Wide BVH Traversal with a Short Stack

K. Vaidyanathan   S. Woop   C. Benthin

Intel Corporation

Abstract

Compressed wide bounding volume hierarchies can significantly improve the performance of incoherent ray traversal, through a smaller working set of inner nodes and therefore a higher cache hit rate. While inner nodes in the hierarchy can be compressed, the size of the working set for a full traversal stack remains a significant overhead. In this paper we introduce an algorithm for wide bounding volume hierarchy (BVH) traversal that uses a short stack of just a few entries. This stack can be fully stored in scarce on-chip memory, which is especially important for GPUs and dedicated ray tracing hardware implementations. Our approach in particular generalizes the restart trail algorithm for binary BVHs to BVHs of arbitrary widths. Applying our algorithm to wide BVHs, we demonstrate that the number of traversal steps with just five stack entries is close to that of a full traversal stack. We also propose an extension to efficiently cull leaf nodes when a closer intersection has been found, which reduces ray primitive intersections by up to 14%.

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

1. Introduction

The advent of GPUs with fixed function acceleration for BVH traversal [KMSB18] has enabled a significant increase in ray tracing performance, driving the recent adoption of real time ray tracing in games. However, as the computational overhead of ray traversal is reduced with fixed function implementations, ray throughput is eventually bound by other constraints, such as on-chip storage and memory bandwidth.

For this reason, compressed wide BVHs [WWB*14, YKL17] make a compelling choice for fixed function traversal [VKJT16], with compressed inner nodes resulting in improved cache hit rates and higher ray throughput. Moreover, the nodes in a wide BVH can be compressed individually, unlike incremental compression schemes for binary BVHs [Kee14, VAMS16]. Wide BVHs also result in shallower trees that can be traversed with lower latency.

Besides inner node fetches, the memory accesses for ray traversal also include reads for the leaf geometry as well as reads and writes to the traversal stack. In this paper we focus on the traversal stack, which has a significant storage and bandwidth cost [AK10, ÁS14, YKL17]. Our first contribution is a traversal algorithm for wide BVHs that performs efficiently with a short stack that can be fully maintained in dedicated on-chip storage.

Our algorithm is a generalization of the binary restart trail approach of Laine [Lai10] to wide BVHs. Similar to the binary restart trail, in the event of a short stack under-run we restart traversal from the root node, which introduces an overhead of additional traversal

steps. We evaluate our algorithm with different path traced scenes and demonstrate that restart overhead is at most 10% with a short stack of just five stack entries, while requiring significantly less memory than a full stack.

Our second contribution is a mechanism that leverages our generalized restart trail to cull leaf node entries on the stack for a closest-hit search. By culling stack entries, we skip expensive primitive intersection tests if a closer intersection has been previously found. Evaluating different scenes, we demonstrate that this technique can reduce the number of expensive ray primitive intersection tests by up to 14%.

2. Related Work

There are several techniques for BVH traversal that store additional information in the BVH node to traverse up the tree or to a sibling node, thereby avoiding a traversal stack. For example, Smits [Smi98] stores skip pointers that reference the next node to traverse if the current node is not intersected. However, this approach enforces a pre-defined traversal order which adversely impacts traversal performance. Hapala et al. [HDW*11] instead store a pointer to the parent node, which enables backtracking of traversal and selecting a dynamic traversal order based on the ray direction and the axis of partitioning at each BVH level.

Barringer et al. [BAM13] introduced stackless traversal algorithms for both implicit and sparse BVHs. For sparse BVHs which are more memory efficient for unbalanced trees, they use parent

pointers to backtrack, similar to Hapala et al. [HDW*11] but traverse child nodes in a front to back order. Afra et al. [ÁS14] extend the backtracking approach to wide BVHs using a bitmask to mark child nodes that can be skipped. They traverse the closest child node first but do not guarantee front to back ordering for the remaining children. While backtracking techniques eliminate the traversal stack they also introduce redundant node fetches and therefore additional latency and bandwidth.

