Accelerated Single Ray Tracing for Wide Vector Units

Valentin Fuetterling
Fraunhofer ITWM
Competence Center
High Performance Computing
valentin.fuetterling@itwm.fhg.de

Carsten Lojewski
Fraunhofer ITWM
Competence Center
High Performance Computing
carsten.lojewski@itwm.fhg.de

Franz-Josef Pfreundt
Fraunhofer ITWM
Competence Center
High Performance Computing
franz-josef.pfreundt@itwm.fhg.de

Bernd Hamann
Department of Computer Science
University of California, Davis
hamann@cs.ucdavis.edu

Achim Ebert
Department of Computer Science
University of Kaiserslautern
ebert@cs.uni-kl.de

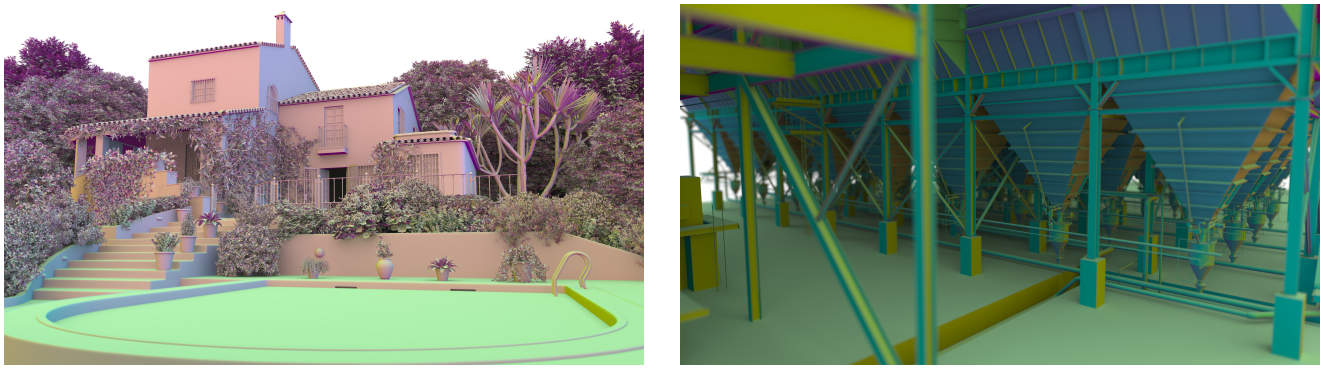

Figure 1: Test scenes rendered with our novel single ray traversal algorithm for wide vector units (WiVe) using diffuse path tracing. On the left side is Villa and on the right side is Powerplant with 37.5M and 12.8M triangles, respectively. Our test implementation of WiVe based on the AVX-512 instruction set delivers 15%-25% increased single ray tracing performance over Embree, shading included.

ABSTRACT

Utilizing the vector units of current processors for ray tracing single rays through Bounding Volume Hierarchies has been accomplished by increasing the branching factor of the acceleration structure to match the vector width. A high branching factor allows vectorized bounding box tests but requires a complex control flow for the calculation of a front-to-back traversal order. We propose a novel algorithm for single rays entirely based on vector operations that performs a complete traversal iteration in constant time, ideally suited for current and future micro architectures featuring wide vector units. In addition we use our single ray technique as a building block to construct a fast packet traversal for coherent rays. We validate our algorithms with implementations utilizing the AVX2 and AVX-512 instruction sets and demonstrate significant performance gains over state-of-the-art solutions.

HPG '17, July 28-30, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5101-0/17/07…$15.00
https://doi.org/10.1145/3105762.3105785

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**;

KEYWORDS

ray tracing, spatial data structure, bounding volume hierarchy, vectorized data processing

ACM Reference format:
Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, Bernd Hamann, and Achim Ebert. 2017. Accelerated Single Ray Tracing for Wide Vector Units. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 9 pages.
https://doi.org/10.1145/3105762.3105785

1 INTRODUCTION

A bounding volume hierarchy (BVH) is a structure often used for efficient ray tracing when considering global illumination effects. Efficiency is especially important in the film and games industry. For over a decade ray tracing research has been focused on mapping traversal and intersection routines onto evolving hardware architectures and instruction sets. In particular the data parallel vector units on CPUs and GPUs have sparked the development of new algorithms. The reason is that the basic traversal of a single ray through a binary BVH offers very little data parallelism. Two

fundamental approaches have emerged from the efforts to remedy the situation, tracing multiple rays simultaneously, and increasing the branching factor of the BVH. Both approach are efficient only if certain constrains apply. Tracing multiple rays requires the availability of multiple rays in the first place which is often not convenient for applications to provide. Ideally rays that are traced together are coherent with respect to the traversal path through the BVH which is difficult to achieve in practice. Increasing the branching factor of a BVH provides data parallel calculations for single rays but reduces culling opportunities and complicates the computation of a front-to-back traversal order. The sweet spot for the branching factor appears to be between two and eight, depending on the hardware architecture and the implementation used. The two approaches have also been combined for hybrid traversal methods. For real world applications the most relevant approach by far is single ray traversal of multi-branching BVHs due to its straightforward integration into complex shading pipelines. Industry-driven development of ray tracing libraries for both CPUs [Wald et al. 2014] and GPUs [Parker et al. 2010] has led to highly optimized traversal implementations for current micro architectures which are challenging to compete with. However the introduction of new hardware features such as AVX-512 sometimes makes the efficient implementation of novel algorithmic ideas feasible.

Our contribution is a novel single ray traversal algorithm which maps all relevant traversal operations to vector instructions, including front-to-back ordering for multi-branching BVHs with branching factors of 8 (BVH8) or higher. The benefit is significantly reduced algorithm complexity and constant-time execution, which is ideal for current and future wide vector micro architectures. In addition we use our single ray algorithm as a building block to construct a fast packet-based traversal for coherent rays. We validate our algorithms with implementations utilizing the AVX2 and AVX-512 instruction sets and demonstrate significant performance gains over state-of-the-art solutions.

