	114k	35.4
hairball	2.4m	505.0	1.8m	493.0	1.2m	386.5
car	5.3m	132.0	2.8m	132.5	1.7m	96.6

TABLE II. Relative SAH cost and node count for a traditional *binary* BVH without (BVH_{LT1}) and with an explicit leaf/triangle threshold of 4 (BVH_{LT4}) and for our QBVH with a similar threshold ($QBVH_{LT4}$).

C. Traversal and Intersection

In the following we will sketch the three major components of the traversal and intersection: lane-based SIMD node intersection test, lane-based SIMD triangle intersection test, and control flow. The pseudo code for the complete traversal kernel is given in Section VIII.

1) Node Intersection Kernel: The node intersection kernel performs all four box's slabs tests in parallel. For a given box $[b_{min}, b_{max}]$, the slabs test first computes, for each dimension d, the ray's distance to that box' lower and upper bounding planes $t_{lo,d} = (b_{min,d} - org_d) * rcp_d$; $t_{up,d} = \ldots$, then uses these to get the entry and exit distances of this slab $t_{in,d} = min(t_{lo,d}, t_{hi,d})$; $t_{out,d} = max(t_{lo,d}, t_{hi,d})$, and finally computes the ray's final entry and exit distance as the maximum respectively minimum of all entry respectively exit distances $t_{in} = min_d t_{in,d}$, $t_{out} = min_d t_{,d}$.

For our SIMD box test, we process all four boxes in parallel (one per lane), with each lane processing its box' three dimensions:

```
t_lo_d   = mul(sub(org4,free_load(node_min)),rcp4);
t_hi_d   = mul(sub(org4,free_load(node_max)),rcp4);
t_in_d   = min(t_lo_d,t_hi_d);
t_out_d  = max(t_lo_d,t_hi_d);
```

Then, computing the final t_{in}, t_{out} requires only horizontal min/max-reductions inside the four lanes (using free swizzles); and comparing each lane's first element for $t_{in} \leq t_{out}$ gives us the bit mask indicating which of the four boxes had a valid intersection:

```
t_in   = max(t_in_d , t_in_d{yxwz});
t_in   = max(t_in_d , t_in{zwxy});
t_out  = ...
hit_mask = cmple(0x8888, t_in , t_out);
// - hit_mask = 0x8888 & t_in[i] '<=' t_out[i]
```

Note in particular that through our node layout and pre-loading and replicating of the ray, this can all be done with free loads and free swizzles, with no scalar code at all.

2) Triangle Intersection Kernel: As triangle intersection kernel we use a modified variant of the intersection test by Shevtsov et al. [26]. As the QBVH leaves have at most four triangles, we use a kernel that always processes four triangles without any scalar loop code at all (for leaves with less triangles we simply include the next leaf's triangles). Each lane then computes a different triangle's intersection test, using the lane's four elements to process three dimensions in parallel, and using free swizzles for any reduce operations (e.g., in dot products). If at least one lane had a successful triangle intersection, we can use horizontal operations across lanes to determine which of the lanes had the closest intersection; the corresponding lane's values are then written back to the ray's intersection data.

Unlike the box test, our triangle intersection kernel first has to "gather" the four triangles' data (using masked load-and-broadcast's) from the four triangle records sequentially stored in memory (see Section IV-A.3). This costs 16 extra instructions (loading 16×16 bytes) which is between 20-30 % of the total kernel instruction count.

3) Control Flow: On a high level, traversal works by maintaining a stack of yet-to-be-traversed nodes; for each node we store two 32-bit values: the 32-bit `data` field describing the respective node, and the distance to that subtree. Like for a binary BVH we do not maintain a fully sorted stack but instead greedily descend into the child with the closest hit distance, and pushing all others onto the stack. Note that we do *not* sort the pushed nodes by hit distance as in 90% of all traversal steps only 2 or less nodes are hit (see Table III). In this common case picking the closest child is the same as sorting and for those 10% of the cases where more than 2 nodes were hit we *may* push (some) nodes in reverse order. Compared to fully sorting the distances in the 2 or more nodes case, the maximum introduced overhead of 3% additional traversal steps (for the ray distributions in Table III) is negligible.

Stack Compaction: Every time an intersection is found all stack-entries with a distance greater than this hit distance are no longer valid. Rather than discarding such nodes when popping them off the stack we explicitly perform a *stack compaction* every time a leaf yielded a valid intersection: Using Intel® MIC's compaction instructions this is rather cheap.