Binder at. al. [BK16] avoid redundant node fetches using a perfect hashing scheme to backtrack in constant time. However, the hash lookups require additional memory accesses, which they reduce by storing references to the uncle and grand uncle in each BVH node.

Our traversal algorithm on the other hand, is closely related to the restart trail approach of Laine [Lai10], which we generalize to wide BVHs. Similar to Laine, we support a short stack of a few entries which can eliminate most of the redundant node fetches. Our approach does not require additional pointers in the BVH node or hash lookups for backtracking which leads to a simpler implementation and a more compact node layout.

Algorithm 1 BVH-N Traversal

1: $trail \leftarrow (0,0,0,...)$
2: $level \leftarrow 0$
3: $node \leftarrow root$
4: $shortstack \leftarrow empty$
5: **while** $exit \neq true$ **do**
6:     **if** $node$ is internal node **then**
7:         $k \leftarrow trail[level]$
8:         $H \leftarrow$ list of child nodes that intersect the ray
9:         $S \leftarrow$ sort H by increasing hit distance
10:        **if** $k = N$ **then**
11:           $Q \leftarrow$ remove all but last entry in S
12:        **else**
13:           $Q \leftarrow$ remove the first k entries in S
14:        **end if**
15:        **if** $|Q| = 0$ **then**
16:           $exit = $**Pop(trail,level)**
17:        **else**
18:           $node \leftarrow$ first entry in Q
19:           remove the first entry from Q
20:           **if** $|Q| = 0$ **then**
21:              $trail[level] \leftarrow N$
22:           **else**
23:              mark last entry in Q as the last child
24:              **PushBackToFront**(Q)
25:           **end if**
26:           $level \leftarrow level + 1$
27:        **end if**
28:     **else**
29:         **IntersectLeaf()**
30:         $exit = $**Pop(trail,level)**
31:     **end if**
32: **end while**

---

3. Algorithm

We generalize the binary restart trail of Laine [Lai10] to N-wide BVHs using an array of counters, one for each BVH level. Each counter $trail[i]$, ranges between 0 and $N-1$ and indicates the number of children at level i that have already been processed. These counters allow traversal to restart from the root node by skipping already processed subtrees. As a special case, if the currently processed subtree is the last one at the current level, the corresponding counter is set to N, indicating that there are no more entries on the stack for that particular level. By skipping these levels in the restart trail we can determine the level of the topmost stack entry at any given time, as described in Section 3.1.

Algorithm 1 describes our traversal technique for N-wide BVHs. The restart trail is initialized to zeros (no subtree gets skipped) and traversal starts from the root node at $level = 0$ (line 1). For every internal N-wide node, we compute intersections between the N child bounding boxes and the ray to find a list H of child nodes that the ray intersects (line 8). H is then sorted based on increasing hit distance to obtain a sorted list S.

If traversal restarts from the root node, one or more child nodes in S may have been previously traversed, which is indicated by the count value k at the current level. Therefore, we remove the first k nodes from S (line 13) resulting in a list Q of valid nodes for traversal. As described earlier, if the value of k equals N, then the current subtree is the last subtree to be traversed, in which case we remove all child nodes in S, except the last one (line 23).

If Q is empty, traversal continues with the next node popped from the traversal stack (line 15). Otherwise, traversal proceeds to the next level with the first entry in Q (closest remaining child) and the remaining entries of Q, if any, are pushed on the stack in back to front order (line 24). If the closest child is the last child at this level, no entries are pushed onto the stack, and we set the restart trail counter at the current level to N (line 21). We reserve one additional bit per stack entry to mark the last node in Q, which corresponds to the last child to be traversed at the current level (line 11).

While searching for the closest hit, the current hit is updated every time a ray intersects a leaf node (line 29). This culls subsequent child node intersections at a distance greater than the current hit distance, reducing the number of traversal steps. Therefore, when an internal node is re-visited after a restart, some of its children that previously intersected the ray may be culled. Traversing the child nodes in a front to back order not only leads to better performance, but also ensures that the nodes that get culled are at the end of the traversal order. This guarantees that the count value in the restart trail still refers to the same child or can be used to determine that the child got culled. Front to back ordering is not strictly required for an any-hit search, since traversal is terminated on the first leaf intersection.

