Volume 0 (2013), Number 0 pp. 1–11 COMPUTER GRAPHICS *forum*

Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing

Attila T. Áfra[1,2] and László Szirmay-Kalos[1]

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

Abstract

Stackless traversal algorithms for ray tracing acceleration structures require significantly less storage per ray than ordinary stack-based ones. This advantage is important for massively parallel rendering methods, where there are many rays in flight. On SIMD architectures, a commonly used acceleration structure is the multi bounding volume hierarchy (MBVH), which has multiple bounding boxes per node for improved parallelism. It scales to branching factors higher than two, for which, however, only stack-based traversal methods have been proposed so far.
In this paper, we introduce a novel stackless traversal algorithm for MBVHs with up to 4-way branching. Our approach replaces the stack with a small bitmask, supports dynamic ordered traversal, and has a low computation overhead. We also present efficient implementation techniques for recent CPU, MIC (Intel Xeon Phi), and GPU (NVIDIA Kepler) architectures.

Keywords: ray tracing, MBVH, stackless traversal, SIMD processors

ACM CCS: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray tracing

1. Introduction

Ray shooting is an elementary operation in ray tracing that generally involves traversing a hierarchical acceleration structure such as a kd-tree or bounding volume hierarchy (BVH). Ray traversal algorithms can be divided into two main categories: *stack-based* and *stackless* algorithms.

Using a stack for the traversal is typically the most straightforward and efficient approach, especially if the traversal order is dynamic. However, if many rays are traced in parallel, the storage and bandwidth costs of maintaining a full stack for each ray can be very high (i.e., about 256–1024 bytes of memory per ray). Notable examples for this scenario are dynamic ray scheduling algorithms that improve memory access coherence for random ray distributions [PKGH97, NFLM07, AK10, KSS*13]. For such ray tracing methods, stackless traversal is better suited, particularly if the number of resident rays per core is on the order of thousands or more.

In recent years, the BVH has become the most popular acceleration structure thanks to its high performance [SFD09, ALK12], low memory footprint, fast construction [GPM11, KA13], and efficient dynamic updating [KIS*12]. However, all prior work on stackless BVH traversal has focused on traditional binary BVHs, which are not always optimal on certain SIMD architectures (e.g., CPUs).

The *multi bounding volume hierarchy* (Multi-BVH or MBVH) [WBB08, EG08, DHK08] is an N-ary tree that provides higher SIMD utilization for shooting incoherent rays. It stores N bounding boxes or triangles per node, organized into SIMD packets, which can be efficiently intersected in parallel with a single ray. Although the MBVH was originally designed for high branching factors, the same principles can be applied to binary trees as well, improving data- or instruction-level parallelism [Ern11, AL09].

In this paper, we propose a new efficient stackless ray traversal algorithm for MBVHs that supports distance-based ordered traversal without restarts. We add parent and sibling pointers to the tree without necessarily increasing the mem-

ory footprint, and we replace the regular stack with a compact *bitstack*, an integer that fits into one or two machine registers. In the bitstack we store *skip codes* that indicate which siblings of a node must be traversed.

Two variations of the algorithm are presented: one variation for 4-way branching MBVHs (*MBVH4*) and one for binary BVHs having two child boxes per node. We call this kind of binary tree *MBVH2* in order to distinguish it from classical BVHs that store a single box per node. Our method can be extended to even higher branching factors (e.g., 8, 16), but this requires operations on larger bitmasks.

The MBVH4 is primarily used on CPUs with 4-wide or 8-wide SIMD, and also on the recent Intel MIC architecture with 16-wide SIMD. MIC was introduced with Larrabee [SCS*08], and its latest implementation is the Xeon Phi coprocessor [Int13]. On the other hand, the MBVH2 is the preferred choice on current NVIDIA GPUs [AL09, ALK12]. We have optimized our method (Section 6) and evaluated its performance (Section 7) for all these hardware platforms.

2. Related Work

Most previous research on stackless ray traversal targeted either of two widely used acceleration structures: the kd-tree or the binary BVH.

2.1. Stackless kd-tree traversal

One approach for stackless kd-tree traversal is to store *neighbor-links*, also called *ropes*, in the leaf nodes for all six sides, which point to spatially adjacent nodes [MB90, HBŽ98, PGSS07]. During traversal, these links are used to directly jump to the next node (either inner or leaf node) that must be traversed after exiting a leaf node. This eliminates the need for a stack and also decreases the amount of traversed inner nodes, but it has a substantial storage overhead.

Foley and Sugerman [FS05] introduced the *kd-restart* and *kd-backtrack* algorithms, which are based on shortening the ray from the start when a stack pop would be necessary. By advancing the starting point of the ray to the leaf exit point, the node will be skipped in subsequent traversal steps. Kd-restart continues the search by simply restarting the traversal from the root node using the shortened ray. To avoid restarting after every leaf intersection, the kd-backtrack algorithm adds parent pointers and bounding boxes to the tree. These are used for ascending in the tree after processing a leaf.

Horn et al. [HSHH07] proposed two improved algorithms based on kd-restart: *kd-push-down* and *short-stack* traversal. Kd-push-down identifies the deepest node that fully contains the valid intersection interval and then uses that node, instead of the root, as the starting point for the traversal restarts. Short-stack traversal reduces the amount of necessary restarts by maintaining a small, fixed-size stack. The traversal must be restarted only when the short-stack underflows.

2.2. Stackless BVH traversal

Most of the stackless kd-tree traversal techniques, with the notable exception of short-stack traversal, cannot be directly applied to BVHs because the nodes of a BVH may overlap [Lai10].

