An Evaluation of Multi-Hit Ray Traversal in a BVH Using Existing First-Hit/Any-Hit Kernels

Jefferson Amstutz
Intel Corporation

Christiaan Gribble
Applied Technology Operation
SURVICE Engineering

Johannes Günther
Intel Corporation

Ingo Wald
Intel Corporation

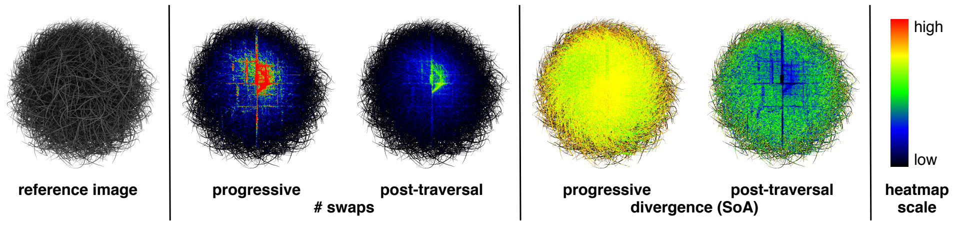

Figure 1. **Efficient multi-hit ray tracing using existing BVH traversal algorithms.** Many ray-tracing applications in optical and non-optical domains require multiple intersections along each ray, or so-called *multi-hit ray traversal*. Sorting hit points efficiently in light of memory bandwidth limitations has a major impact on multi-hit ray-tracing performance in a BVH. Here, heatmap visualizations depicting the number of swaps imposed by sorting, as well as SIMD utilization of the sorting process, reveal the increased efficiency of our post-traversal selection sort technique.

Abstract

We explore techniques for multi-hit ray tracing in a bounding volume hierarchy (BVH) using existing ray traversal kernels and intersection callbacks. BVHs are problematic for implementing multi-hit ray traversal correctly due to the potential for spatially overlapping leaf nodes. We demonstrate that the intersection callback feature of modern, high performance ray-tracing APIs enable correct and efficient implementation of multi-hit ray tracing despite this concern. Moreover, the callback-based approach enables multi-hit ray tracing using existing, highly optimized BVH data structures, mitigating maintenance issues imposed by hand-tuned multi-hit traversal kernels across various hardware architectures. Results show that memory-bandwidth limitations and SIMD vector width of the target hardware platform dictate ideal hit-point memory layout, as well as the point at which sorting should occur, in order to maximize performance with existing BVH traversal algorithms.

1. Introduction

Ray tracing is an important technique in many optical and non-optical rendering applications. Research over the past 30 years has addressed performance of both ray traversal and acceleration structure construction. For example, first-hit ray traversal is used for visibility [Appel 1968] and, when applied recursively [Whitted 1980; Kajiya 1986], for advanced effects including direct and indirect illumination in traditional optical-rendering (image synthesis) applications. Likewise, any-hit ray traversal can be used for efficient occlusion queries when computing shadows or ambient occlusion. Many applications in non-optical rendering—our particular area of interest—also benefit from fast first-hit and any-hit traversal.

However, some applications require more than one intersection along each ray, or so-called *multi-hit ray traversal*. In these instances, simply using first-hit traversal to collect multiple hit points is an enticing approach: at each valid intersection, a new ray is traced using the recently-generated hit point, adjusted by a small ϵ term, as its origin.

Unfortunately, an approach to multi-hit traversal based on first-hit traversal with ϵ-offsets is not usable in practice, since intersections may be erroneously repeated or missed entirely, leading to incorrect results [Gribble et al. 2014]. An operation that correctly and efficiently solves the multiple intersection problem, even in numerically difficult situations, is required.

A multi-hit ray traversal algorithm is one that returns information concerning the N-closest ray/primitive intersections in ray order. Multi-hit traversal generalizes both first-hit traversal (where $N = 1$) and all-hit traversal, a scheme in which ray queries return information concerning every intersected primitive (where $N = \infty$), while accommodating arbitrary values of N between these extremes.