3.1. Stack

Similar to the approach of Laine [Lai10], the restart trail can be accompanied by a short stack that retains the top few stack entries. When one or more entries get evicted from the bottom of the short stack, they are processed later by restarting traversal from the root node.
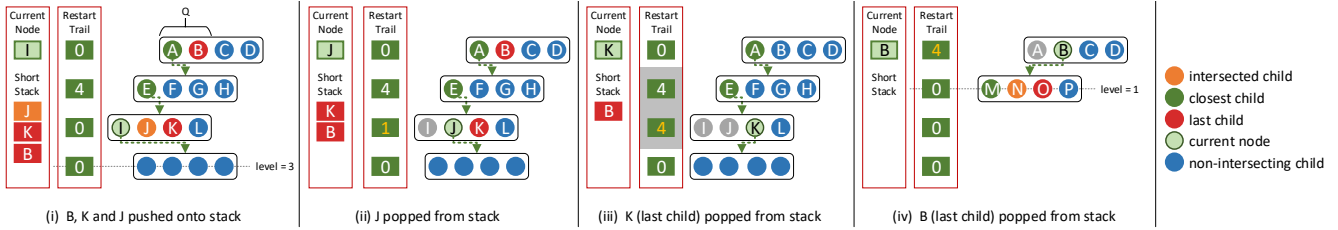
Figure 1: *Example BVH-4 traversal with a short stack, showing the sorted children Q and the restart trail count at each level. (i) The count at level 1 is set to 4, as there are no entries pushed onto the stack at this level. (ii) Processing node I yields no hit children. Therefore, node J is popped from the stack, incrementing the count at level 2. (iii) Processing node J yields no hit children. Therefore, node K (the last node corresponding to level 2), is popped from the stack and the counter is set to 3. (iv) Processing node K yields no hit children. The following pop operation skips levels 1 and 2 (gray) as they indicate the last child was already traverseds and node B (level 0) is popped from the stack.*

A pop operation on the stack is described in Algorithm 2. First, the BVH level (*parentLevel*) corresponding to the parent of the top-most stack entry is computed by scanning the restart trail upwards from the current level (Algorithm 3). During this scan, levels with a value *N* are skipped since they have no child nodes to process.

If all scanned levels have value *N*, the stack is empty and traversal is complete (line 4). Otherwise the counter at *parentLevel* is incremented to indicate that the next child node is being traversed (line 6) and all counters below *parentLevel* are cleared to 0.

If the short stack is not empty, traversal continues with its top-most entry. If this entry is marked as the last child, the counter at *parentLevel* is set to *N*, indicating that there are no more child nodes to be processed at this level (line 13). The stack is then popped and the current level is updated to *parentLevel* + 1. If the short stack is empty, traversal restarts from the root node, by setting the level to 0 and the node to the root node (line 9). Figure 1 shows an example traversal for a 4-wide BVH.

When we store the value *N* into the restart trail to indicate that we enter the last child, we no longer know the index of that last child. It is important that from now on the last hit child never changes for that parent node, otherwise we might later enter a different child (which has suddenly become last). However, a found hit may cause a child that was last to get culled, changing the last child. Under some assumptions this situation cannot happen: When we traverse down the BVH we may mark a child last, but we directly enter it. If geometry inside that child is properly bounded by the child's bounding box, no hit that culls that child can get found when traversing that child. Similarly, when we pop a child *C* from the short stack that is marked last, this child may no longer be last as a closer hit may have been found while it was pushed. This means that we mark that child last on the restart trail, and if we would now trigger a restart we would enter the wrong last child (as *C* is no longer last). However, a restart cannot get triggered as the child *C* is either a leaf node whose intersection will trigger a pop, or it is an internal node. If the bounds of the children of the internal node *C* are properly contained inside the bounds of *C* itself, then no child of *C* can get hit as *C* itself is not hit, thus again a pop is triggered.