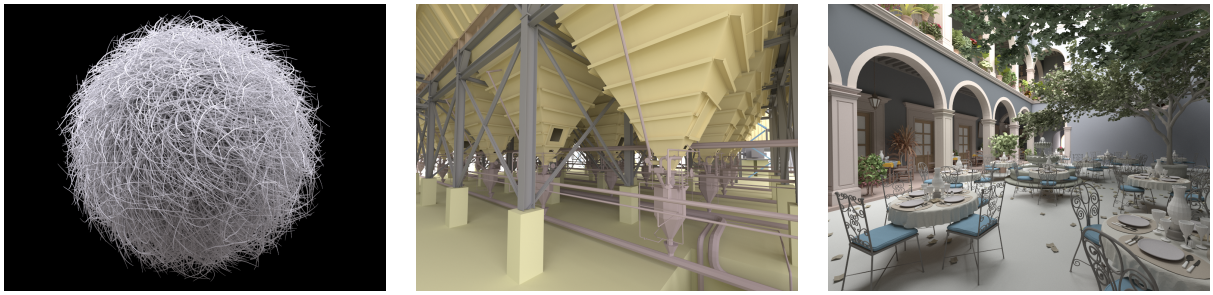
Smits [Smi98] suggested the storage of a *skip pointer* in each BVH node, which points to the next node to process if the current node is missed by the ray. The downside of this approach is that it can traverse the BVH only in a predefined order, without taking into account ray directions or node distances, which incurs a major performance penalty. Torres et al. [TMG09] developed a GPU-optimized version for coherent rays using ray packets.

The *restart trail* method by Laine [Lai10] enables traversal restarts for BVHs (or other kinds of binary trees) by encoding which part of the tree has been visited so far in a 32- or 64-bit *trail*. This value stores one bit of information per tree level. When a restart is triggered, the trail guides the downward traversal from the root to the next unprocessed node. The advantage of this approach is that it supports ordered traversal according to the node distances, but due to the restarts, it visits more than twice as many nodes as stack-based traversal. This overhead can be alleviated, at the expense of increasing the traversal state size, by adding a short-stack.

Hapala et al. [HDW*11] add a parent pointer to each node to backtrack in the tree instead of restarting from the root. Their method determines the next node to traverse using simple state logic, and it performs the same box and triangle intersection tests as an equivalent stack-based version. It needs to store only two bits of state in addition to the current node pointer, which is significantly less than the size of a trail or bitstack. However, it has to re-evaluate the traversal order heuristic for all revisited nodes, which practically restricts the heuristic to a simple ray direction based technique. Using the intersection distances to determine the near and far children of a node would be too expensive because both would have to be re-intersected. Such distance-based sorting, though, would not necessarily always lead to fewer traversal steps [Dam11].

Very recently, Barringer and Akenine-Möller [BAM13] introduced a stackless algorithm for binary trees that efficiently supports dynamic traversal order, based on the child distances, without restarting. They described three variants of the algorithm: two for implicit trees, and one for sparse trees with parent pointers. The traversal state is maintained using two integers: the current node's index (or address) and *left-first descent level index*. The *level index* is the relative index of a node of an implicit tree with regard to the first node on the same level. However, this tree is not the actual tree stored in memory but a *virtual* implicit version of it with nodes sorted according to the dynamic traversal order. Traversing this virtual tree in left-first order is equivalent to traversing the original one in dynamic order. Thus, the

(a) CONFERENCE (282K triangles) (b) CRYTEK SPONZA (262K triangles) (c) FAIRY (174K triangles)



(d) HAIRBALL (2.9M triangles) (e) POWER PLANT (12.7M triangles) (f) SAN MIGUEL (10.5M triangles)

Figure 1: *Test scenes used for the performance measurements of the ray traversal algorithms. The images were rendered using simple 8-bounce diffuse path tracing.*

left-first level index of the current node can be used to efficiently backtrack in the original tree, without re-intersecting the nodes.

3. Algorithm Overview

Our goal is to traverse the same sequence of nodes as a stack-based algorithm with a distance-based order heuristic, but using only a few state variables. Also, we want to avoid intersecting the same nodes multiple times.

A standard stack-based approach for N-way trees performs the following operations for each visited inner node: First, it intersects all N child bounding boxes, computing the intersection distances. It then selects the nearest child as the next node to traverse, and pushes the other (up to $N-1$) children to the stack. If all children were missed by the ray, a node is popped from the stack, and the traversal continues with that node.

For each visited leaf node, the primitives stored in the respective node are intersected with the ray, and then the stack is popped to get the next node.

Our algorithm replaces the stack pop with backtracking in the tree from the current node. The purpose of this operation is to find the next unprocessed node. This is a node whose bounding box was hit by the ray while processing the parent, but which has not been traversed yet. It is a sibling of either the current node or one of its ancestors. To be able to ascend in the tree, we add a parent pointer to each node. We also store pointers to the siblings for accessing them without taking a round trip to the parent. These additional links do not necessarily increase the node size as the original layout is often padded with unused values.

The backtracking is guided by a bitmask that encodes which part of the tree needs to be traversed. It stores $N-1$ bits for each visited tree level (except the root level), and is updated similarly to a stack, using bitwise push and pop operations. Hence, we call this special bitmask a *bitstack*. The per-level values in the bitstack are *skip codes*. These indicate which siblings of the most recently visited node on the respective level have already been processed and thus must be skipped.

In the following section, we describe in detail a simple version of our traversal algorithm for binary trees, which we later extend to support 4-way trees (Section 5).