A correct multi-hit ray traversal algorithm is a necessary, but insufficient, condition for modern applications, however; performance is also critical for both interactivity and fidelity in many scenarios. For example, interactivity is often limited by performance of ray traversal or the rate at which acceleration structures for dynamic geometry can be updated. Likewise, performance directly impacts accuracy of ray-based Monte Carlo simulations that converge to the correct solution by tracing large numbers of rays. Unfortunately, satisfying multi-hit traversal constraints has contributed to the inability of many existing multi-hit applications to employ modern, fast ray-tracing APIs, and thus, to their dismal performance.

Modern ray-tracing engines address performance concerns by hiding complicated, highly optimized ray-tracing kernels behind clean, well-designed APIs. Embree [Wald et al. 2014] implements a highly optimized ray-tracing engine for x86-based CPUs and Intel® Xeon Phi™ coprocessors. Similarly, OptiX[1] [Parker et al. 2010] is a

[1]OpitX is a trademark of the NVIDIA Corporation.

highly optimized ray-tracing engine designed for NVIDIA[2] GPUs. To accelerate ray queries, Embree and OptiX utilize numerous bounding volume hierarchy (BVH) variants based on application characteristics provided to the engine by the user. These engines provide fast first-hit and any-hit ray traversal operations for use in applications across optical and non-optical domains.

Previous work on multi-hit ray traversal [Gribble et al. 2014] assumes an acceleration structure based on spatial subdivision, in which leaf nodes of the structure do not overlap. With such structures, ordered traversal—and therefore generating ordered hit points—is straightforward: sorting is required only within, not across, leaf nodes. However, ordered traversal in a structure based on object partitioning, such as a BVH, is not achieved so easily; in fact, ordered ray traversal is at odds with the BVH structures employed by modern, high performance ray-tracing engines. At first glance, then, it is unclear that a BVH can provide reasonable multi-hit performance. Surprisingly, however, we demonstrate several techniques to implement multi-hit traversal in a BVH efficiently.

Specifically, we explore multi-hit ray tracing and its performance characteristics in Embree and OptiX. We are motivated by an API feature we call *intersection callbacks*, which allow clients to inject application-specific ray/primitive intersection processing into traversal. In Embree, *intersection-filter* functions enable users to express per-hit processing, while in OptiX, user-defined *any-hit* programs implement this feature.

Intersection-filter functions are a recent addition to the Embree API. If a filter function is defined, it is invoked whenever a triangle is successfully intersected, irrespective of distance along the ray. Intersection filters are intended to inject user-defined ray/primitive rejection tests into traversal. For example, an otherwise valid intersection could be rejected based on the primitive's material properties (an alpha texture, for example); in this case, traversal continues until either another candidate intersection is found (and accepted) or when the ray exits the structure.

We observe that intersection rejection is not the only operation that can be implemented using intersection filters; when viewed more generally, this feature enables client applications to inject arbitrary logic, executed whenever a valid ray/primitive intersection is found, and continue or terminate ray traversal.

Like intersection-filter functions, any-hit programs in OptiX are invoked at any ray/primitive intersection, irrespective of distance along the ray. Occlusion queries are a common application of any-hit programs: in the case of shadow or ambient occlusion rays, the distance at which the ray encounters a valid intersection is not of concern, only that it does, in fact, encounter such an intersection. Combined with OptiX's `rtTerminateRay` function, occlusion rays can be terminated as soon as any

[2]NVIDIA is a trademark of the NVIDIA Corporation.

primitive generates a valid intersection. In a similar manner, we combine any-hit programs with OptiX's `rtIgnoreIntersection` function to continue, rather than terminate, ray traversal.

Though we recognize the diversity of applications using multi-hit ray traversal—for example, optical and non-optical domains may impose very different ray-tracing requirements—we investigate multi-hit traversal within a set of constraints that characterize many, but not all, applications of multi-hit ray tracing. In particular, we assume:

1. *Existing traversal kernels are used without modification.* Embree and OptiX already provide highly tuned traversal kernels for first-hit and any-hit queries. We are motivated by non-optical rendering in production environments and, therefore, seek to avoid additional maintenance burden imposed by unique multi-hit traversal kernels. Moreover, a ray-tracing engine may not support user-defined kernels, as in the case of OptiX. We therefore use existing traversal kernels without modification.