Efficient Implementation: In a packet tracer, control flow can be amortized over all rays, and its cost is low compared to box and triangle intersection kernels. For single-ray tracing where the four box intersection tests require only a dozen vector instructions, minimizing scalar control flow cost is crucial. Several examples of cutting down on control flow cost have already been mentioned: Using a fixed leaf size of four to remove all leaf loop-code; not sorting pushed nodes; and using stack compaction. In addition, Intel® MIC's aforementioned *pairing* feature offers a potent way of hiding scalar control flow operations. For example, since popping from stack uses only simple scalar operations we can hide any and all popping cost by performing a *speculative* pop operation in parallel to the node intersection test; in which case all scalar pop instructions get paired with the node test's vector instructions (a specially marked *sentinel* leaf-node is used to mark the bottom of the stack). If the hit-mask is zero we can directly continue the top-down traversal with the already popped node.

If the hit-mask is non-zero, we first count the number of bits set in this mask. Looking at the probability distribution of how often a node has 0,1,2,3, or 4 intersections (see Table III) we can then optimize for the most common case of 2 or less nodes by using a separate code path that determines the closest child with at most one (scalar) comparison rather than a full cross-lane reduce. A costly cross-lane vector reduction to find the closest child is then only needed in the unlikely "3 or more" path. Obviously we also design the code to pair the common code path wherever possible.

# hit nodes	shading	0	1	2	3	4
conference	2-bounce diffuse	**22.5**	**42.4**	**25.2**	7.9	2.0
conference	8-bounce diffuse	**22.1**	**43.0**	**25.3**	7.8	1.7
hairball	ambient occlusion (AO)	**17.5**	**46.0**	**25.5**	7.3	3.7

TABLE III. Distribution (in %) of number of child nodes hit during single-ray traversal for different ray distributions and scenes. Roughly 90% of traversals hit two or less nodes.

D. Performance

The ultimate goal of our single-ray tracing scheme is to be used only for handling incoherent rays in a hybrid tracing scheme. Before looking into such a scheme it is helpful to first analyze the performance of the single-ray kernel itself. In Table IV we compare our single-ray kernel's performance to a QBVH-based packet tracer (16 rays per packet). For all ray distributions tested, the maximum difference in performance between using a binary BVH and a QBVH for packet tracing has been measured to be less than 3-4% and is therefore negligible. In each traversal step the QBVH-based packet tracer tests 16 rays against each of the four nodes. For the triangle intersection, 16 rays are tested against one triangle.

For the sake of simplicity, in this preliminary evaluation we intentionally compare only the two most extreme cases of ray distributions: highly coherent primary rays, and highly incoherent rays from an 8-bounce diffuse path tracer (see Section VI-D). To minimize the masking effect of high shading cost both cases use the minimum amount of shading.

	coherent (primary)		incoherent (8-bounce-diffuse)	
	P/S	S/P	P/S	S/P
hairball	**2.43x**	0.41	**0.60x**	1.65
fairy	**3.44x**	0.29	**0.58x**	1.71
conference	**3.84x**	0.26	**0.71x**	1.40

TABLE IV. Performance ratios between single-ray (S) and packet tracing (P), for coherent (primary) rays and incoherent (8-bounce diffuse) rays. Single-ray tracing outperforms packet tracing for incoherent rays, but is significantly slower for coherent rays.

Not unexpectedly, Table IV confirms that our single-ray tracing scheme is significantly faster than a packet tracer for incoherent rays, but significantly slower for coherent rays. For coherent rays, the packet tracer is hard to beat: it utilizes almost every SIMD element in every vector instruction, it can amortize addressing overhead, control flow, etc. and overall makes very good use of the hardware. For coherent rays, the single-ray scheme is significantly slower ($2.4-3.8\times$): even in the best of cases we use only 12 of 16 vector elements (4 lanes times 3 dimensions), and despite all our efforts to reduce and hide control flow cost wherever possible a significant cost still remains (at 11 instructions for four box tests even a handful of cycles for control are significant!). For incoherent rays, however, the single-ray scheme performs just as well as for coherent rays, and eventually outperforms packet tracing by $1.4-1.7\times$.

V. Hybrid Packet/Single-Ray Tracing

As just shown the single-ray scheme itself is faster than a packet tracer for incoherent rays, but slower for coherent ones. However, for practically relevant applications rays are neither fully coherent nor fully incoherent. Our solution to this is to use a hybrid scheme in which packets are used for coherent rays, and single rays for incoherent ones. At its simplest, this can be achieved by *manually* calling packet code for rays that are *known* to be coherent (such as primary rays, and primary shadow rays to a point light), and single-ray code for everything else.

However, while some rays are typically coherent (or not), others are more difficult to classify: For example, many but not all of the reflection rays in the car model (see Figure 1) are coherent; the degree of coherence for shadow rays to an area light source depends on the size of this light source; and even shadow rays to an HDRI environment map depend on how diffuse vs. directed that illumination is. Consequently, a more automatic way of combining these techniques is desirable. Fortunately, since our data structure can be used for *both* packet and single-ray tracing we do not actually have to *guess* a packet's coherence at all, and can, in fact, switch between packet tracing and single-ray tracing at any time in mid traversal.