3.2. Stack Culling

Nodes on the stack can be culled by additionally storing the node hit distance for each stack entry [WWB*14,FLP*17]. When a node is popped from the stack, if its hit distance is greater than the current hit distance (clipped ray length), it can be culled as it cannot contribute to a closer hit. Although this approach can improve performance by reducing the number of traversal steps and primitive intersections, it approximately doubles the required stack storage, which can be prohibitive for GPUs and custom hardware implementations where on-chip storage is limited.

We propose an alternative approach for stack culling that leverages our short stack traversal algorithm to achieve a reduction in expensive ray primitive intersections with a relatively small increase in traversal steps. If more than two child nodes intersect a ray and if any of these child nodes is a leaf, we push the parent node on the stack instead of the children and continue traversal with the closest child. Later, when the parent node is popped from the stack it is re-intersected, potentially culling several child nodes that have a hit distance greater than the current hit distance.

Our stack culling approach requires one additional bit per stack entry to mark parent nodes. When a node popped from the stack is marked as a parent, the current depth is set to the depth of the parent node instead of its children (line 17). Note that previously traversed children are marked by the restart trail and therefore will not get re-traversed.

4. Results

Our traversal algorithm is a good fit for hardware implementations where the short stack can be stored locally in registers and the stack operations can be efficiently implemented in hardware. We evaluate a functional model of such a hardware implementation and measure the number of traversal steps and ray primitive intersection tests which are key parameters that impact hardware cost.

We use a compressed 6-wide BVH for our analysis, since we can store information for up to six child nodes in 64 bytes, which is a typical cache line size on many compute architectures. For constructing the BVH we use the Embree builders [WWB*14] which we modify for generating 6-wide BVH nodes. We do not enable additional optimizations such as spatial splits. The BVH is

Figure 2: *Six example scenes used in our evaluation and their primitive counts (triangles/quads), rendered with three-bounce path tracing.*

	Full Stack (136 bytes)				Stack 5 (44 bytes)				Stack 5 + culling (44 bytes)	
	Closest Hit		**Any Hit**		**Closest Hit**		**Any Hit**		**Closest Hit**	
Scene	#trav	#quad	#trav	#quad	#trav	#quad	#trav	#quad	#trav	#quad
Bathroom	11.312	2.604	11.596	3.037	11.51(1.02×)	2.6 (1×)	11.65 (1×)	3.04 (1×)	11.59 (1.02×)	2.54 (0.97×)
Crown	18.373	5.323	14.429	4.248	20.3(1.1×)	5.31 (1×)	15.22 (1.05×)	4.25 (1×)	20.45 (1.11×)	5.2 (0.98×)
SanMiguel	30.788	8.291	7.443	2.207	33.7(1.09×)	7.96 (0.96×)	7.64 (1.03×)	2.21 (1×)	34.28 (1.11×)	7.73 (0.93×)
Villa	16.984	3.377	8.005	1.624	17.32(1.02×)	3.37 (1×)	8.03 (1×)	1.62 (1×)	17.55 (1.03×)	3.04 (0.9×)
Classroom	14.467	3.01	8.786	1.698	14.68(1.01×)	2.86 (0.95×)	8.81 (1×)	1.7 (1×)	15.43 (1.07×)	2.6 (0.86×)
BMW	8.044	3.07	4.718	2.258	8.13(1.01×)	2.97 (0.97×)	4.75 (1.01×)	2.26 (1×)	8.71 (1.08×)	2.91 (0.95×)

Table 1: *Comparison of the number of traversal steps (#trav) and quad intersections (#quad) per ray for path traced scenes traversed with a compressed 6-wide BVH. Compared to using a full stack, our short stack introduces a traversal step overhead of just 1-10% for* closest-hit *queries and 0-5% for* any-hit *queries, with approximately the same number of quad intersections. Applying our stack culling approach, the traversal overhead for* closest-hit *rays increases slightly but the reduction in quad intersections becomes more significant (2-14%). Our short stack also requires* ∼ 3× *less memory than a full stack.*