2. *Rays are coherent.* Vectorization of ray traversal and hit-point sorting is critical to good performance. We focus on spatially coherent rays, which are more likely to be traversed and intersected coherently, and which are, therefore, amenable to vector processing.

3. *Hit-point buffers are pre-allocated.* The number of gathered hit points varies among rays, but using a large, pre-allocated buffer for collecting hit points alleviates the need for memory allocation during traversal. Though we believe better strategies for managing hit-point buffers may be possible—particularly when guided by application-specific constraints—we opt for simplicity and use a large, pre-allocated buffer in this work.

BVH traversal algorithms specific to the multiple intersection problem, incoherent ray distributions, and clever memory management strategies are beyond the scope of this work and represent areas of future work.

With these assumptions in mind, we show that correct and efficient multi-hit traversal can be implemented using intersection callbacks and state-of-the-art, finely-tuned BVHs with performance competitive to that achieved with algorithms specific to multi-hit ray traversal in structures based on spatial subdivision [Gribble et al. 2014]. Importantly, this callback-based approach integrates multi-hit traversal into existing APIs in a maintainable way: intersection-callback logic belongs to the client, so leveraging this approach does not require custom, hand-coded kernels that further compound the already numerous maintenance tasks necessary to support production ray-tracing applications.

Valid hit points must be sorted to meet the ray-order criterion imposed by multi-hit traversal, so we also examine two factors that impact sorting performance: in-memory hit-point structure layout and the point at which sorting occurs. We show that hardware platforms with high memory bandwidth per-ray benefit from vectorized post-traversal sort, while lower memory bandwidth platforms benefit from scalar progressive sort during traversal.

2. Methodology

Multi-hit ray traversal is a class of ray traversal algorithms that finds one or more, and possibly all, primitives intersected by a ray, ordered by point of intersection. Gribble et al. [2014] introduce two algorithms for multi-hit traversal in acceleration structures based on spatial subdivision: a naive algorithm that essentially implements all-hit traversal but returns (at most) the closest-N hit points, and a buffered algorithm with early-exit, which exploits ordered traversal to terminate after (at least) the closest-N hit points have been found.

2.1. Naive vs. Buffered Multi-Hit Traversal

In the naive algorithm, rays iteratively traverse the acceleration structure, recording information about each intersection in sorted order. Once traversal is complete, a per-hit user-level callback is invoked to process each hit point. The return value of this callback indicates whether or not additional intersections should be processed.

The naive algorithm is simple and effective: it imposes very few constraints on an actual implementation, it does not assume a particular acceleration structure, and it allows the user to process as many intersections as desired.

However, the naive algorithm is potentially very slow: it essentially implements the all-hit traversal scheme, as opportunities for early-exit occur during intersection processing, only after all intersections have been found, and not during traversal.

The buffered multi-hit traversal algorithm addresses these issues. As in the naive algorithm, this version maintains a per-ray data structure to record valid intersections. However, rather than allow the list to grow without bound, an ordered buffer of a fixed size is used. This algorithm also exploits opportunities for early-exit when $N < \infty$: as before, the per-hit callback indicates whether or not additional intersections are desired. For cases in which they are not, ray traversal—not just intersection processing—ends; otherwise, processing continues with the next node.

In acceleration structures based on spatial subdivision, buffered multi-hit traversal is straightforward: leaf nodes do not overlap and are easily traversed in front-to-back order.

These properties are illustrated in the left panel of Figure 2. In this example, the ray traverses the near node first and computes a hit point with the blue triangle, but

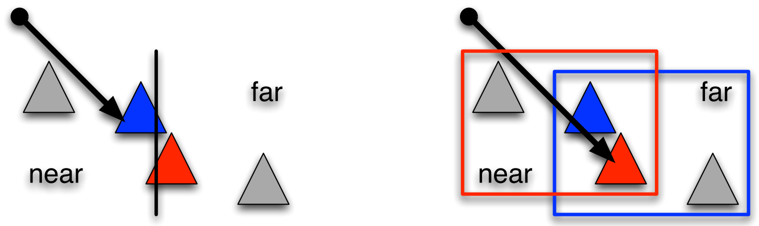

Figure 2. The problem of overlapping nodes. Ordered traversal in acceleration structures based on spatial subdivision is straightforward (left). Though primitives spanning a split plane must be handled appropriately, spatial subdivision simplifies the problem of returning multiple intersections in ray-order. In contrast, structures based on object subdivision complicate this situation (right). Here, a valid intersection is identified in the near node, though the hit point is clearly second in ray-order. Generally, all nodes must be traversed before the hit points are correctly ordered.