4. MBVH2 Traversal

The binary variant of our stackless algorithm is shown in Algorithm 1. We use two state variables: a pointer to the current node (*node*) and the bitstack, a 32- or 64-bit integer (*bitstack*). For binary trees, the skip codes pushed onto the bitstack are 1-bit flags, which have the following semantics:

Algorithm 1 Stackless MBVH2 traversal

```
SHOOTRAY(ray)
 1  node ← ROOT
 2  bitstack ← 0
 3  loop
 4    if ISINNER(node) then
 5      intersect ray with children
 6      if any child was hit then
 7        bitstack ← bitstack ≪ 1
 8        if one child was hit then
 9          node ← the child that was hit
10        else
11          node ← the nearest child
12          bitstack ← bitstack ∨ 1
13        end if
14        continue
15      end if
16    else
17      // Leaf
18      intersect ray with primitives
19      shorten ray if closer intersection found
20    end if
21    // Backtrack
22    while bitstack ∧ 1 = 0 do
23      if bitstack = 0 then
24        return  // Terminate traversal
25      end if
26      node ← PARENT(node)
27      bitstack ← bitstack ≫ 1
28    end while
29    node ← SIBLING(node)
30    bitstack ← bitstack ⊕ 1
31  end loop
```

- **0**: Skip the sibling of the current node; go to the parent.
- **1**: Traverse the sibling of the current node.

The top of the bitstack is implicitly the least significant bit. This means that when pushing or popping an item, all the items in the stack must be shifted by one position, but this can be efficiently implemented using a simple bitwise shift. The initial value of the bitstack is 0, which is equivalent to an empty stack because it indicates that there are no nodes to process (i.e., all skip codes are 0). The advantage of this representation is that the traversal can be terminated earlier than returning to the root node, avoiding unnecessary backtracking steps.

The main traversal loop begins at line 4 with checking whether the current node is an inner node or a leaf node. If it is an inner node, its two child bounding boxes are tested for intersection with the ray (line 5).

If any of the children were hit, we first push a 0 bit to the bitstack by shifting the bits to left (line 7). For a single hit, we only have to set the current node to the intersected child. The skip code of 0 that was just pushed ensures that the other child subtree, which was missed by the ray, will not be later traversed.

Lines 11–12 handle the less frequent case of two hits. We compare the intersection distances and set the current node to the *near* child. Furthermore, we change the skip code to 1 with a binary OR operation in order to enable the traversal of the *far* child (line 12).

After processing the node, we continue the downward traversal (line 14). If no children were hit, backtracking is triggered to find the next node to process.

When we visit a leaf node (lines 18–19), we intersect the primitives in the leaf and shorten the ray if we find an intersection closer than what we have previously recorded. Then, we start backtracking in the tree.

The backtracking is performed in lines 22–30. In a loop, we ascend in the tree until we find a non-zero skip code in the bitstack, if there is any. The top stack item is extracted using a binary AND (line 22). If the bitstack is equal to 0, the entire traversal is terminated (lines 23–25). Otherwise, we jump to the parent node, pop the bitstack, and continue the search. After exiting the loop, we jump to the sibling of the current node, which has not yet been traversed. Finally, we flip the skip code at the top of the bitstack to 0 with a XOR, in order to avoid revisiting the previous, already processed node (line 30).

4.1. Comparison

A similar approach for sparse binary trees has been recently proposed by Barringer and Akenine-Möller [BAM13], but there are some important differences. Although the left-first descent level index in their algorithm is functionally similar to our bitstack, the semantics of the bits in these values are different. Our approach is slightly more efficient for two reasons:

- Barringer terminates the traversal only when it returns to the root node. In contrast, we exit the loop earlier if the bitstack becomes zero, the testing of which has a low cost.
- In many cases, the bitstack or level index must be 64 bits wide. On architectures with only limited native support for 64-bit integers (e.g., current GPUs), most full-width operations have a performance penalty. In Barringer's algorithm, one such operation is the incrementation of the level index before starting to backtrack. Our approach requires only a simple bit flip (at the end), which can be executed at full speed.

For the test scenes in Figure 1, our algorithm is faster by 1–11% on the NVIDIA Kepler GK110 architecture (see Table 3). Another advantage is that it scales naturally to higher branching factors, as described in the next section.

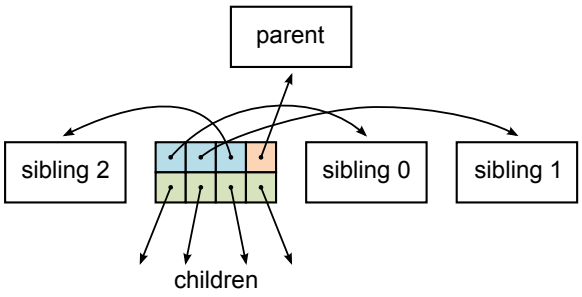


Figure 2: *The stackless MBVH4 traversal algorithm requires 8 pointers in each node: 4 child pointers, 3 sibling pointers, and a parent pointer. The siblings are referenced in circular order, starting from the first sibling after the node in question.*

5. MBVH4 Traversal

For binary trees, 1-bit skip codes are sufficient because each node has only one sibling. In order to traverse 4-way trees, we extend the skip codes to 3 bits, where each bit corresponds to a sibling of the respective node (see Figure 3a for an example). These bits have the same semantics as 1-bit skip codes: a 0 bit means that the sibling must be skipped, whereas a 1 bit means that it must be traversed. The siblings of a particular node are indexed *circularly* starting from the next node in the sibling group, as shown in Figure 2. For example, if the index of a node is 1, its siblings in ascending order are nodes 2, 3, and 0. Nodes that have less than 4 children are padded with invalid or empty node references, thus *every* node has exactly 3 siblings.

The limitation of the skip codes is that they do not encode the order in which the siblings should be processed. Like for binary trees, we always descend into the nearest node first, but we cannot traverse its siblings in front-to-back order. This, however, has a quite small impact on performance because in about 90% of the traversal steps only two or less children are hit by the ray [BWW*12]. For our scenes, the performance hit caused by disabling full sorting in a reference stack-based approach [Ern11] is 2–9%.