2 RELATED WORK

A multi-branch BVH [Dammertz et al. 2008; Ernst and Greiner 2008; Wald et al. 2008] reduces the depth of a binary hierarchy by removing intermediate nodes to make it possible in a single traversal step to test multiple children. In addition to increased memory access, coherence and fewer traversal steps this approach enables data-parallel bounding box intersection tests using vector instructions. Computation time spent during a traversal step is shifted from intersection tests to child ordering and stack operations. Since the introduction of the multi-branch BVH approach, CPU performance of single ray tracing has been stagnant. Recent advancements in stack-less methods [Áfra and Szirmay-Kalos 2014] have reduced the size of a ray's traversal state to facilitate massively parallel single ray tracing, but they have not demonstrated performance improvements. On GPUs multiple single rays can be processed efficiently, independently within vector registers [Aila and Laine 2009] and traced via a binary BVH.

Ray packets [Wald et al. 2001] offer another possibility to map ray tracing to vector instructions by processing a different ray in each vector element simultaneously. Bounding volumes for ray packets such as frustums, intervals and corner rays [Boulos et al. 2006] can be used to cull nodes conservatively to reduce the number of ray-bounding box intersections. Rays within a packet must be coherent, i.e. follow the same traversal path in order to be active at the same time. Such behavior is frequent for camera rays or shadow rays towards small light sources, but not necessarily for higher order effects. Various packet assembly strategies have been investigated to apply packets to effects that produce incoherent rays as well [Boulos et al. 2007]. In order to deal with degrading ray coherence as packets descent deeper down the BVH, Benthin et al. [Benthin et al. 2012] proposed to combine packet and single ray traversal by switching between the two modes depending on packet utilization.

Ray streams is a technique focused on incoherent rays, where for every traversal step a stream of single rays is intersected with the same node and partitioned into hit and miss groups. In order to utilize vector registers, rays need to be gathered from the stream which is an expensive operation on many architectures with wide vector units. Combining ray streams with a multi-branch BVH can reduce or remove gather operations [Tsakok 2009]. For front-to-back traversal algorithms have been developed that allow each ray to follow its preferred order through the BVH which is important for culling [Barringer and Akenine-Möller 2014; Fuetterling et al. 2015]

Accessing a node's children in a front-to-back order for a given ray is facilitated by either the *distance heuristic* [Kajiya and Kay 1986] or the *sign heuristic* [Mahovsky 2005], which are both guaranteed to be accurate only in the case of non-overlapping children's bounding boxes. In case of the distance heuristic, children are ordered by the ray's entry point [Kajiya and Kay 1986], whereas for the sign heuristic only the ray direction's signs are considered to chose among precomputed traversal orders [Mahovsky 2005; Wald et al. 2007]. The sign heuristic has been applied to the BVH4 by storing the intermediate binary BVH [Dammertz et al. 2008], by a local look-up table stored within the nodes [Ernst and Greiner 2008], or by a global look-up table returning a compact list of active nodes [Fuetterling et al. 2015]. The distance heuristic has been extended to arbitrary BVH branching factors by Wald et al. [Wald et al. 2008].

3 WIVE SINGLE RAY TRAVERSAL

Single ray traversal algorithms generally can be divided into separate phases, which are: ray setup, inner node traversal, leaf intersection, and the stack pop. Ray setup performs pre-computation and register alignment with the ray data to facilitate efficient execution of the remaining phases. During inner node traversal the ray descents down the BVH until it misses all children of an inner node or encounters a leaf. In the first case, traversal directly proceeds to the stack pop; in the second case, intersection with the leaf's primitives is performed first, reducing the maximum ray distance t_{far} to the closest primitive intersection (if any). The stack pop takes the top node from the stack (if a node exists) and determines whether the corresponding ray entry distance is still within t_{far}. If this is not the case, the next node is taken from the stack. After the stack pop ray traversal continues with inner node traversal.

Inner node traversal is usually most time-consuming; it can be divided into bounding box intersection (slab test), child ordering,