does not yet consider the red triangle in the far node. At this point, the computed hit point is inserted into the local buffer in ray-order. Importantly, as the ray exits the near node, the hit points gathered thus far can be returned to the application, enabling the client to request more hit points and continue traversal or to terminate.

In this manner, acceleration structures based on spatial subdivision provide opportunities for improved multi-hit performance: the number of hit points—and thus the size of the buffer used to retain these data—can be bounded to optimize traversal performance and memory requirements, and opportunities to terminate traversal after processing each node can be exploited because they are processed in ray-order.

Unfortunately, these same properties are not true for a BVH. Though at any level primitives belong to only one node in the tree, the nodes themselves may overlap, so strict front-to-back traversal cannot be easily resolved. This characteristic, in turn, implies that hit points may be generated out of order.

This issue is illustrated in the right panel of Figure 2. Here, the ray computes a valid intersection with the red triangle in the near node, though the hit point is clearly second in ray-order. However, the ray will not intersect the blue triangle until it traverses the far node. To address this issue, both nodes must be traversed before the hit points are correctly resolved—there is no opportunity for early exit when nodes overlap.

A buffered multi-hit traversal algorithm for a BVH must, therefore, store and sort hit points across leaf nodes until at least the point at which the set of already-traversed nodes contains none that overlap untraversed nodes. Techniques to satisfy this constraint would impose significant burden on BVH construction (to enumerate and store data about overlapping nodes for use during traversal) and ray traversal itself (to correctly determine when the no-overlap constraint is satisfied). In this case, existing

BVH data structures, construction algorithms, and ray traversal kernels cannot be used as is to implement the buffered multi-hit algorithm described by Gribble et al. [2014].

Specifically, in a multi-hit implementation that is based on intersection callbacks and that utilizes unmodified first-hit or any-hit BVH traversal kernels, the interval over which intersections are considered valid is [ε, *t-near* = *t-max*], where *t-max* $\geq$ *t-exit*, the distance at which the ray exits the root-level bounding volume. This interval is required to ensure that all nodes are visited during traversal and, as a result, that all hit points are gathered.

Once initialized, this interval does not—and cannot—change during traversal due to restrictions on intersection callbacks imposed by current ray-tracing APIs. For example, the multi-hit traversal intersection callback for Embree simply saves data corresponding to each valid hit point in a per-ray data structure and rejects the intersection to continue traversal. In this *reject-intersection* case, the original value of *t-near*—that is, *t-max*—is restored by a routine that is automatically invoked by Embree after the intersection callback and that is inaccessible to the client.

We are thus forced to use naive multi-hit traversal. This approach is not without merit, however: ray traversal remains simple—we need only traverse rays all the way through the structure. More importantly, the multiple intersection problem is solved correctly using existing BVH structures and traversal algorithms—the effort required to optimize BVH traversal for various hardware architectures and SIMD vector widths remains intact. We therefore focus on techniques to maximize efficiency of gathering and sorting hit points using the existing, highly optimized BVH acceleration structures provided by Embree and OptiX.

2.2. Hit-Point Data and Memory Layout

The contents of the hit-point data structure have important performance implications, but depend heavily on application-level requirements. Smaller hit-point data structures increase cache density and reduce the cost of memory transactions imposed by hit-point processing, whereas larger structures have the opposite effect. We store a minimally complete set of data for each valid hit point: intersection distance, geometric normal, and primitive/object identifiers.

Effective SIMD programming on both CPUs and GPUs requires careful attention to memory layouts to maximize bandwidth performance. Typically, arrays-of-structures (AoS) result in poor access patterns, as operations involving individual members of the structure cannot be coalesced and thus incur expensive scatter/gather memory operations.