Algorithm 2 : Stack Pop

1: **procedure** POP(*trail*, *level*)
2:     *parentLevel* = **FindNextParentLevel(trail,level)**
3:     **if** *parentLevel* < 0 **then**
4:         **return** true
5:     **end if**
6:     *trail*[*parentLevel*] ← *trail*[*parentLevel*] + 1
7:     reset trail to 0 for levels *parentLevel* + 1 to *MaxLevel*
8:     **if** short stack is empty **then**
9:         *node* ← *root*
10:         *level* ← 0
11:     **else**
12:         *node* = **PopShortStack()**
13:         **if** node is tagged as last child **then**
14:             *trail*[*parentLevel*] ← N
15:         **end if**
16:         **if** node is tagged as parent node **then**
17:             *level* ← *parentLevel*
18:         **else**
19:             *level* ← *parentLevel* + 1
20:         **end if**
21:     **end if**
22:     **return** false
23: **end procedure**

Algorithm 3 : Finding the parent level for the topmost stack entry

1: **procedure** FINDNEXTPARENTLEVEL(*trail*, *level*)
2:     **for** $i ← (level − 1)...0$ **do**
3:         **if** *trail*[*i*] ≠ N **then**
4:             **return** *i*
5:         **end if**
6:     **end for**
7:     **return** -1
8: **end procedure**

---

that share an edge into a quad (triangle-pair). For typical scenes, most triangles get paired into quads (see Figure 2). This reduces the number of primitives by approximately half, resulting in significantly faster BVH construction. Moreover, all the data for a quad including the four vertex positions, the mesh index and per-triangle information such as the triangle indices and vertex order can be stored in a single 64 byte cache line. This also leads to a more efficient hardware implementation as a triangle-pair can be tested with a single cache line fetch.

For our evaluation, we render six example scenes (see Figure 2) with three-bounce path tracing in pbrt [PH17] and process the rays in our functional model. The ray distribution contains incoherent rays corresponding to *closest-hit* and *any-hit* ray queries. Figure 3 shows the average number of traversal steps per ray with our traversal algorithm and a different short stack sizes. The overhead in traversal steps with five stack entries is about 1-10% for *closest-hit* queries and just 0-5% for *any-hit* queries (see Table 1), coming close to the efficiency of a full stack.

With a short stack of five entries, we store the root node (64 bits),

compressed using 8-bit quantized bounding planes, similar to Ylitie et al. [YKL17]. We observe that a 6-wide BVH typically reduces the number of traversal steps per ray by 20-40% compared to a 4-wide BVH and only adds 10-15% more steps than an 8-wide BVH.

Instead of using triangles as the base primitive type, we employ a simple compression scheme for geometry, merging two triangles

current node (32 bits), restart trail (3 × 32 bits), current depth (5 bits) and the short stack entries (5 × 32 bits), requiring less than 44 bytes of storage. Assuming a maximum tree depth of 32, a full stack on the other hand requires 136 bytes of storage, including 64 bits for the root node and 32 × 32 bits for the stack entries. Therefore, our approach consumes approximately 3× less memory, compared to a full stack.

It is interesting to note that in some cases, the number of quad intersections with the short stack is sightly lower than the full stack, even though the leaf nodes are traversed in the same order. This can be attributed to traversal restarts which results in some nodes being re-tested and culled if a closer primitive hit is found in between.
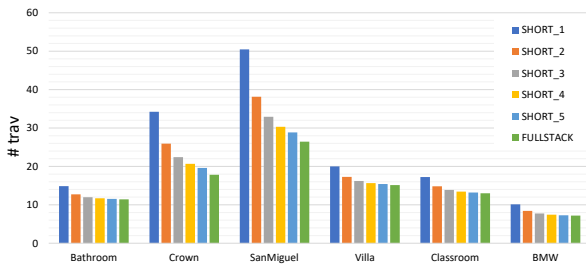