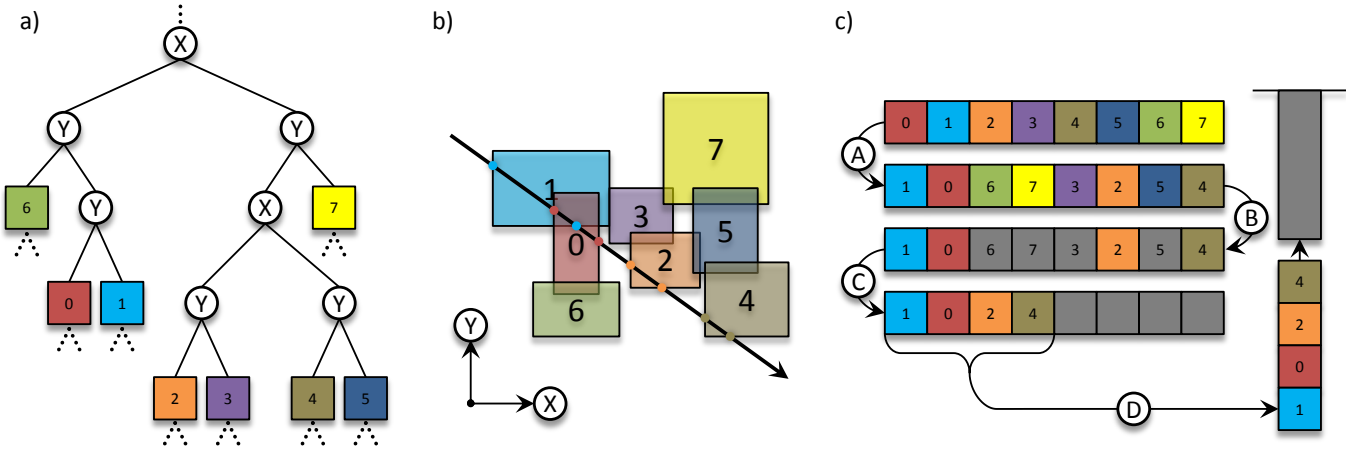

Figure 2: WiVe single ray algorithm for wide vector units. (a) A treelet embedded in a larger binary BVH. Collapsing the treelet yields a BVH8 node cluster (colored squares). The inner nodes of the treelet (disks) are labeled according to the split axis used during binary BVH construction. (b) Bounding boxes of the BVH8 node cluster from (a) in a 2D, xy-coordinate system. A ray with positive x and negative y sign is shown, with marked entry and exit points on the edges of bounding boxes. (c) Ordered traversal for the ray in (b). The initial order of the nodes is based on the order in memory, which is chosen to conform with positive ray signs. Step A performs the node permutation for the ray, which can be deduced from (a) by flipping the children of the Y nodes due to the negative y sign. Step B performs the intersection test resulting in a mask that is applied in step C for compressing the valid nodes into a continuous array. This array is stored on the stack with the next node to be traversed on top.

and stack push. The key idea of the method described here is to map child ordering and stack push operations to permute and compress vector operations, respectively, yielding a traversal algorithm with reduced complexity and increased performance compared to state-of-the-art approaches [Wald et al. 2014]. Instead of ordering children by ray distance we use precomputed front-to-back traversal orders based on the split axes of the BVH construction and ray signs. This sign heuristic has been previously employed only for BVHs with branching factors less than or equal to four.

3.1 Construction of the Multi-branching BVH

A multi-branching BVH is constructed either natively or by collapsing an existing binary BVH. Native construction is faster while collapsing is easier to implement on top of a pre-existing framework [Wald et al. 2008]. Collapsing is illustrated in Figure 2a. The binary BVH is divided into treelets starting at the root node and following branches with largest surface area first. The treelet's intermediate leaves form a node cluster, whereas the treelet's inner nodes are removed after establishment of the front-to-back traversal order. The permutation vector for a particular ray direction's sign combination is derived from the original split axes of the inner nodes. If the split axis of an inner node corresponds to a positive sign, the left branch will be traversed before the right branch; if the sign is negative, the order will be reversed. The example shown in Figure 2 shows a treelet with inner nodes labeled by split axis and intermediate leaves labeled by memory order (a), and the corresponding node cluster together with a ray in positive $x-$ and negative $y-$direction (b). The ordering step (c,A) can be performed manually by swapping the branches in (a) with a $y-$direction split axis.

3.2 Algorithm

We start with a high-level description of our algorithm, referencing the pseudo code in Listing 1. The traversal begins with the initial *node* and tests if it is an inner node or a leaf, i.e. references a node cluster or a primitive cluster, respectively (line 4). In the first case the *traverseCluster* function returns a sorted list of elements referencing intersected children and the corresponding entry distances (line 5). This list is pushed to the stack (line 6). In the second case the ray is intersected with the primitives (line 7), and if a valid intersection exists the stack is compressed (line 8) by keeping only elements with a node entry distance closer compared to the primitive distance. The algorithm continues by checking the stack (line 9) and terminates if the stack is empty. Otherwise the stack is popped and the top element becomes the next node (10).

```
1 def traverseRay(node, ray)
2   stack = {}
3   while(true)
4     if(node.isInner())
5       (elems, num) = traverseCluster(node.cluster, ray)
6       stack.push(elems, num)
7     else if(intersectLeaf(node, ray))
8       stack.compress(ray.tfar)
9     if(stack.isEmpty()) return
10    node = stack.pop(1)
```

Listing 1: Main traversal function for WiVe.

The *traverseCluster* function and the stack push are the key components of the algorithm, illustrated for a BVH8 in Figure 2c, referring to the node cluster and the ray shown in Figure 2b, and the hierarchical representation of the node cluster together with the original split axes as represented in Figure 2a. Child ordering during traversal relies on pre-computed traversal orders for the nodes in

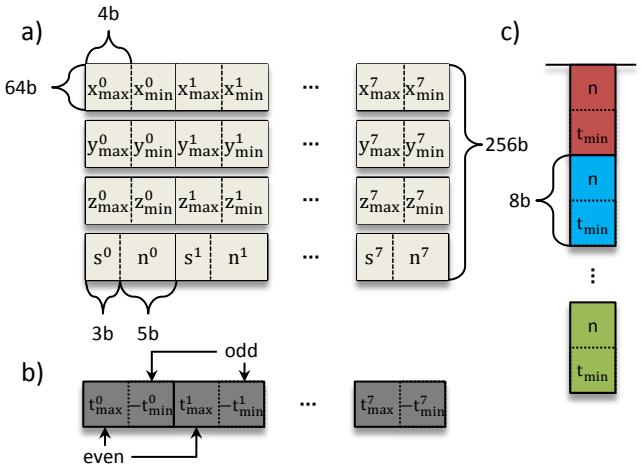

Figure 3: Data layouts. (a) BVH8 node cluster of total size 256 bytes. The nodes' bounding boxes are stored as separate x-, y- and z-vectors with alternating max/min coordinates. A fourth vector contains five bytes for child offset and flags (n) and three bytes for permutation indices (s) per node. (b) Register layout for bounding box intersection. Entry (t_{min}) and exit (t_{max}) distances are computed simultaneously within 8-byte lanes for each node, which requires one to treat t_{min} as negative. (c) Stack layout. Stack elements are 8 byte in size with interleaved child offset and entry distance (t_{min}) for culling. Different colors indicate different nodes.