We start out with a traditional, 16-wide packet traversal, performing 16-wide box tests and updating a given stack of yet-to-be traversed subtrees as usual. At any point in time, counting the bits in the active mask tells us how many of the packet's rays are still active for this subtree. If this number falls below a given threshold, we leave the packet traversal code and, for this subtree, sequentially trace all active rays in single-ray mode.

A. When to Switch?

The most obvious time to check for this switch would be right at the beginning of each traversal step. However, though *relatively* cheap this test introduces an overhead to each and every traversal step. As it turned out, it is slightly faster to *not* test at all during downward traversal steps, but *only* check when popping nodes and their intersection distances off the stack (the *active* mask after a stack pop is generated by comparing the node distances against the ray intersection distances). This *can* lead to a packet with a single active ray

scene	render mode	traversal packet	traversal single	intersection packet	intersection single	num switches
fairy	primary	85%	15%	80%	20%	0.7
fairy	AO	31%	69%	26%	74%	8.4
conference	8-bounce	28%	72%	19%	81%	12.4
hairball	AO	13%	87%	17%	83%	18.6
fairy	primary	98%	2%	92%	8%	
fairy	AO	71%	29%	48%	52%	
conference	8-bounce	56%	44%	61%	39%	
hairball	AO	60%	40%	33%	66%	

TABLE V. Top: relative number of box and triangle intersection tests done in packet vs. single-ray mode for our hybrid tracing, and average number of switches performed during traversal (switch threshold set to 7). For incoherent ray distributions, the majority of all intersection tests are done in single-ray mode. Bottom: relative distribution of total *per ray* box and triangle intersection tests (taking packet utilization into account). Even for incoherent ray distributions a large fraction of ray box and triangle intersection tests are still done in packet mode.

going down all the way to a leaf node, but nevertheless we have chosen this approach as it is slightly better in practice. The actual switch overhead in terms of instructions is rather low as approximately only a dozen instructions are required to reload either packet or single ray data from cache into registers (intersection data is always kept in cache memory).

The switch to single-ray tracing always applies only to the current subtree; any other subtree popped off the stack at a later time will again decide whether it is to be processed in packet or single-ray form. Thus, we might actually switch back-and-forth between packet and single-ray mode several times during traversal, and subtrees for which the packet is still coherent enough will still be processed in packet form (see Table V).

While the method itself is fully automatic, we still have to determine the parameter at which we switch from packet mode to single-ray traversal mode. To do this, we have taken a variety of scenes and ray distributions, and measured the hybrid algorithm's performance for each of the 16 possible values. Based on the these experiments (Figure 4 shows the results for two example scenes, while all other scenes show similar behavior), we adopt a threshold of 7, which is consistently within 5% of the optimum performance across all ray distributions and scenes.

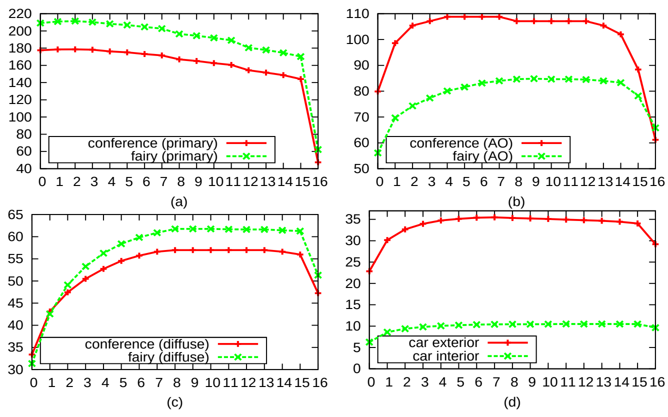


Fig. 4. Performance in Mrays/s (including shading, sampling etc) for varying packet-to-single switch thresholds and different amounts of coherence (see Section VI): a) primary rays (*fairy*, *conference*); b) ambient-occlusion (see Section VI-A) c) 8-bounce diffuse path tracing, and d) path tracing with realistic BRDFs for the *car* interior (incoherent) and exterior (mixed coherence, also see Figure 1). In all those experiments, the switch threshold of 7 provides close ($\leq$5%) to the optimum performance across all ray distributions and scenes.