Figure 3: *The number of traversal steps per ray, averaged over any-hit and closest-hit ray queries with different short stack sizes and with a full stack. The overhead in traversal steps with a single short stack entry is as high as 90%, a stack size of four brings the overhead close to 16% and with five entries, it is less than 10%.*

We also analyze our explicit stack culling extension in the last two columns of Table 1. Stack culling reduces the number of ray quad intersection tests by 2-14% while increasing traversal steps by 1-7%. With hardware based traversal implementations, the cost of a ray primitive intersection test can be significantly higher than the cost of ray box tests for an inner node [Kee14]. Therefore, stack culling achieves a good trade-off with a slight increase in the number of traversal steps for a more significant reduction in ray primitive intersection tests.

5. Conclusion and Future Work

We have proposed a generalization of binary BVH traversal with a short stack to BVHs with an arbitrary width. Our algorithm does not require additional data in the BVH nodes or separate structures in memory to backtrack traversal. Compared to a full stack, our algorithm requires 3× less stack memory and only increases the number of traversal steps by a small percentage. We also introduce an extension of our traversal algorithm for stack culling that can reduce the number of expensive ray primitive intersection tests with a small overhead in the number of traversal steps.

Our approach lends itself naturally to a dedicated hardware implementation where the available on-chip memory is heavily constrained. However, with an implementation that uses compressed BVH nodes as well as our short stack traversal algorithm, the latency and bandwidth for fetching leaf geometry is likely to become the next dominant bottleneck. Although we do apply some degree

of geometry compression by converting triangle pairs to quads, a more effective geometry compression scheme could be an important goal for the future.

References

[AK10]   AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics* (2010), Eurographics Association, pp. 113–122. 1

[ÁS14]   ÁFRA A. T., SZIRMAY-KALOS L.: Stackless Multi-BVH traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140. 1, 2

[BAM13]   BARRINGER R., AKENINE-MÖLLER T.: Dynamic stackless binary tree traversal. *Journal of Computer Graphics Techniques (JCGT) 2*, 1 (2013), 38–49. 1

[BK16]   BINDER N., KELLER A.: Efficient stackless hierarchy traversal on GPUs with backtracking in constant time. In *Proceedings of High Performance Graphics* (2016), pp. 41–50. 2

[FLP*17]   FUETTERLING V., LOJEWSKI C., PFREUNDT F.-J., HAMANN B., EBERT A.: Accelerated single ray tracing for wide vector units. In *Proceedings of High Performance Graphics* (2017), ACM, pp. 1–9. 3

[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* (2011), SCCG '11, ACM, pp. 7–12. 1, 2

[Kee14]   KEELY S.: Reduced precision for hardware ray tracing in GPUs. In *Proceedings of High Performance Graphics* (2014), pp. 29–40. 1, 5

[KMSB18]   KILGARIFF E., MORETON H., STAM N., BELL B.: NVIDIA Turing Architecture In-Depth, 2018. URL: http://devblogs.nvidia.com/nvidia-turing-architecture-in-depth. 1

[Lai10]   LAINE S.: Restart trail for stackless BVH traversal. In *Proceedings of the Conference on High Performance Graphics* (2010), pp. 107–111. 1, 2

[PH17]   PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory To Implementation*, 3nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017. 4

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

[VAMS16]   VAIDYANATHAN K., AKENINE-MÖLLER T., SALVI M.: Watertight ray traversal with reduced precision. In *Proceedings of High Performance Graphics* (2016), Eurographics Association, pp. 33–40. 1

[VKJT16]   VIITANEN T., KOSKELA M., JÄÄSKELÄINEN P., TAKALA J.: Multi bounding volume hierarchies for ray tracing pipelines. In *SIGGRAPH ASIA 2016 Technical Briefs* (2016), SA '16, ACM, pp. 8:1–8:4. 1

[WWB*14]   WALD I., WOOP S., BENTHIN C., JOHNSON G., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (2014), 1–8. 1, 3

[YKL17]   YLITIE H., KARRAS T., LAINE S.: Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics* (2017), pp. 1–13. 1, 3