every node cluster, where the appropriate order for a specific ray is selected based on the signs of the ray's directional components. Since there are the $x-$, $y-$ and $z-$components which can either be negative or positive, $2^3 = 8$ predetermined orders are required per node cluster. A traversal order is represented by a permutation vector that specifies how to re-arrange the nodes with respect to the base order, i.e., the order in which the nodes are laid out in memory. Thus, eight permutation vectors need to be pre-computed and stored with every node cluster. With the permutation vectors in place the node elements in Figure 2c are permuted (A), followed by the slab test producing an active mask (B). The mask is used to compress the active elements into a continuous array (C) which is stored directly on top of the stack (D). A detailed description is provided in Section 3.3

In the case of single-instruction support for the permutation and compression operations, this algorithm has a time complexity of $O(1)$ for both the child ordering and stack push operations compared to the typical $O(n \log n)$ complexity for sorting n active nodes and $O(n)$ complexity for pushing them onto the stack. Even though n is usually small (two or three), the unified treatment of all cases of n makes possible a simpler and more efficient implementation compared to previous approaches.

We note that the WiVe algorithm is general and applicable to any vector architecture supporting the proper permutation and compression operations. It also works for all branching factors in principle (practical up to BVH16).

3.3 Implementation

Our algorithm is suitable for BVH branching factors corresponding to either the full vector width or half the vector width. We provide BVH8 implementations for both variants based on the AVX2 and AVX-512 instruction sets, respectively. Since the AVX-512 instructions map better to WiVe, we focus on the half vector width variant and point out the key differences for the AVX2 version in the last paragraph of this section.

The AVX-512 vector registers are 512 bit in size for computation on 16×32-bit elements or 8×64-bit elements. We fully utilize the single precision floating point capabilities by computing the minimum and maximum distances of the slab test interleaved in the same vector register. In the following we refer to 64-bit elements as node lanes and to lower and upper 32-bit elements within 64-bit elements as even and uneven lanes, respectively (Figure 3b). The instructions key to our algorithm are:

- *vpermq a, b, c*: Copy 64-bit elements from *c* selected by the lower three bit of the 64-bit elements in *b* to the corresponding positions in *a*.
- *vpcompressq a{k}, b*: Select 64 bit elements in *b* using mask *k* and compress the selected elements to form a continuous array aligned to the low element in *a*.

A detailed description of the AVX-512 instruction set is provided in the official documentation [Intel Corporation 2017b].

The memory layout of a BVH8 cluster is illustrated in Figure 3a, which has a standard 256-byte footprint corresponding to four 64-byte AVX-512 vectors. The nodes' bounding boxes are stored in a separate vector for every axis, with alternating maximum and minimum bounds. The fourth vector encodes the permutation vectors and the node data, which includes a flag to indicate an inner node or a leaf, the corresponding child cluster or primitive cluster offset, a mask to identify valid nodes in a child cluster or the number of primitive clusters within a leaf. Permutation indices are three bit in size to reference one of eight nodes, and the eight permutation vectors are compressed into three bytes per node.

Listing 2 shows the *traverseCluster* function which is described in detail in the next paragraphs, referencing the corresponding line numbers.

For the slab test the three bounding box vectors for the $x-$,$y-$ and $z-$axes are loaded into registers (line 2), so that the max/min pairs align with the node lanes, see Figure 3a. If the sign of a ray component is negative, the corresponding max/min values will be swapped within node lanes to conform with the ray's point of view (lines 3-5). The swaps are performed efficiently with masked 64-bit rotate operations, where the mask for every axis is assumed to have been pre-computed during the ray setup phase. After swapping, the slab test is performed for all eight nodes in parallel (lines 6-9) computing t_{max} and $-t_{min}$ in even and odd lanes, respectively, based on the following four equations, utilizing full vector width, see also Figure 3b:

$$
\begin{aligned}
t^{n,i}_{max} &= (b^{n,i}_{max} - o^i) * d^i \\
t^{n}_{max} &= \min_{i=x,y,z} t^{n,i}_{max} \\
-t^{n,i}_{min} &= (b^{n,i}_{min} - o^i) * (-d^i) \\
-t^{n}_{min} &= \min_{i=x,y,z} -t^{n,i}_{min}
\end{aligned} \tag{1}
$$

Here, i and n denote the axis and the node lane, respectively, o^i is a component of the ray origin, d^i is the inverse component of the ray direction, and $b^{n,i}$ represent the minimum and maximum bounding box components after the initial swap. Both o^i and d^i are constant throughout traversal, and the sign of the $d^{n,i} = (-1)^n d^i$ vector can be adjusted during the ray setup phase to alternate between d^i and $-d^i$. The t^n_{max} and $-t^n_{min}$ values are further clipped to the active segment of the ray defined by t_{near} and t_{far}, and the final result is laid out in the vector register as illustrated in Figure 3b.

Next, the slab test results are arranged using the *vpermq* instruction according to the front-to-back traversal order stored in the node cluster. The appropriate permutation vector is extracted (line 10) from the fourth vector shown in Figure 3a by bit-shifting with the concatenated sign bits of the ray direction, i.e., a pre-computed three-bit value, to align the permutation vector's components to the lower three bits of the vector register's 64-bit elements. Following the permutation step (line 11), the $\boxed{t_{max} \mid -t_{min}}$ pairs are ordered such that the first node to be traversed corresponds to the last active node lane in the register.