The full traversal algorithm is given in Algorithm 2. For a single intersected child (line 9), the skip code is 000. When more than one child is hit, we first determine the index of the nearest child (*nearPos*), which we select as the next node (lines 12–13). Then, we compute the skip code for this node using the SKIPCODE function (line 14). The implementation of this function can be seen on line 36. The skip code is computed from the *hit mask*, a 4-bit mask indicating which children are hit (*mask*), and the index of the selected child (*pos*). The bit for the selected node is removed from the hit mask, and the remaining 3 bits are rotated so that their positions match the indices of the corresponding siblings. For example, on Figure 3a the hit mask is 0111, and the index of the nearest child is 1, thus, the resulting skip code is 101.

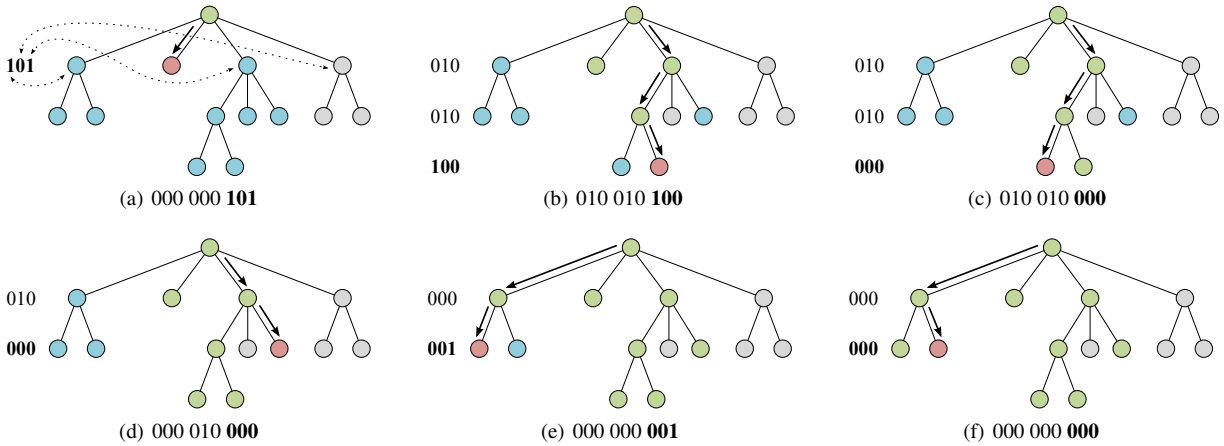
When backtracking is triggered, we first look for a node that has a non-zero skip code by following the parent pointers (lines 25–31). If this was successful, we have to jump to the next unprocessed sibling of this node. We determine the index of this sibling by scanning the skip code for the first set bit using the BITSCAN function (line 32). On most processors, BITSCAN can be implemented with a single in-

Algorithm 2 Stackless MBVH4 traversal

```
SHOOTRAY(ray)
 1  node ← ROOT
 2  bitstack ← 0
 3  loop
 4    if ISINNER(node) then
 5      intersect ray with children
 6      hitMask ← bitmask of child hits
 7      if any child was hit then
 8        bitstack ← bitstack ≪ 3
 9        if one child was hit then
10          node ← the child that was hit
11        else
12          nearPos ← index of the nearest child
13          node ← CHILD(node, nearPos)
14          skipCode ← SKIPCODE(hitMask, nearPos)
15          bitstack ← bitstack ∨ skipCode
16        end if
17        continue
18      end if
19    else
20      // Leaf
21      intersect ray with primitives
22      shorten ray if closer intersection found
23    end if
24    // Backtrack
25    while (skipCode ← bitstack ∧ 7) = 0 do
26      if bitstack = 0 then
27        return // Terminate traversal
28      end if
29      node ← PARENT(node)
30      bitstack ← bitstack ≫ 3
31    end while
32    siblingPos ← BITSCAN(skipCode)
33    node ← SIBLING(node, siblingPos)
34    bitstack ← bitstack ⊕ SKIPCODENEXT(skipCode)
35  end loop

SKIPCODE(mask, pos)
36  return ((mask ≫ (pos + 1)) ∨ (mask ≪ (3 − pos))) ∧ 7

SKIPCODENEXT(code)
37  newCode ← code ≫ (BITSCAN(code) + 1)
38  return newCode ⊕ code
```



(a) 000 000 **101** (b) 010 010 **100** (c) 010 010 **000**

(d) 000 010 **000** (e) 000 000 **001** (f) 000 000 **000**

Figure 3: *Example for stackless MBVH4 traversal. Only the steps where backtracking is triggered are depicted. Blue-colored nodes represent unprocessed nodes, green nodes have been already processed, gray nodes have been culled (i.e., the ray does not intersect them), and the red node is the current node. The invalid/empty nodes used for padding are not shown. The bold arrows indicate the path from the root to the current node. The value of the bitstack can be seen below the tree in binary form. The skip codes in the bitstack are also shown on the corresponding tree levels. The top of the bitstack is highlighted in bold. The dotted arrows in (a) connect the bits in the skip code with the nodes which they refer to.*

struction. Finally, we have to update the skip code at the top of the bitstack. This is done in line 34 by XORing the bitstack with a mask generated from the current skip code with the SKIPCODENEXT function. In this function (lines 37–38), we compute the next skip code by shifting out the trailing 0 bits and the next 1 bit. Then, we XOR this new skip code with the old one to produce the mask.

An example traversal with this algorithm is illustrated in Figure 3.

6. Implementation

In this section, we provide implementation details for three different processor architectures: CPU (Intel Ivy Bridge), MIC (Intel Knights Corner), and GPU (NVIDIA Kepler).

6.1. CPU

Our CPU implementation is based on the MBVH4 traversal method introduced in the Intel Embree ray tracer [Ern11]. The simplified source code for the stackless traversal kernel is listed in Appendix A.