Conventional wisdom suggests that wide SIMD architectures benefit more from the structures-of-arrays (SoA) layout, which is used to coalesce member accesses and minimize incoherent memory reads and writes. However, when presented with incoherent sorting operations among rays, scatter/gather operations occur per-element of the SoA hit-point structure. Each member of the structure is significantly smaller than a SIMD vector, resulting in vast underutilization. Alternatively, with the AoS data layout, vector memory operations read or write entire hit-point structures, which uses each operation more efficiently during sorting.

In order to better understand the effect of these factors on multi-hit ray-tracing performance, we examine both AoS and SoA data layouts for hit-point buffers in Section 3.

2.3. Hit Point Sorting

Gathering all hit points along a ray requires the ability to continue BVH traversal after a hit point has been found. As noted above, we leverage Embree's intersection filters and OptiX's any-hit programs to accomplish this task. In each implementation, these callbacks simply store hit-point data in a local buffer and unconditionally continue traversal.

Hit points must be sorted to meet the ordering constraint of multi-hit ray traversal, so we explore two sorting methods: progressive insertion sort during traversal and post-traversal selection sort.

Progressive insertion sort. In this first approach, intersections are sorted as they are inserted into the local buffer. Importantly, this approach follows the general form of standard BVH traversal. However, when a valid intersection is found, it is interserted into the hit list in ray-order. This algorithm flows naturally from a direct application of naive multi-hit traversal, under the constraints imposed by a BVH.

This technique strives to maximize cache locality—hit points are likely already in cache during traversal. Specifically, if a hit point needs to be swapped, it will most often swap with its closest neighboring hit points. Thus, when a valid intersection is found, adjacent hit points will likely be valid in cache and can thus be swapped without the penalty of global memory latency.

Post-traversal selection sort. In the second approach, intersections are gathered into the local buffer without regard for proper front-to-back ordering. In this case, the traversal process is nearly identical to the previous approach. However, a valid hit point is simply appended to the local buffer, and the entire collection is sorted after traversal is complete but before returning to the client.

This approach strives to keep both traversal and sorting operations amenable to SIMD processing. During traversal, rays within a SIMD vector may stall because of neighboring rays that must find and store additional hit points. The stall period is directly proportional to the time required to insert hit-point data into the local buffer. If sorting is postponed until after traversal, the potential stall period is reduced. Furthermore, divergence among rays during traversal negatively impacts sorting coherence. As shown in Section 3, the actual sorting process is significantly more coherent when the set of intersection points to be sorted is well-known and bounded (after ray traversal) than when this information is unknown (during ray traversal).

3. Results

We implemented each technique described in Section 2 using the eight scenes and viewpoints depicted in Figure 3. For CPU and Intel Xeon Phi coprocessor results, we build on the OSPRay rendering framework [Intel Corporation 2014], which provides a flexible mechanism to execute Embree kernels in various hardware and software configurations. For GPU results, we use OptiX SDK examples as the basis for our implementation. We also compare against Rayforce[3], the open source GPU ray-tracing engine used in previous work [Gribble et al. 2014]. Rayforce employs an acceleration structure—most aptly characterized as a graph—that is based on spatial subdivision; these comparisons characterize the impact of naive multi-hit ray traversal, which is required when using existing BVH traversal algorithms, as discussed above.

For each scene, we render a series of 100 frames at 1024×1024 pixel resolution. Each image is generated using visibility rays from a pinhole camera, with a single sample per pixel. The Embree results are collected on dual 18-core Intel® Xeon® E5-2699 v3 processors or a single Intel Xeon Phi 7120A coprocessor. The OptiX results are collected on a single NVIDIA GTX Titan GPU.