B. Hybrid Tracing Statistics

To illustrate how this hybrid scheme works in practice, in Table V we show, for a variety of scenes and ray distributions, the percentage of node tests and triangle intersections done in packet vs. single-ray mode. In terms of nominal *number of executions* of these kernels it becomes clear that the majority of kernels is executed in single-ray mode: For box tests, up to 87% and for intersection tests up to 83% of executions are in single-ray mode. Many of the packet ray steps are performed during the initial top-down traversal where rays are still reasonably coherent.
However, these numbers mask the fact that a single-ray kernel always executes exactly one test whereas the packet code may do up to 16 tests per execution (the number of *actual* tests depends on the current packet utilization). When adjusting for this the numbers (see Table V) shift towards packet tracing, showing that there is indeed a significant number of node and triangle tests that are coherent enough to benefit from packet processing. In addition, Table V also shows how the hybrid scheme adapts to varying degrees of coherence (coherent distributions have relatively more packet steps); and that we do in fact switch several times during traversal, yet not as often as to introduce excessive switching overhead.

VI. Results

With all parameters fixed we can now evaluate our methods' performance relative to a packet tracer. All experiments use a Knights Ferry PCI card clocked at 1.2 GHz, screen resolution of 1024×1024 pixels, a primary packet size of 4×4 pixels, and a switch threshold of 7; all performance numbers include ray generation, traversal, shading, sampling, etc. As test scenes, we use the *conference*, *fairy*, *hairball*, and *car* models as shown in Figures 1 and 7.

A. Ambient Occlusion

In our first experiment we have each primary ray generate 16 ambient occlusion (AO) rays, traced in 16 successive packets. The AO rays' directions are generated by sampling the cosine-weighted hemisphere. AO rays do not use a maximum distance; instead, rays are traced until they either hit something or leave the scene.

Scene	Performance (Mrays/s)			Speedup		
	(P)acket	(S)ingle	(H)ybrid	S/P	H/P	H/S
fairy	56.9	65.6	89.1	1.15x	**1.56x**	1.35x
conference	79.2	61.4	114.9	0.77x	**1.45x**	1.87x
hairball	13.5	14.6	19.2	1.08x	**1.42x**	1.31x

TABLE VI. Performance in Mrays/s for Ambient Occlusion shading with 16 samples (+1 primary ray) per pixel.

AO rays are not actually that incoherent: All rays start at a similar locations, and though they eventually do diverge, at least part of their traversal is coherent. Consequently, Table VI shows that packet tracing still performs rather well (in particular for the *conference* scene, which contains lots of large polygons), and pure single-ray tracing can at most compete with packet tracing. *Hybrid* however can adapt to the coherence, and consistently outperforms both schemes, with up to 56 percent compared to packet tracing.

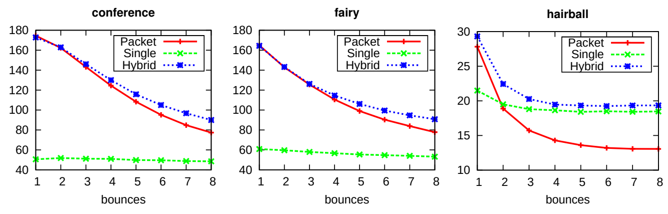


Fig. 5. Absolute performance for N specular bounces in Mrays/sec for *conference*, *fairy*, and *hairball* with $N = 1..8$. Hybrid tracing consistently outperforms both single and packet tracing.

B. N-Bounce Specular

In addition to ambient occlusion we also measured performance for an artificial "N bounce specular" distribution where all rays are specularly reflected N times. For this distribution, coherence depends strongly on scene type: in scenes with low surface variation (*conference* and *fairy*) rays will stay reasonably coherent for several bounces, while in *hairball* even the first bounce diverges wildly.

As can be seen in Figure 5, for *conference* and *fairy* and up to 8 bounces packet tracing still performs rather well, while for *hairball* even single-ray eventually performs better than packet tracing. In all scenes, *hybrid* is on par with packet for small reflection depths and consistently outperforms it for larger depths.

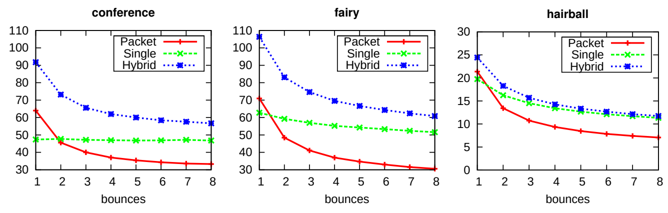


Fig. 6. Absolute performance for N diffuse bounces in Mrays/sec for *conference*, *fairy*, and *hairball* with $N = 1..8$. Hybrid tracing consistently outperforms both single and packet tracing.

C. N-Bounce Diffuse

As an example for truly incoherent rays we also included an N-bounce diffuse path tracer, where each ray performs N diffuse bounces (no russian-roulette termination, and everything is diffuse). For this experiment Figure 6 shows that even single ray tracing starts outperforming packet tracing for just two bounces.

The hybrid approach is even faster, outperforming packet tracing by up to $2\times$ (see Table VII), and, except for being slightly slower (less than 4%) for primary rays consistently performs best for all scenes and ray depths.