The slab test is completed by comparing t_{max} and t_{min} to retrieve the active mask, which requires the values to be in separate registers aligned to the odd lanes. This requirement is met via a 64-bit rotate operation to form $\boxed{-t_{min} \mid t_{max}}$ pairs (line 12) and a sign flip with an exclusive or operation to obtain $\boxed{t_{max} \mid t_{min}}$ pairs (line 13). Since $t_{min} \geq 0$ always holds, the predicate of the test $t_{min} \leq t_{max}$ can be determined correctly with integer arithmetic by re-interpreting the floating-point patterns of the pairs as 64-bit signed integers (line 14).

For the stack push operation, the permutation applied to the slab test results is repeated for the node data (line 15), which is then interleaved with the t_{min} values to form 64-bit stack elements (line 16). The stack elements are compressed into a continuous array with the *vpcompressq* instruction (line 17), using the active mask, and stored to the stack (line 19), see Figure 3c. The stack pointer is incremented according to the number of set bits in the active mask (line 18-19). This method supports up to 32-bit node data. If all available 40 bits are required, it will not be possible to interleave the node data with the t_{min} values. Instead, it is compressed and stored separately on a second stack.

Primitives in the leaf nodes are packed into primitive clusters with four primitives per cluster, which we have found to be near-ideal for performance. Larger clusters increase vector utilization at the cost of performing more intersection tests and increased bandwidth demand. Smaller clusters have the inverse effect. Once traversal reaches a leaf node the contained clusters are tested and when an actual intersection is found the maximum ray distance t_{far} is updated accordingly. Thus, nodes on the stack with $t_{far} < t_{min}$ may be removed when such an actual intersection is found (Listing 1, line 8). This pruning procedure is efficiently implemented by loading eight stack elements at a time, starting from the stack bottom, performing the comparison, and compressing and storing the remaining valid elements on the stack.

```
def traverseCluster(cluster, ray)
  (bx, by, bz, node) = cluster.load()
  if(ray.sign.x) bx = swapEvenOdd(bx)
  if(ray.sign.y) by = swapEvenOdd(by)
  if(ray.sign.z) bz = swapEvenOdd(bz)
  tx = (bx - {ray.org.x}) * {-ray.idir.x, ray.idir.x}
  ty = (by - {ray.org.y}) * {-ray.idir.y, ray.idir.y}
  tz = (bz - {ray.org.z}) * {-ray.idir.z, ray.idir.z}
  t = min(tx, ty, tz, {-r.tnear, r.tfar})
  index = shift(node, {ray.sign.xyz})
  t = permute(t, index)
  tmax = swapEvenOdd(t)
  tmin = flipSignsOdd(t)
  mask = compare(tmin, tmax)
  node = permute(node, index)
  elems = interleaveOddOdd(node, tmin)
  elems = compress[mask](elems)
  num = countBits(mask)
  return (elems, num)
```

Listing 2: Core traversal function for WiVe. All local variables are vectors with the exception of *mask* and *num*. The {} operator performs a broadcast of scalar values.

For the full vector width implementation based on AVX2 t_{min} and t_{max} calculations are no longer interleaved in the same vector register. They are performed separately instead which makes the algorithm easier to implement but also requires more instructions. To support this approach, the node layout in Figure 3 is changed to a de-interleaved format corresponding to eight 32-byte AVX2 vectors, and the combined stack is split into separate node and t_{min} stacks. Since AVX2 does not provide a compression instruction, we emulate the operation by a permute instruction and a table look-up using the slab test mask as index. Since the mask is a 8-bit value the table requires 256 entries. The information for a single entry can be compressed into 3 bytes (8×3 bit), encoding eight indices referencing one of eight elements. Decompression is performed by broadcasting an entry into an AVX2 register and shifting each lane by a different amount to align the corresponding index bits.

3.4 Any-hit Traversal

Any-hit traversal is used for shadow rays and can terminate as soon as an intersection with a primitive is found. For this type of query, a front-to-back order is not important and specialized traversal orders are more efficient [Ogaki and Derouet-jourdan 2016]. These specialized traversal orders are identical for all rays and can be encoded into a BVH by arranging the nodes in memory accordingly. The same approach is compatible with WiVe.

3.5 Multi-hit Traversal

Multi-hit traversal [Gribble et al. 2014] attempts to find the first n closest intersections with primitives along a ray. Our WiVe algorithm is compatible with current multi-hit optimization techniques [Gribble 2016]. We expect multi-hit traversal to perform more efficient with WiVe compared to other approaches. The explanation is the fact that limited culling possibilities in a multi-hit scenario lead to a higher number of active children, reducing the performance of other approaches but not WiVe's.

4 WIVE COHERENT TRAVERSAL

Packet tracing [Wald et al. 2001] is an efficient ray tracing technique using vector instructions for groups of coherent rays, i.e., rays that have similar origins and directions. The computational cost per ray when performing packet tracing can be significantly lower compared to single-ray traversal due to efficient vector utilization, amortization of memory access latency, node ordering and stack operations. The optimal packet size is defined by vector width, and, when more coherent rays are available, tracing multiple packets simultaneously can be an additional benefit. Packet tracing can be combined with frustum culling [Wald et al. 2007], reducing the number of bounding box tests considerably during traversal. This section discusses the elegant idea of augmenting packet tracing with our WiVe single ray algorithm for interval arithmetic (IA) culling. The combination is a high-performance coherent traversal method (WiVeC) for the BVH8, while also being applicable to other branching factors.