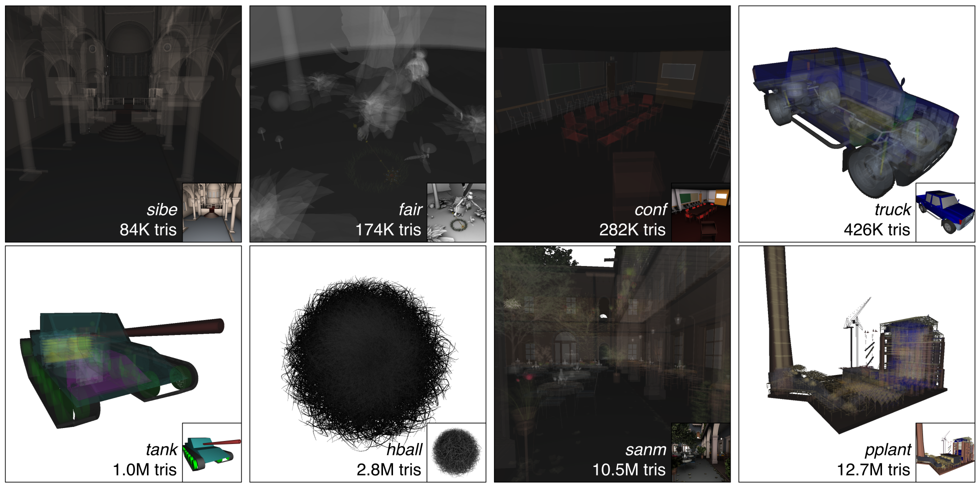

Figure 3. Scenes used for performance evaluation. Eight scenes of varying geometric and depth complexity are used to evaluate the performance of our multi-hit ray traversal schemes. The first-hit visible surfaces in many of these scenes hide significant internal complexity, making them particularly useful as tests of multi-hit traversal.

[3]*Rayforce*: Exceptional performance through non-traditional means. Source code is available via `http://rayforce.survice.com`.

3.1. First-hit vs. Unsorted All-hit

As a baseline, we compare first-hit traversal performance to the performance of gathering all hit points without sorting. This comparison is instructive in two ways. First, first-hit performance is a well-understood quantity for measuring ray traversal costs and, therefore, provides insight into the relative impact of gathering all hit points using existing BVH traversal algorithms. Second, unsorted all-hit traversal performance provides an upper bound for naive multi-hit traversal: we cannot possibly gather and sort all hit points any faster than we can gather, but not sort, the same points. Performance is reported in terms of millions of rays per second (Mrps); the results are shown in Figure 4. We observe a significant performance penalty—about 44% for our test scenes—when gathering all unsorted hit points, particularly for scenes of high depth complexity, or those with a larger average number of hit points per ray.

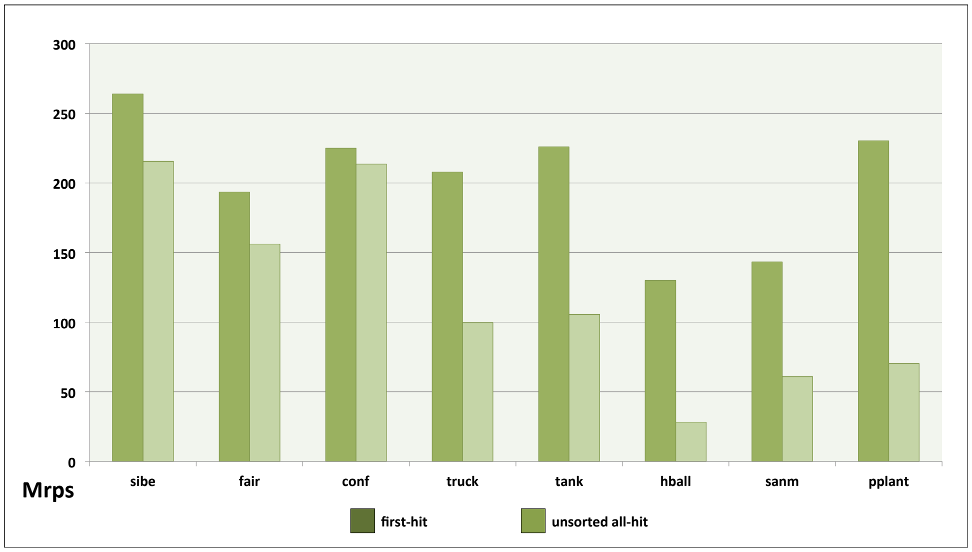

Figure 4. First-hit vs. unsorted all-hit. Here, the graph compares performance (in Mrps) between first-hit traversal and unsorted all-hit traversal. Gathering all hit points along a ray clearly impacts performance (about 44% for our test scenes) and provides an upper bound on performance for naive multi-hit traversal.