Scene	Performance (Mrays/s)			Speedup		
	(P)acket	(S)ingle	(H)ybrid	S/P	H/P	H/S
	2 diffuse bounces					
hairball	13.4	16.2	18.2	1.20x	**1.35x**	1.12x
fairy	48.4	59.2	83.1	1.22x	**1.71x**	1.40x
conference	47.8	50.0	77.1	1.04x	**1.61x**	1.54x
	8 diffuse bounces					
hairball	7.0	11.2	11.7	1.60x	**1.67x**	1.04x
fairy	30.4	51.6	60.8	1.69x	**2.0x**	1.17x
conference	34.9	49.0	60.3	1.40x	**1.72x**	1.23x

TABLE VII. Performance in Mrays/sec for 2 and 8 diffuse bounces.

Maybe most interesting, hybrid tracing performs measurably better than single-ray tracing even for 8 diffuse bounces, proving that it can extract some amount of coherence even from this near-random ray distribution.

D. Realistic Path Tracing

In addition to these artificial ray distributions we also integrated our kernels into a highly realistic path tracer operating on a non-trivial car model of 4.3 million triangles, rendered with a variety of realistic BRDFs (glass, car paint, chrome, and lambertian), with depth of field, and with illumination from an importance-sampled HDRI environment map. To properly handle the complex glass bodies we use a maximum path length of 13; to avoid an excessive number of diffuse bounces we use russian-roulette termination for paths whose accumulated path weight drops below 15%. Sampling is done with padded-replication sampling [27] using a scrambled Hammersley pattern (16 paths per pixel).

This example is particularly interesting in that it is not only more complex and a more representative workload (e.g., in terms of shading and sampling), but also in that it generates an interesting mix of different ray distributions in a single scene: From the outside, the glass as well as the interior seen through the window are challenging, but the reflections off the car paint are rather coherent, and even the diffuse shadows are relatively well-behaved. On the interior, the same scene generates ray distributions that are vastly more challenging. In addition, thanks to depth-of-field and high tessellation density even the primary rays are not perfectly coherent.

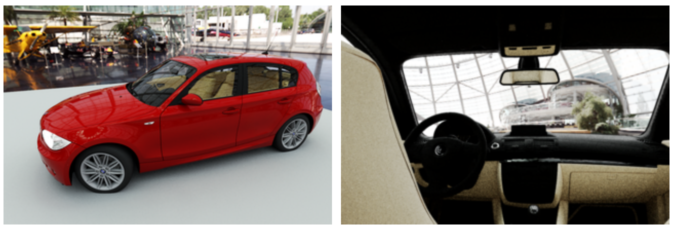


Fig. 7. A realistic path tracer operating on the *car* model with a variety of BRDFs (glass, car paint, Blinn, Lambertian), rendered with depth of field and HDRI environment lighting.

Table VIII shows that for these kind of ray distributions the single-ray scheme clearly outperforms the packet tracing approach, from 26% for the exterior, to 60% for the interior. Again, the hybrid scheme provides even better performance, achieving speedups of 68-80% for constant ambient illumination, and 39-62% for HDRI lighting. Compared to single ray tracing, the hybrid scheme achieves speedups of 8-33%. It is worth mentioning that these speedups are for "full-frame" rendering times even though this application spends a non-trivial amount of time in sampling and shading, e.g. the HDRI sampling alone costs roughly 30-40% of the render time.

VII. Summary And Discussion

In this paper, we have introduced two different, but related, techniques: First, we have proposed a SIMD single-ray tracing scheme for the Intel® MIC architecture that exploits its *four lanes of four* SIMD elements design to efficiently realize single-ray *Quad*-BVH traversal on a 16-wide SIMD architecture.

Second, we have introduced a hybrid traversal scheme that automatically uses packet tracing for coherent rays, and single-ray tracing for incoherent ones. Being able to use exactly the same data structure for both packet and single-ray tracing not only avoids the need for maintaining two separate structures, it also allows our hybrid scheme to switch back and forth in mid traversal, allowing to pick the best traversal mode not only on a per packet basis, but even for different subtrees traversed by the same packet.

View	fps			Speedup		
	(P)acket	(S)ingle	(H)ybrid	S/P	H/P	H/S
	Constant Ambient Illumination					
exterior	0.50	0.63	0.84	1.26x	**1.68x**	**1.33x**
interior	0.10	0.16	0.18	1.6x	**1.8x**	**1.25x**
	Including HDRI lighting					
exterior	0.36	0.42	0.50	1.16x	**1.39x**	**1.19x**
interior	0.08	0.12	0.13	1.50x	**1.62x**	**1.08x**

TABLE VIII. Performance in frames per second for path-traced (using a maximum path length of 13) car exterior and interior (see Figure 7), with ambient illumination (all directions equally important) or importance-sampled HDRI illumination.