Applying IA to ray packets generates intervals for the x-,y- and z-coordinates of ray origins and directions for all rays of a packet to perform a conservative rejection test for nodes outside these intervals. The slab test is an IA operation as well, producing the $[t_{min}, t_{max}]$ interval. By expanding the definition of an origin o^i and inverse direction d^i from points and vectors to the intervals $[\underline{o}^i, \overline{o}^i]$ and $[\underline{d}^i, \overline{d}^i]$, respectively, t_{min} and t_{max} can be computed conservatively for a set of rays with the following changes applied to Equation 1:

$$\begin{aligned} t^{n,i}_{max} &= (b^{n,i}_{max} - \underline{o}^i) * \overline{d}^i \\ -t^{n,i}_{min} &= (b^{n,i}_{min} - \overline{o}^i) * (-\underline{d}^i) \end{aligned} \tag{2}$$

Here, we assume that $(\underline{d}^i, \overline{d}^i)$ does not contain 0, i.e., all ray directions in the packet have the same sign combinations. During the set-up phase, a special interval ray is created with maximum and negative minimum values located in the even and odd lanes, respectively, for the origin and the direction vectors. Bounding boxes missed by this interval ray will be missed by the ray packet and can be ignored.

If multiple ray packets are to be traced in parallel, the interval ray must be expanded accordingly and a mechanism is required to track active packets. We achieve this by means of a first packet index (FPI), initialized to the first element in the packet list. Bounding box intersection starts with the FPI packet, and if a valid intersection exists the remaining packets will be assumed to hit the node as well; otherwise, the FPI will be incremented until either the first packet with a valid intersection is found, which will continue traversal, or all packets have been tested, triggering a stack-pop operation. The assumption is that if packets have high coherence the result of a single packet correctly predicts the behavior of the remaining packets, reducing bounding box tests considerably. Wrong predictions will drag uninvolved packets down the BVH and increase the number of bounding box tests instead. This method is well-suited for primary rays.

Listing 3 provides pseudo code for the WiVeC algorithm. The input variable *packets* is a list containing one or more ray packets (line 1). The interval ray is calculated (line 3) to enclose all rays within the packets. If the current node points to a node cluster the *traverseCluster* function defined in Listing 2 is performed on the interval ray (line 6) and the sorted list of active elements is stored to the stack (line 7). The function is slightly modified in the sense that it returns different stack elements compared to those illustrated in Figure 3c. Instead of a direct reference to the child cluster, a reference to the parent node is stored, along with the current FPI. The t_{min} value is not required. If the current node points to a primitive cluster, primitive intersection is performed (line 9). The following loop (line 10) repeatedly pops the stack to retrieve a new node (line 12) until either the stack is empty (line 11) or a valid packet intersection exists (line 16). The *setNext* method (line 13) restores the current FPI from the stack, the *current* method (line 15) returns the packet pointed to by the FPI, and the *next* method (line 17) advances the FPI to the next packet. If only a single packet is traversed, lines 13, 14 and 17 can be omitted.

```
1  def traversePackets(node, packets)
2    stack = {}
3    ray = packets.intervalRay()
4    while(true) outerLoop:
5      if(node.isInner())
6        (elems, num) = traverseCluster^(node.cluster, ray)
7        stack.push(elems, num)
8      else
9        intersectLeaf(node, packets)
10     while(true)
11       if(stack.isEmpty())return
12       (node, fpi) = stack.pop()
13       packets.setNext(fpi)
14       do
15         if(intersect(node, packets.current())
16           goto outerLoop
17       while(packets.next())
```

Listing 3: Main traversal function for WiVeC.

The addition of our WiVe single-ray algorithm to the standard packet traversal approach leads to two improvements: First, the number of bounding box intersections is reduced significantly via IA culling. Second, front-to-back traversal is done implicitly through the stack. The result is a simple and highly efficient technique for multi-branching BVH coherent traversal.

5 RESULTS

We have evaluated our ray traversal algorithm by generating performance data based on our AVX2 and AVX-512 implementations on the dual-socket Intel® Xeon™ E5 2680v3 @ 2.5GHz (HW) and the Intel® Xeon Phi™ 7250 @ 1.4GHz (KNL), respectively. We compare our results with those obtained with Embree 2.15.0 [Wald et al. 2014], the leading high-performance ray tracing library for CPUs. In order to ensure comparability of performance data, we integrated our code into the open source Embree benchmark suite Protoray [Intel Corporation 2017a], which by default offers Embree and Nvidia® OptiX™ [Parker et al. 2010] kernels. A comparison to the GPU-leading OptiX ray tracing library is outside the scope of this paper, however results from the Protoray benchmark have been published elsewhere [Farber 2017]. Embree constructs a native BVH8 using SAH-based centroid binning[Wald 2007], which we directly convert to our own data layout retaining the exact same topology. We have disabled spatial splits [Stich et al. 2009] to ensure better comparability of our results with results obtained with other methods. In order to generate the permutation indices for the

Table 1: Traversal statistics for sign and distance ordering based on Embree's SAH-binned BVH8. The columns *Inner Nodes*, *Leaves* and *Triangles* list the per-ray average numbers of inner nodes visited, leaves intersected, and triangles intersected, respectively. The SAH cost associated with each scene is also broken down by *Inner Nodes* and *Leaves*. The rendered images are shown in Figures 1, 4, 5 and 6.

	Sign			Distance			SAH	
	Inner Nodes	Leaves	Triangles	Inner Nodes	Leaves	Triangles	Inner Nodes	Leaves
Mazda	14.1	3.6	4.1	14.1	3.6	4.1	4.20	2.23
San Miguel	21.2	4.5	5.4	21.2	4.2	5.1	4.01	1.90
Art Deco	11.1	2.3	2.9	11.0	2.2	2.8	4.59	2.74
Powerplant	20.5	5.6	9.3	20.3	5.6	9.2	5.78	4.21
Villa	17.4	4.6	5.5	17.4	4.5	5.4	20.7	15.6

sign heuristic we have modified the Embree code to annotate each node cluster with the original split hierarchy. Triangle intersection is served by the same Möller-Trumbore [Möller and Trumbore 1997] implementation and triangle data structure as used in Embree. Therefore, the traversal algorithms are solely responsible for the observed performance differences. The performance evaluation is based on five scenes consisting of between 5.7M and 37.5M triangles. On the KNL, these benchmarks were processed by all 272 threads, with all data allocated in the high-bandwidth MCDRAM memory segment. The on-chip mesh network was configured in quadrant mode. On the HW all 48 threads were active.