3.2. Observed Efficiency

We next examine the impact of SIMD hardware on multi-hit traversal. SIMD efficiency correlates directly to throughput, so we measure SIMD utilization to understand how sorting operations disrupt coherence in the SoA layout.

We observe that SIMD utilization is improved by deferring the sort until after traversal. Divergence of neighboring rays is an issue even with first-hit ray traversal;

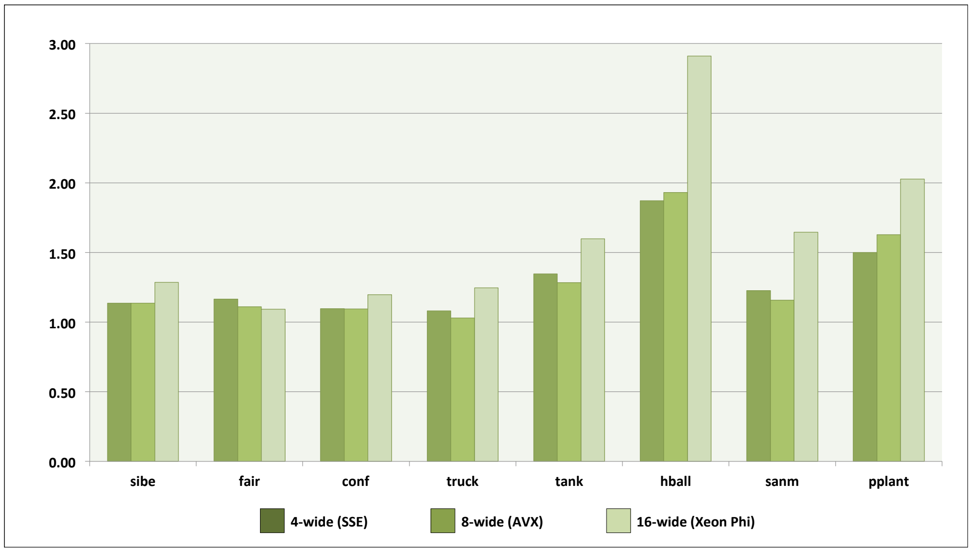

Figure 5. **SIMD utilization improvement across vector widths.** Here, the graph depicts SIMD utilization improvement across 4-, 8-, and 16-wide SIMD vector units when deferring hit-point sorting until after traversal. On average, post-traversal sorting improves SIMD utilization by a factor of about 1.41 for our test scenes.

sorting intersections during traversal only increases the probability that rays and the corresponding sorting operations will diverge because the number of intersections along each ray may differ. Post-traversal sort alleviates this issue and, as noted in Section 2, exploits a priori knowledge of the number of hit points to sort. As can be seen in Figure 5, SIMD utilization is improved over all vector widths tested and, once again, the benefit is higher for dense scenes and wider vector units.

In particular, we see an overall SIMD utilization improvement by a factor of about 1.41 when sorting occurs after traversal. A breakdown of specific values for 8-wide vectors is shown in Table 1. Here, we see that triangle density also impacts sorting efficiency. For example, we observe that *sanm* renders with 60% more SIMD efficiency in sorting than does *hball*, despite having nearly four times the number of triangles. Dense geometry not only has a higher total intersection count, but the underlying BVH is more likely to have a larger number of overlapping leaf nodes, which incurs a higher sorting burden during traversal.