The child bounding boxes in the nodes are stored in structure-of-arrays (SoA) format to facilitate SIMD processing. The size of the node data structure in the original stack-based method is 112 bytes without any padding or 128 bytes with cache line padding. To support stackless traversal, we need to extend this layout with both parent and sibling pointers. These values can be fit inside the padding, thus, the total node size is 128 bytes. We have to add the pointers to the leaf nodes as well, which, depending on the chosen triangle representation, may or may not increase their size.

Computing the skip codes during traversal is relatively costly. We solve this problem by employing lookup tables for both the SKIPCODE and SKIPCODENEXT functions, which are small enough to easily fit into the L1 cache. The SKIPCODE table is addressed with a 6-bit value composed of the hit mask and the node index. Using one byte per entry, the size of the table is 64 bytes (i.e., a single cache line). The SKIPCODENEXT table can be even smaller as it contains only 8 entries.

We use a 128-bit bitstack to be able to handle complex scenes that require deep trees. This way, the maximum permitted tree depth is 42. Since current CPUs do not have native 128-bit integer support, we implement the bitstack operations with 64-bit instructions. The most demanding operations are the bit shifts, which we implement using the special double precision shift instructions SHLD and SHRD. This approach is faster than using only regular shifts.

6.2. MIC

The Xeon Phi implementation is built upon the MBVH4 single-ray traversal algorithm by Benthin et al. [BWW*12], which has many similarities to the CPU algorithm (see Appendix B for the source code). Our stackless approach can be applied to hybrid single/packet traversal as well, but we opted for single traversal because of its simplicity and close-to-optimal performance for highly incoherent rays.

The node bounding boxes are packed in an array-of-

Scene	Method	MBVH4-CPU Intel Core i7-3770			MBVH4-MIC Intel Xeon Phi SE10P			MBVH2-GPU NVIDIA Tesla K20c		
		Mray/s	N_B	N_T	Mray/s	N_B	N_T	Mray/s	N_B	N_T
CONFERENCE	stack	23.4	10.4	2.5	118.3	10.8	2.1	142.7	24.6	4.3
	stackless	21.3	11.2	2.7	103.1	11.6	2.3	101.0	24.0	4.0
	Δ	*−9%*	*+8%*	*+11%*	*−13%*	*+7%*	*+10%*	*−29%*	*−3%*	*−7%*
CRYTEK SPONZA	stack	14.9	16.6	3.0	70.3	18.6	2.5	93.5	39.5	5.6
	stackless	12.4	19.6	3.9	60.3	20.4	2.7	64.2	37.6	4.6
	Δ	*−17%*	*+18%*	*+31%*	*−14%*	*+10%*	*+9%*	*−31%*	*−5%*	*−17%*
FAIRY	stack	19.1	13.5	3.2	92.1	14.6	2.5	73.1	30.3	7.8
	stackless	17.2	14.7	3.5	78.0	15.6	2.8	58.2	29.9	7.6
	Δ	*−10%*	*+9%*	*+11%*	*−15%*	*+7%*	*+11%*	*−20%*	*−1%*	*−2%*
HAIRBALL	stack	7.6	25.6	5.4	37.3	27.6	4.7	24.1	58.8	15.3
	stackless	6.5	29.5	6.3	31.7	30.8	5.2	18.6	58.6	15.0
	Δ	*−15%*	*+15%*	*+16%*	*−15%*	*+12%*	*+11%*	*−23%*	*−0%*	*−2%*
POWER PLANT	stack	12.5	18.2	4.2	56.9	19.6	3.9	51.7	44.0	13.1
	stackless	10.4	21.9	4.9	47.6	22.8	4.3	40.8	41.8	12.1
	Δ	*−17%*	*+20%*	*+17%*	*−16%*	*+17%*	*+9%*	*−21%*	*−5%*	*−7%*
SAN MIGUEL	stack	7.8	25.0	4.6	38.5	26.6	4.1	33.3	56.9	9.8
	stackless	6.5	29.3	5.6	33.2	30.4	4.6	26.3	55.1	9.1
	Δ	*−17%*	*+17%*	*+20%*	*−14%*	*+14%*	*+11%*	*−21%*	*−3%*	*−6%*

Table 1: *Performance measurements for 8-bounce diffuse path tracing (no Russian roulette) with trivial shading (no colors or textures) on CPU, MIC, and GPU architectures. The scenes were rendered from the views depicted in Figure 1, and the image resolution was* 1024×768 *pixels. We have compared the state-of-the-art stack-based ray traversal methods [Ern11, BWW*12, ALK12] with our stackless methods in terms of: ray tracing speed (including shading) in million rays per second (Mray/s), number of multi-box intersections (N_B), and number of single- or multi-triangle intersections (N_T). Relative values (stackless/stack − 1) are also listed (Δ).*

structures (AoS) layout to better exploit the 16-wide SIMD units. The SIMD vectors are divided into 4-wide lanes, each containing a 3D vector. We insert the node pointers into the unused slots in the SIMD vectors, so the extended node data structure does not occupy more space than the basic one (128 bytes).

Because of the AoS layout, the ray-box intersection routine produces a sparse 16-bit hit mask. Every fourth bit is a hit flag, whereas all the other bits are zero. Building a SKIPCODE table for such a large mask or directly compacting the mask would not be practical. Therefore, we also produce a 4-bit hit mask by permuting (with VPERMD) the *near* and *far* distance vectors, and then comparing them. We compute the lookup table index from this compact mask and a 2-bit code that identifies the closest node. For two hits, this code is either 0 or 1 (for the first or second hit, respectively), and for more hits it is the node index. Thus, the table has 64 entries, just like the one used on the CPU.