A. Discussion

In this paper, we have considered the per-lane SIMD approach only for the special case of single-ray QBVH traversal. While we believe that the general concept carries further than that, how exactly to use it for other applications—or for other SIMD architectures—remains to be investigated in more detail. However, an extension to handle packets with more than 16 rays is straightforward. Whether such larger packets are efficient for tracing incoherent ray paths is rather questionable as a large fraction of the L1/L2 cache will be consumed for just holding ray data.

Our scheme is currently implemented using Intel® C++ Compiler intrinsics (quite similar to [28]). This is somewhat more laborious than coding in an auto-vectorizing language like OpenCL, but otherwise works in exactly the same way as on any other SIMD CPU. The intrinsics code such generated can also be wrapped in a library and be made available to high-level languages in a transparent fashion.

Mapping the hybrid algorithm to other hardware platforms using high-level languages like OpenCL should be possible. The packet tracing part maps well and the cross-SIMD operations in the single ray tracing part could be emulated by using shared memory to exchange data between SIMD lanes. If the underlying hardware offers hardware support for a general *permutation* of SIMD lanes within a register, the storing and loading to and from shared memory could be omitted completely. However, a detailed evaluation of the efficiency of our approach for other hardware architectures is beyond the scope for this paper.

Performance-wise, despite accelerating incoherent rays by up to 2× (vs. packet tracing), such rays are still about 2× slower than coherent ones. This, is quite in line with other architectures [3], but might still leave room for further improvements.

On the upside, the resulting scheme is fully automatic, and does not need any user intervention at all. In terms of performance our scheme combines the best of two worlds: it is much faster than single-ray tracing and equally fast as packet tracing for coherent rays, and up to $2\times$ faster than packets ($1.2-1.3\times$ faster than single ray tracing) for incoherent rays; and for all tested scenes and ray distributions is faster than either of those two techniques.

As previously mentioned KNF shares many architectural properties with KNC, and we therefore believe that our approach is directly applicable to KNC and will provide the same benefits.

B. Future Work

It would be interesting to also map our technique to the 8-wide AVX instruction set that is already available on mainline CPUs. For applications where a certain amount of coherence is available it also looks promising to further extend our techniques to start with packets much larger than SIMD width: ultimately, one could even start with aggressive frustum traversal techniques, gradually fall back to packets (possibly including re-packing), and eventually fall back to single-ray tracing. Most interestingly, this approach would also allow some re-packing for shading.