Table 2: Performance in million-rays per second (MRays/s) for our sign-based WiVe algorithm and Embree based on AVX2 and AVX-512 implementations. Rendering is performed at a resolution of 3840×2160 pixels using diffuse path tracing with up to eight bounces. Colors are based on surface normals and the shading cost is included in the results, accounting for 8-12% of the rendering time. The rendered images are shown in Figures 1, 4, 5 and 6.

	Mazda	San Miguel	Art Deco	Powerplant	Villa
# triangles[M]	5.7	10.5	10.7	12.8	37.5
AVX2					
WiVe	**74.0**	**46.0**	**97.2**	**57.4**	**48.8**
Embree	70.9	43.0	93.7	51.9	46.2
WiVe[+%]	**4**	**7**	**4**	**11**	**6**
AVX-512					
WiVe	**126.7**	**73.1**	**165.0**	**85.4**	**87.4**
Embree	110.0	63.2	143.4	68.4	76.3
WiVe[+%]	**15**	**16**	**15**	**25**	**15**

Table 3: Distribution of numbers regarding valid child node intersections (in percent) for a single traversal step.

	0	1	2	3	>3
Mazda	24	39	22	9	6
San Miguel	29	35	19	9	8
Art Deco	30	38	17	8	7
Powerplant	40	24	14	10	12
Villa	25	36	20	11	8

The key comparison between distance and sign heuristics compares how well they approximate a front-to-back traversal order to maximize node culling, see Table 1. The three per-ray average indicators (inner nodes visited, leaves intersected, and triangles intersected) are very close to being equal across the scenes, with a slight bias towards the distance heuristic. A notable discrepancy is observed for SanMiguel, where the number of intersected leaves and triangles is up 6-7% for the sign heuristic, which we attribute to the high degree of overlap of the alpha-textured leaf triangles. In such a setting, the distance heuristic can be more precise as it considers the actual intersection point of the ray. For completeness Table 1 also lists the surface area heuristic (SAH) cost [Goldsmith and Salmon 1987; Wald et al. 2008] associated with each of the generated BVH8s.

Table 2 provides performance data measured in million-rays per second (MRay/s) for a basic diffuse path tracer with up to eight bounces per sample. When comparing the AVX-512 implementations of our sign-based WiVe traversal to the distance-based Embree algorithm, we observe a seizable speed-up of between 15% and 25% across all scenes. The increased efficiency can only originate from the traversal phase since all other parts share the same implementation. This implies that the reduced code complexity due to our novel algorithm is the only significant differentiating factor. Variance in memory access patterns due to slight differences in traversal order between the two heuristics is negligible, which follows from the nearly identical data listed in Table 1. Our algorithm is especially advantageous when rays frequently overlap with more than three children during a traversal step, e.g., in the Powerplant scene. The resulting performance data are shown in Table 3. In this case, the distance heuristic requires increasingly expensive sorting and stack operations while our algorithm's execution is *independent* of the number of active children.

Performance comparison of the AVX2-based implementations of WiVe and Embree demonstrates that WiVe is faster also for the "BVH branching factor equals vector width" variant, albeit with a smaller relative difference of between 4% and 11%. Since Embree also uses an interleaved slab test for AVX-512 and a regular slab test for AVX2, the only notable differences between the WiVe variants is the compress operation and the double stack. The compress operation is emulated due to the lack of hardware support by a sequence of four instruction including a memory access into a sizable table. The double stack is intrinsic to the algorithm and requires an additional compress operations and an additional store to memory.

Finally, we provide performance comparison data for our WiVeC traversal and Embree's hybrid traversal for primary rays in Table 4.

We note that we consider this a valid comparison since WiVeC is executed on a single ray packet at a time, thus exploiting coherence at the same granularity as the hybrid traversal. The results demonstrate a significant and consistent speed-up of between 83% and 142% for WiVeC across all scenes. The culling statistics shown in Table 4 indicate that the frustum ray avoids between 73% and 82% of all node intersection tests, partly explaining these impressive results. The other important aspect, again, is reduced code complexity resulting from our integrated culling and ordering technique. Since we only trace a single ray packet at a time conventional culling implementations would pose a significant overhead. We have chosen high image resolution favoring frustum culling methods due to high ray coherence. Less coherence would reduce culling efficiency and speed-up accordingly. However, this is true for packet tracing in general. As an avenue for future work we imagine that fusing WiVeC and hybrid traversal could be beneficial to further accelerate partly coherent ray packets such as those occurring for shadows and specular effects.

Table 4: Performance in million-rays per second (MRays/s) for our WiVe coherent algorithm (WiVeC) and Embree's hybrid traversal based on AVX-512 implementations. The packet size is 4×4 pixels. An image is rendered at a resolution of 3840×2160 pixels using primary rays. The camera perspectives in the scenes correspond to Figures 1, 4, 5, 6.