6.3. GPU

We have optimized the GPU kernels for the NVIDIA Kepler GK110 architecture [Nvi12]. Our baseline traversal method is the stack-based speculative *while-while* kernel by Aila et al. [AL09, ALK12], which traverses binary BVHs.

We implement the stackless MBVH2 traversal algorithm using the following *while-if-while* loop organization:

```
while true
  while node is inner
    go to nearest child or break
  if node is leaf
    intersect primitives
  while skip code is zero
    go to parent or return
  go to sibling
```

We do not postpone leaf intersections because it would be inefficient in combination with backtracking and would also increase the size of the traversal state. This way, the state consists of only a node pointer and a 64-bit bitstack.

Our node data structure has the same size as the original (64 bytes) because there is enough free space for the parent and sibling pointers. When intersecting a node, we fetch the node data (including triangles) through the texture cache, but for backtracking we use regular memory loads.

The source code for the kernel can be seen in Appendix C.

7. Results

We evaluated the performance of our stackless ray traversal algorithms and the corresponding stack-based algorithms us-

Tree	Max depth	Method	State size (B)
MBVH2	64	stack	264
		stack+dist	520
		stackless	12
MBVH4	42	stack	512
		stack+dist	1016
		stackless	20

Table 2: *Traversal state sizes (in bytes) for different types of BVHs and traversal methods. The listed methods are: simple stack-based, stack-based with node distances (stack+dist), and stackless traversal. All sizes include a 32-bit pointer to the current node.*

Scene	[BAM13] Mray/s	Our Mray/s	Δ
CONFERENCE	91.2	101.0	*+11%*
CRYTEK SPONZA	60.8	64.2	*+6%*
FAIRY	53.5	58.2	*+9%*
HAIRBALL	18.0	18.6	*+3%*
POWER PLANT	39.2	40.8	*+4%*
SAN MIGUEL	26.0	26.3	*+1%*

Table 3: *Performance of stackless MBVH2-GPU traversal using [BAM13] versus our algorithm for 8-bounce diffuse path tracing (also see Table 1). The GPU used was an NVIDIA Tesla K20c.*

ing a simple but highly optimized diffuse path tracer on all three processor architectures.

The different types of BVHs were constructed using the same high-quality primitive partitioning techniques: we used both object and spatial partitioning optimized with the surface area heuristic (SAH), as proposed by Stich et al. [SFD09]. The MBVH4 nodes were generated using the top-down greedy splitting method by Wald et al. [WBB08]. The MIC tree leaves were limited to 4 triangles (i.e., a single multi-triangle), whereas the CPU and GPU ones were limited to 8 triangles. All traversal algorithms used variants of the Möller-Trumbore ray-triangle intersection test [MT97].

The CPU used for the CPU benchmarks was an Intel Core i7-3770 (Ivy Bridge, 4 cores, 8 threads, 3.4 GHz, 8 MB L3 cache) with 16 GB RAM (DDR3-1600, dual channel). The code was compiled for 64-bits and AVX. Both the CPU and MIC implementations were written in C++ with SIMD intrinsic functions and OpenMP, and they were compiled with Intel C++ Compiler XE 13.1.

The MIC card was an Intel Xeon Phi SE10P coprocessor (Knights Corner, B1-stepping, 61 cores, 244 threads, 1.1 GHz, 8 GB GDDR5, ECC on) installed in a compute node of the Stampede supercomputer at the Texas Advanced Computing Center (TACC).

The GPU was an NVIDIA Tesla K20c (Kepler GK110, 13 multiprocessors, 2496 CUDA cores, 0.7 GHz, 5 GB GDDR5, ECC on). The path tracer was implemented as a traditional *megakernel* in CUDA 5.0.

The performance results, including the ray tracing speeds and the number of box and triangle intersection tests, are shown in Table 1. Our stackless algorithms, similarly to previous methods, are somewhat slower than the reference stack-based ones when used for ordinary ray tracing; however, they maintain about 22–51× smaller traversal states (see Table 2). For special rendering methods that trace a very large amount of rays in parallel, low memory footprint is essential and could lead to a much higher overall performance.

For our test scenes, stackless traversal is slower by 9–17% on the CPU, 13–16% on the MIC, and 20–31% on the GPU. This is caused by the more complex traversal logic, more irregular memory accesses, and on the CPU and MIC the slightly higher number of box and triangle intersections. One reason for the latter is that the stack-based versions store node distances in the stack, and thus can skip previously pushed nodes that no longer need to be traversed.

Also, on the CPU we always sort the pushed nodes by hit distance, which further decreases the number of intersections by a small amount. Without these two minor optimizations, the stack-based algorithms would visit the same (in case of MBVH2) or very similar sequence of nodes as the respective stackless ones.

On the GPU, our stackless algorithm has slightly *fewer* intersections than the stack-based version because it does not postpone leaf nodes; however, this results in lower SIMD efficiency. It outperforms the sparse traversal algorithm by Barringer and Akenine-Möller [BAM13] for all test cases, as can be seen in Table 3.

8. Conclusions and Future Work

We have presented a novel and efficient stackless ray traversal algorithm for the MBVH acceleration structure. Two algorithm variations have been discussed: one for 4-way trees and one for binary trees. To our knowledge, this is the first published stackless method for wide BVHs.

The results show that on current architectures our algorithm performs competitively to stack-based approaches, but it is not the fastest option for conventional ray tracers. However, the main advantage of our algorithm is that it has a much smaller traversal state. Because of this, it can significantly enhance the efficiency of advanced, massively parallel in-core or out-of-core ray tracing schemes.

As future work, we would like to analyze the proposed algorithm in the context of out-of-core ray tracing, where the traversal of many rays must be suspended and later resumed.

We also plan to investigate both stack-based and stackless MBVH4 traversal on latest-generation GPU architectures.