VIII. Pseudo Code

Simplified pseudo C++ code for switching between packet and single ray tracing, and for the optimized single ray traversal kernel:

```
#define SIMD_UTIL_SWITCH_THRESHOLD 7
mic_f  // - vector SIMD class for 16 floats
mic_i  // - vector SIMD class for 16 ints
mic3f  // - vector SIMD class for 3 x 16 floats
mic_m  // - class for the 16-bit mask type
// - utility functions
c = sel(mask,a,b);   // - c[i] = mask[i] ? a[i] : b[i]
b = per_lane_max(a); // - per-lane max : two max ops + swizzle
b = per_lane_min(a); // - per-lane min : two max ops + swizzle
a = toAOS4f(i,b);    // - SoA to AoS, mic3f -> mic_f
// - node and distance stack for the ray packet
int stack_node_p[MAX_STACK_DEPTH];
mic_f stack_dist_p[MAX_STACK_DEPTH];
// - node and distance stack for a single ray
int stack_node_s[MAX_STACK_DEPTH];
float stack_dist_s[MAX_STACK_DEPTH];
// - ray packet data
mic3f origin, direction, rcp_direction;
mic_f max_dist;   // - max intersection distance
mic_i triangleID; // - intersection primitive ID
mic_m m_active;   // - mask for active rays
// - dummy node to remove branch in inner-loop
stack_node_p[0] = -1;
stack_dist_p[0] = infinity;
stack_node_s[0] = -1;
stack_node_p[1] = qbvh_root; // - QBVH root node
stack_dist_p[1] = sel(m_active,epsilon,max_distance);
int sindex = 2; // - stack index
while (1) {
  stack_index--;
  int curNode = stack_node_p[sindex];
  mic_f dist = stack_dist_packet[sindex];
  mic_m m_dist = lt(dist,max_distance);
  if (curNode == (int)-1) break; // - curNode == dummy node
  if (m_dist == 0) continue;
  if (countbits(m_dist) <= SIMD_UTIL_SWITCH_THRESHOLD) {
    // - SINGLE RAY TRACING
    int ray_index = -1;
    while((ray_index = bitscan(ray_index,m_dist)) != -1) {
      // - Extract and convert data for single ray from
      // - SoA to AoS layout and pre-load into registers
      mic_f org_aos  = toAOS4f(ray_index,origin);
      mic_f dir_aos  = toAOS4f(ray_index,direction);
      mic_f rdir_aos = toAOS4f(ray_index,rcp_direction);
      mic_f dist_aos = max_dist[ray_index];
      ...
     }
   }
  else {
    ... // - PACKET RAY TRACING
   }
}

// === fast QBVH single ray traversal code ===
while (!isLeaf(curNode)) {
  QuadNode* qptr = childPtr(qbvh,curNode);
  mic_f minXYZ = (free_load(qptr->min) - org_aos) * rdir_aos;
  mic_f maxXYZ = (free_load(qptr->max) - org_aos) * rdir_aos;
  // - speculative stack 'pop', 'sindex' is stack index
  curNode = stack_node_single[--sindex];
  // - minimum/maximum x,y,z slabs and distance
  mic_f minXYZ_min = sel(0x7777,min(minXYZ,maxXYZ),dist_aos);
  mic_f maxXYZ_max = sel(0x7777,max(minXYZ,maxXYZ),max_dist_aos);
  // - each lane set to minimum/maximum of x,y,z, and distance
  mic_f near4 = per_lane_max(minXYZ_min);
  mic_f far4  = per_lane_min(maxXYZ_max);
  mic_m m_lr_hit = le(0x8888,near4,far4);
  mic_f near4_min = sel(m_lr_hit,near4,infinity);
  if (m_lr_hit == 0) continue; // - no hit
  int pos_first = bitscan(m_lr_hit);
  int num_m_lr_hit = countbits(m_lr_hit);
  sindex++;
  curNode = ((int*)b_min)[pos_first];
  if (num_m_lr_hit == 1) continue; // - just single hit
  int pos_sec = bitscan(pos_first,i_lr_hit);
  if (num_m_lr_hit == 2) { // - two hits
    int dist_first = ((int*)&near4)[pos_first];
    int dist_sec   = ((int*)&near4)[pos_sec];
    int node_first = curNode;
    // - compare as integer
    if (dist_first <= dist_sec) {
      int node_sec = ((int*)b_min)[pos_sec];
      stack_node_single[sindex] = node_sec;
      ((int*)stack_near_single)[sindex] = dist_sec;
    } else {
      int node_sec   = ((int*)b_min)[pos_sec];
      stack_node_single[sindex] = curNode;
      ((int*)stack_near_single)[sindex] = dist_first;
      curNode = node_sec;
    }
    sindex++; continue;
  }
  // - 3 or 4 hits, find closest first, push others onto stack
  mic_f child_min_dist = min_across_4lanes(near4_min);
  mic_m m_child_min_dist = eq(m_lr_hit,child_min_dist,near4);
  int pos = bitscan(m_child_min_dist);
  mic_m m_pos = andnot(m_lr_hit, // - clear first bit set
               andnot(m_child_min_dist,m_child_min_dist-1));
  mic_i b_min_node = load16i((int*)qptr->min);
  curNode = ((int*)b_min)[pos];
  // push all other nodes onto stack
  packstore16i(m_pos,&stack_node_single[sindex],b_min_node);
  packstore16f(m_pos,&stack_near_single[sindex],near4);
  // - increase stack index
  sindex += countbits(i_lr_hit) - 1;
 }
}
```

References

[1] Intel Corp., "AVX Extensions," http://software.intel.com/en-us/avx, 2011.

[2] Intel MIC, "Intel Many Integrated Core Architecture," http://download.-intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf, 2010.

[3] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of High Performance Graphics 2009*, 2009.

[4] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive Rendering with Coherent Ray Tracing," *Computer Graphics Forum*, vol. 20, no. 3, pp. 153–164, 2001, (Proceedings of Eurographics 2001).

[5] H. Dammertz, J. Hanika, and A. Keller, "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays," in *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)*, 2008, pp. 1225–1234.

[6] M. Ernst and G. Greiner, "Multi Bounding Volume Hierarchies," in *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 2008, pp. 35–40.