	Mazda	San Miguel	Art Deco	Powerplant	Villa
# triangles[M]	5.7	10.5	10.7	12.8	37.5
AVX-512					
WiVeC	**555**	**533**	**796**	**472**	**337**
Culling[%]	73	82	82	78	76
Embree	275	220	409	212	184
WiVeC[+%]	**102**	**142**	**95**	**123**	**83**

6 CONCLUSIONS

Almost ten years ago the multi-branching bounding volume hierarchy led to the last major performance gain in single ray traversal, by utilizing vector instructions for bounding box tests. We have continued and completed the formulation of an innovative, fully vectorized traversal algorithm by introducing the WiVe algorithm. The efficiency gain obtained by WiVe is made possible by transforming node ordering and stack-push operations from conditional scalar execution paths to constant-time vector operations, making them ideal for current and future massively parallel micro architectures. Furthermore, we have introduced WiVeC to accelerate traversal of coherent rays, using the WiVe methodology. We have demonstrated the performance gains resulting from our algorithms with implementations for the AVX-512 instruction set. Our performance data document that we outperform the industry-leading ray tracing library Embree by between 15% and 25%, and by between 83% and 142% on an Intel®Xeon Phi™ CPU. In addition, we have investigated an AVX2 implementation of WiVe on an Intel® Xeon™ CPU that is faster compared to Embree by between 4% and 11% despite limited instruction support. WiVe promises to accelerate single ray traversal for multi-branching bounding volume hierarchies on the GPU as well. We plan to investigate this possibility in the future.

7 ACKNOWLEDGEMENTS

The authors would like to thank Carsten Benthin for his generous support. This research was supported by the German Research Foundation (DFG) as part of the the IRTG 2057 "Physical Modeling for Virtual Manufacturing Systems and Processes".

REFERENCES

Attila T. Áfra and László Szirmay-Kalos. 2014. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum* 33, 1 (2014), 129–140. DOI: https://doi.org/10.1111/cgf.12259

Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 145–149. DOI: https://doi.org/10.1145/1572769.1572792

Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic Ray Stream Traversal. *ACM Trans. Graph.* 33, 4, Article 151 (July 2014), 9 pages. DOI: https://doi.org/10.1145/2601097.2601222

Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R. Mark. 2012. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics* 18, 9 (Sept 2012), 1438–1448. DOI: https://doi.org/10.1109/TVCG.2011.277

Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. 2007. Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007 (GI '07)*. ACM, New York, NY, USA, 177–184. DOI: https://doi.org/10.1145/1268517.1268547

Solomon Boulos, Ingo Wald, and Peter Shirley. 2006. *Geometric and Arithmetic Culling Methods for Entire Ray Packets.* Technical Report. SCI Institute, University of Utah, 2006.

Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum* 27, 4 (2008), 1225–1233. DOI: https://doi.org/10.1111/j.1467-8659.2008.01261.x

Manfred Ernst and Günther Greiner. 2008. Multi Bounding Volume Hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*. 35–40. DOI: https://doi.org/10.1109/RT.2008.4634618

Rob Farber. 2017. Redefining HPC Visualization Using CPUs. (2017). http://www.hpctoday.com/state-of-the-art/redefining-hpc-visualization-using-cpus

Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. 2015. Efficient Ray Tracing Kernels for Modern CPU Architectures. *Journal of Computer Graphics Techniques (JCGT)* 4, 4 (2015), 89–109.

Jeffrey Goldsmith and John Salmon. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. (May 1987). DOI: https://doi.org/10.1109/MCG.1987.276983

Christiaan Gribble. 2016. Node Culling Multi-hit BVH Traversal. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations (EGSR '16)*. Eurographics Association, Goslar Germany, Germany, 85–90. DOI: https://doi.org/10.2312/sre.20161213 arXiv:arXiv:1011.1669v3

Christiaan Gribble, Alexis Naveros, and Ethan Kerzner. 2014. Multi-Hit Ray Traversal. *Journal of Computer Graphics Techniques (JCGT)* 3, 1 (feb 2014), 1–17. http://jcgt.org/published/0003/01/01/

Intel Corporation. 2017a. Embree Protoray. (2017). https://github.com/embree/embree-benchmark-protoray

Intel Corporation. 2017b. Intel® Architecture Instruction Set Extensions Programming Reference. (2017). https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference

James T. Kajiya and Timothy L. Kay. 1986. Ray Tracing Complex Scenes. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 269–278. DOI: https://doi.org/10.1145/15886.15916

Jeffrey Mahovsky. 2005. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies.* Ph.D. Dissertation. University of Calgary.

Tomas Möller and Ben Trumbore. 1997. Fast, Minimum Storage Ray-triangle Intersection. *J. Graph. Tools* 2, 1 (Oct. 1997), 21–28. DOI: https://doi.org/10.1080/10867651.1997.10487468

Shinji Ogaki and Alexandre Derouet-jourdan. 2016. An N-ary BVH Child Node Sorting Technique for Occlusion Tests. *Journal of Computer Graphics Techniques (JCGT)* 5, 2 (2016), 22–37.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David Mcallister, and Martin Stich. 2010. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics TOG* 29, 4 (2010), 1–13. DOI: https://doi.org/10.1145/1833349.1778803

Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial Splits in Bounding Volume Hierarchies. *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)* (2009), 7–14. DOI: https://doi.org/10.1145/1572769.1572771

John A. Tsakok. 2009. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)* (2009), 151–158. DOI: https://doi.org/10.1145/1572769.1572793

Ingo Wald. 2007. On Fast Construction of SAH-based Bounding Volume Hierarchies. *RT'07 - IEEE/ EG Symposium on Interactive Ray Tracing 2007 Proceedings* 1 (2007), 33–40. DOI: https://doi.org/10.1109/RT.2007.4342588

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. *RT'08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings* (2008), 49–57. DOI: https://doi.org/10.1109/RT.2008.4634620

Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* 26, 1 (2007), 1–10. DOI: https://doi.org/10.1145/1186644.1186650

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum* 20, 3 (2001), 153–165. DOI: https://doi.org/10.1111/1467-8659.00508

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33, 4 (2014). DOI: https://doi.org/10.1145/2601097.2601199

APPENDIX

Figure 4: The Mazda scene with 5.7M triangles.

Figure 5: The Art Deco scene with 10.5M triangles.

Figure 6: The San Miguel scene with 10.7M triangles.