Acknowledgments

This work was possible with the financial support of the Sectoral Operational Programme for Human Resources Development 2007-2013, co-financed by the European Social Fund, under the project number POSDRU/107/1.5/S/76841 with the title *Modern Doctoral Studies: Internationalization and Interdisciplinarity*. The research was also supported by OTKA K-104476 (Hungary). We would like to thank Paul Navrátil and the Texas Advanced Computing Center at The University of Texas at Austin for kindly providing us access to the Stampede supercomputer. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K20c GPU used for this research.

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

References

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, 2010), HPG '10, Eurographics Association, pp. 113–122. 1

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 145–149. 1, 2, 7

[ALK12] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012. 1, 2, 7

[BAM13] BARRINGER R., AKENINE-MÖLLER T.: Dynamic stackless binary tree traversal. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (March 2013), 38–49. 2, 4, 8

[BWW*12] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W.: Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (September 2012), 1438–1448. 5, 6, 7

[Dam11] DAMMERTZ H.: *Acceleration Methods for Ray Tracing based Global Illumination*. PhD thesis, Ulm University, 2011. 2

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2008), 1225–1233. 1

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008* (2008), pp. 35–40. 1

[Ern11] ERNST M.: Embree: Photo-realistic ray tracing kernels. *ACM SIGGRAPH 2011 Exhibitor Tech Talk* (2011). 1, 5, 6, 7

[FS05] FOLEY T., SUGERMAN J.: KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 15–22. 2

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 59–64. 1

[HBŽ98] HAVRAN V., BITTNER J., ŽÁRA J.: Ray tracing with rope trees. In *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)* (Budmerice, Slovak Republic, April 1998), pp. 130–139. 2

[HDW*11] HAPALA M., DAVIDOVIČ T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient stack-less BVH traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics* (New York, NY, USA, 2011), SCCG '11, ACM, pp. 7–12. 2

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 167–174. 2

[Int13] INTEL: *Intel Xeon Phi System Software Developer's Guide*, June 2013. 2

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 89–99. 1

[KIS*12] KOPTA D., IZE T., SPJUT J., BRUNVAND E., DAVIS A., KENSLER A.: Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 197–204. 1

[KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 121–128. 1

[Lai10] LAINE S.: Restart trail for stackless BVH traversal. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, 2010), HPG '10, Eurographics Association, pp. 107–111. 2

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (May 1990), 153–166. 2

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28. 8

[NFLM07] NAVRÁTIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), RT '07, IEEE Computer Society, pp. 95–104. 1

[Nvi12] NVIDIA: *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Whitepaper, NVIDIA Corporation, 2012. 7

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424. 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA,

1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 101–108. 1

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (August 2008), 18:1–18:15. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 7–13. 1, 8

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (February 1998), 1–14. 2

[TMG09] TORRES R., MARTÍN P. J., GAVILANES A.: Ray casting using a roped BVH with CUDA. In *Proceedings of the 25th Spring Conference on Computer Graphics* (New York, NY, USA, 2009), SCCG '09, ACM, pp. 95–102. 2

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets – efficient SIMD single-ray traversal using multi-branching BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008* (2008), pp. 49–57. 1, 8

Appendix A: CPU Traversal Code (C++)

```
// Utility classes
// vfloat -> 4-wide SIMD vector of floats
// vmask  -> 4-wide SIMD vector of bools

// Utility functions
// select(m, a, b) -> r[i] = m[i] ? a[i] : b[i]
// shl128, shr128  -> optimized 128-bit shifts

// MBVH4 traversal loop
static const uint8_t  SKIPCODE[64]     = {...};
static const uint64_t SKIPCODE_NEXT[8] = {...};

int nodeId = 0;
uint64_t bitstack  = 0; // low
uint64_t bitstackH = 0; // high

for (; ;) {
  // Inner node loop
  while (isInner(nodeId)) {
    const Node& node = getNode(nodeId);

    // Box intersection
    vfloat near, far;
    intersectBoxes(ray, node, near, far);
    vmask hitVecMask = (near <= far);
    size_t hitMask = toIntMask(hitVecMask);
    if (UNLIKELY(hitMask == 0)) goto backtrack;

    // 1 hit
    size_t hitPos = bitScan(hitMask);
    int hitCount = bitCount(hitMask);
    shl128(bitstack, bitstackH, 3);
    if (LIKELY(hitCount == 1)) {
      nodeId = getChildId(node, hitPos);
      continue;
    }

    // 2 hits
    if (LIKELY(hitCount == 2)) {
      size_t hitPos2 = bitScan(
        bitTestAndComplement(hitMask, hitPos));
      if (near[hitPos] <= near[hitPos2]) {
        bitstack |= SKIPCODE[hitMask*4 + hitPos];
        nodeId = getChildId(node, hitPos);
      } else {
        bitstack |= SKIPCODE[hitMask*4 + hitPos2];
        nodeId = getChildId(node, hitPos2);
      }
      continue;
    }

    // 3-4 hits
    near = select(hitVecMask, near, INF);
    hitPos = indexOfMin(near);
    bitstack |= SKIPCODE[hitMask*4 + hitPos];
    nodeId = getChildId(node, hitPos);
  }

  // Leaf node
  ...

  // Backtrack
backtrack:
  size_t skipCode;
  while ((skipCode = (bitstack & 7)) == 0) {
    if (UNLIKELY((bitstack | bitstackH) == 0))
      return;
    nodeId = getParentId(getNode(nodeId));
    shr128(bitstack, bitstackH, 3);
  }
  size_t siblingPos = bitScan(skipCode);
  nodeId = getSiblingId(
    getNode(nodeId), siblingPos);
  bitstack ^= SKIPCODE_NEXT[skipCode];
}
```