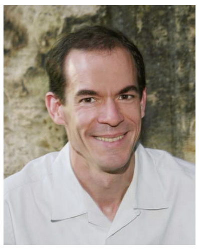
[7] I. Wald, C. Benthin, and S. Boulos, "Getting Rid of Packets: Efficient SIMD Single-Ray Traversal using Multi-branching BVHs," in *Proc. of the IEEE/EG Symposium on Interactive Ray Tracing*, 2008, pp. 49–57.
[8] A. Appel, "Some Techniques for Shading Machine Renderings of Solids," in *AFIPS Conference Proceedings*, vol. 32, 1968, pp. 37–45.
[9] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.
[10] R. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Proceedings of SIGGRAPH '84*, pp. 137–144, 1984.
[11] J. T. Kajiya, "The Rendering Equation," in *Computer Graphics (Proceedings of ACM SIGGRAPH)*, vol. 20, 1986, pp. 143–150.
[12] M. Pharr and G. Humphreys, *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufman, 2004.
[13] V. Havran, "Heuristic Ray Shooting Algorithms," Ph.D. dissertation, Faculty of Electrical Engineering, Czech TU in Prague, 2001.
[14] P. Shirley and R. K. Morley, *Realistic Ray Tracing*, 2nd ed. A K Peters, 2003, ISBN 1-56881-198-5.
[15] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE CG and App.*, vol. 7, no. 5, pp. 14–20, 1987.
[16] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the Art in Ray Tracing Animated Scenes," in *State of the Art Reports, Eurographics 2007*, 2007.
[17] C. P. Gribble and K. Ramani, "Coherent Ray Tracing via Stream Filtering," in *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 2008, pp. 59–66.
[18] S. Boulos, I. Wald, and C. Benthin, "Adaptive Ray Packet Reordering," in *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 2008, pp. 131–138.
[19] R. Overbeck, R. Ramamoorthi, and W. R. Mark, "Large Ray Packets for Real-time Whitted Ray Tracing," in *Proc. IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 41–48.
[20] J. Tsakok, "Faster Incoherent Rays: Multi-BVH Ray Stream Tracing," in *Proceedings of High Performance Graphics 2009*, 2009, pp. 151–158.
[21] J. Hurley, A. Kapustin, A. Reshetov, and A. Soupikov, "Fast Ray Tracing for Modern General Purpose CPU," in *Proc. of GraphiCon 2002*, 2002.
[22] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali, "Ray Tracing for the Movie 'Cars'," in *Proc. IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 1–6.
[23] V. Havran, R. Herzog, and H.-P. Seidel, "On the fast construction of spatial hierarchies for ray tracing," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 71–80.
[24] W. Hunt, G. Stoll, and W. Mark, "Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.
[25] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Experiences with Streaming Construction of SAH KD-Trees," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.
[26] M. Shevtsov, A. Soupikov, A. Kapustin, and N. Novorod, "Ray-Triangle Intersection Algorithm for Modern CPU Architectures," in *Proceedings of GraphiCon 2007*, 2007, pp. 33–39.
[27] T. Kollig and A. Keller, "Efficient Bidirectional Path Tracing by Randomized Quasi-Monte Carlo Integration," in *Monte Carlo And Quasi-Monte Carlo Methods 2000*, 2002, pp. 290–305.
[28] Intel LRBni, "C++ Larrabee Prototype Library," http://software.intel.com/en-us/articles/prototype-primitives-guide/, 2009.

Carsten Benthin Carsten Benthin is a research scientist at Intel Labs. His research interests include all aspects of ray tracing and high performance rendering, throughput and high performance computing, low-level code optimization, and massively parallel hardware architectures. Carsten received his Diploma in Computer Science and his PhD from Saarland University in Germany.

Ingo Wald Ingo Wald is a research scientist at Intel Labs. He holds a PhD in engineering from Saarland University. After his PhD, he was a post-doctoral research associate at the Max Planck Institute for Informatics in Saarbruecken, Germany, followed by a Research Assistant Professorship at the Scientific Computing and Imaging Institute (SCI) and School of Computing at the University of Utah. His work concentrates on all aspects of real time ray tracing and photo-realistic rendering, high-performance graphics, throughput computing, and parallel/high-performance hardware architectures.

Sven Woop Sven Woop is a research scientist at Intel Labs. His research interests include computer graphics, parallel programming, programming languages, and hardware design. Before joining Intel in summer 2007, he worked on a ray tracing hardware architecture and a shading language for a real-time ray tracing system. Sven received his Diploma in Computer Science and his PhD from Saarland University in Germany.

Manfred Ernst Manfred Ernst is a research scientist at Intel Labs, where he is leading the Augmented Reality Lab. His primary research interests are photo-realistic rendering, high-performance ray tracing, data compression and scene-graph architectures. Before joining Intel in 2009, Manfred co-founded Bytes+Lights, a company that developed software for CAD data preparation and visualization. Manfred received his Diploma in Computer Science and his PhD from the University of Erlangen-Nuremberg in Germany.



William R. Mark Bill Mark is a senior researcher at Intel. His career has crossed between academia and industry several times, with the unifying theme of working on future-oriented graphics techniques and architectures. Bill led the design of the Cg language at NVIDIA, building on earlier work with collaborators at Stanford University. Most recently Bill has led and collaborated on efforts to develop high-performance visibility techniques beyond the standard Z buffer, including the irregular Z buffer, real-time micropolygon rendering and techniques for efficient real-time ray tracing. This work was begun at the University of Texas at Austin, where Bill was a faculty member, and is continuing at Intel.