Appendix B: MIC Traversal Code (C++)

```
// Utility classes
// vfloat -> 16-wide SIMD vector of floats
// vint   -> 16-wide SIMD vector of ints
// vmask  -> 16-bit vector mask

// Utility functions
// select(m, a, b) -> r[i] = m[i] ? a[i] : b[i]
// cmpOp(m, a, b)  -> r[i] = m[i] & (a[i] Op b[i])
// permute(a, p)   -> r[i] = a[p[i]]
// reduceOpPerLane(a)   -> reduce Op and broadcast
//                         within 4-wide lanes
// reduceOpCrossLane(a) -> reduce Op and broadcast
//                         across 4-wide lanes

vfloat origin, invDir, tMin, tMax; // ray

// MBVH4 traversal loop
static const uint8_t  SKIPCODE[64]     = {...};
static const uint64_t SKIPCODE_NEXT[8] = {...};

const vint HIT_PERM(0, 4, 8, 12); // 4x broadcasted

int nodeId = 0;
uint64_t bitstack  = 0; // low
uint64_t bitstackH = 0; // high

for (; ;) {
  // Inner node loop
  while (isInner(nodeId)) {
    const Node& node = getNode(nodeId);

    // Box intersection
    vfloat t0 = (node.lower - origin) * invDir;
    vfloat t1 = (node.upper - origin) * invDir;
    uint64_t bitstackShr61 = bitstack >> 61;
    vfloat nXyz = select(0x7777, min(t0,t1), tMin);
    vfloat fXyz = select(0x7777, max(t0,t1), tMax);
    vfloat near = reduceMaxPerLane(nXyz);
```

```
    vfloat far  = reduceMinPerLane(fXyz);
    vmask hitVecMask = cmpLe(0x8888, near, far);
    vfloat nearHit = select(hitVecMask, near, INF);
    if (UNLIKELY(none(hitVecMask))) goto backtrack;

    // 1 hit
    size_t hitMask = toIntMask(hitVecMask);
    size_t hitPos = bitScan(hitMask); // i*4+3
    int hitCount = bitCount(hitMask);
    nodeId = getChildId(node, hitPos);
    prefetchNodeL1(nodeId);
    bitstack <<= 3;
    bitstackH = (bitstackH << 3) | bitstackShr61;
    if (LIKELY(hitCount == 1)) continue;

    // 2 hits
    vfloat nearPak = permute(near, HIT_PERM);
    vfloat farPak  = permute(far,  HIT_PERM);
    size_t hitMaskPak = toIntMask(
      cmpLe(0xf, nearPak, farPak));
    size_t hitPos2 = bitScan(hitMask, hitPos);
    int nodeId2 = getChildId(node, hitPos2);
    int hitDist = asInt(near)[hitPos];
    if (LIKELY(hitCount == 2)) {
      int hitDist2 = asInt(near)[hitPos2];
      prefetchNodeL1(nodeId2);
      if (hitDist <= hitDist2) {
        bitstack |= SKIPCODE[hitMaskPak*4];
      } else {
        bitstack |= SKIPCODE[hitMaskPak*4 + 1];
        nodeId = nodeId2;
      }
      continue;
    }

    // 3-4 hits
    prefetchChildrenL2(node);
    vmask minHitVecMask = cmpEq(hitVecMask,
      nearHit, reduceMinCrossLane(nearHit));
    hitPos = bitScan(toIntMask(minHitVecMask));
    bitstack |=
      SKIPCODE[hitMaskPak*4 + (hitPos >> 2)];
    nodeId = getChildId(node, hitPos);
    prefetchNodeL1(nodeId);
  }

  // Leaf node
  ...

  // Backtrack
backtrack:
  size_t skipCode;
  while ((skipCode = (bitstack & 7)) == 0) {
    if (UNLIKELY((bitstack | bitstackH) == 0))
      return;
    nodeId = getParentId(getNode(nodeId));
    bitstack = (bitstack >> 3) | (bitstackH << 61);
    bitstackH >>= 3;
  }
  size_t siblingPos = bitScan(skipCode);
  nodeId = getSiblingId(
    getNode(nodeId), siblingPos);
  bitstack ^= SKIPCODE_NEXT[skipCode];
}
```

Appendix C: GPU Traversal Code (CUDA)

```
// MBVH2 traversal loop
int nodeId = 0;
uint64_t bitstack = 0;
int parentId, siblingId; // cached node links

for (; ;) {
  // Inner node loop
  while (isInner(nodeId)) {
    // Load node through texture cache
    // node links  -> nid
    // node bounds -> n0xy, n1xy, nz
    int4   nid  = __ldg((  int4*)bvh + nodeId);
    float4 n0xy = __ldg((float4*)bvh + nodeId + 1);
    float4 n1xy = __ldg((float4*)bvh + nodeId + 2);
    float4 nz   = __ldg((float4*)bvh + nodeId + 3);
    parentId  = nid.x;
    siblingId = nid.y;

    // Box intersection
    float near0, far0, near1, far1;
    intersectBoxes(ray, n0xy, n1xy, nz,
      near0, far0, near1, far1);
    bool hit0 = (far0 >= near0);
    bool hit1 = (far1 >= near1);
    if (!hit0 && !hit1) break;

    bitstack <<= 1;
    if (hit0 && hit1) {
      nodeId = (near1 < near0) ? nid.w : nid.z;
      bitstack |= 1;
    } else {
      nodeId = hit0 ? nid.z : nid.w;
    }
  }

  // Leaf node
  if (!isInner(nodeId)) {
    ...
  }

  // Backtrack
  while ((bitstack & 1) == 0) {
    if (bitstack == 0) return;
    nodeId = parentId;
    int4 nid = *((int4*)bvh + nodeId);
    parentId  = nid.x;
    siblingId = nid.y;
    bitstack >>= 1;
  }
  nodeId = siblingId;
  bitstack ^= 1;
}
```