We also observe that data layout impacts measurements of SIMD divergence when sorting hit points. When using the SoA memory layout, these measurements are a direct indicator of sorting coherence, as characterized by the results by Table 1. However, AoS hit points are sorted with scalar instructions, so SIMD vectors are no longer used across multiple rays and hit points; instead, entire hit-point structures are accessed via vector memory instructions. As a result, the AoS layout actually improves performance on wider SIMD hardware, as discussed below.

scene	**avg hits/ray**	**# swaps**	**% util**	**% fewer swaps**	**util improve**
sibe	2.4	0.3	78.7	0.00	1.1×
		0.3	89.4		
fair	2.9	1.0	84.7	40.0	1.1×
		0.6	94.1		
conf	2.0	0.4	83.0	0.0	1.1×
		0.4	90.9		
truck	18.0	22.3	86.3	64.7	1.0×
		7.9	88.0		
tank	10.2	5.7	68.3	45.6	1.3×
		3.1	87.6		
hball	21.3	24.1	33.5	64.7	1.9×
		8.5	64.7		
sanm	6.5	4.1	66.7	56.1	1.2×
		1.8	77.2		
pplant	14.5	29.0	48.1	72.4	1.6×
		8.0	78.3		

Table 1. Observed efficiency of our multi-hit ray traversal techniques in Embree. Here, the data compare important performance characteristics for progressive insertion sort (top row, columns 3 and 4) and post-traversal selection sort (bottom row, columns 3 and 4) using two AVX/AVX2-enabled Intel® Xeon® processors.

Finally, we see that swap counts are reduced when using post-traversal selection sort. Dense scenes, in particular, benefit the most, as more hit points are likely to be out of order. While swap counts do not significantly impact overall performance, we nevertheless observe a reduction in the amount of work imposed by sorting.

3.3. Observed Performance

Multi-hit ray-tracing performance reveals that different factors determine the runtime characteristics on each of the hardware platforms considered here. The two largest factors contributing to overall performance are memory bandwidth and throughput. As discussed above, SIMD utilization improves throughput, but results show that memory bandwidth becomes increasingly important with wider SIMD vectors and more in-flight rays. We report performance in terms of millions of hits per second, or Mhps, throughout this section.

CPU performance. The CPU results shown in Figure 6 reveal a noticeable speedup across all scenes when using coherent progressive sort with SoA hit-point buffers, particularly for scenes with high depth-complexity. In this case, the 8-wide SIMD vectors

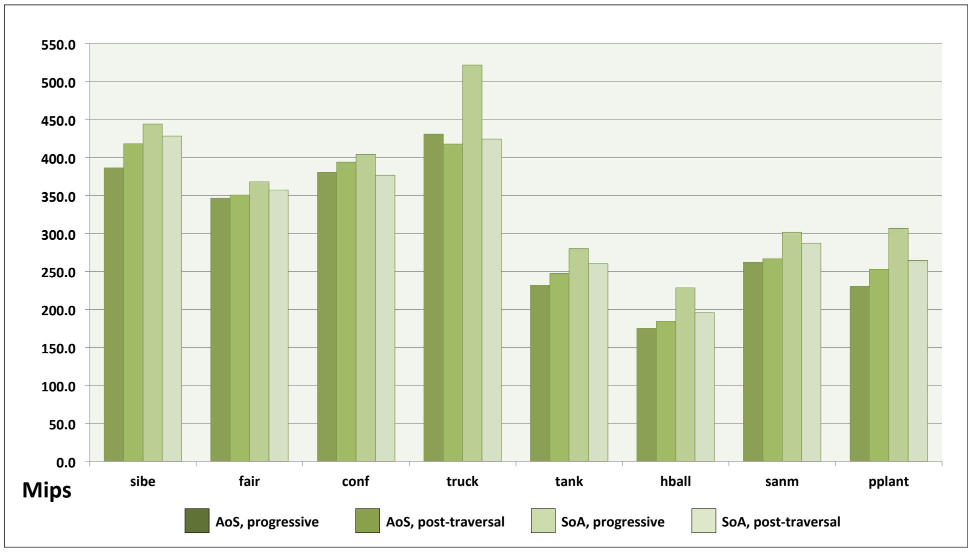

Figure 6. **Performance on two Intel® Xeon® E5-2699 v3 processors.** Here, the graph depicts performance (in Mhps) of our sorting techniques and hit-point data layouts on the CPU. On average, coherent progressive sort with SoA hit-point data performs best for our test scenes.

are able to maintain better coherence and thus benefit from sorting during traversal. Moreover, of the hardware platforms tested, the CPUs have the best available cache

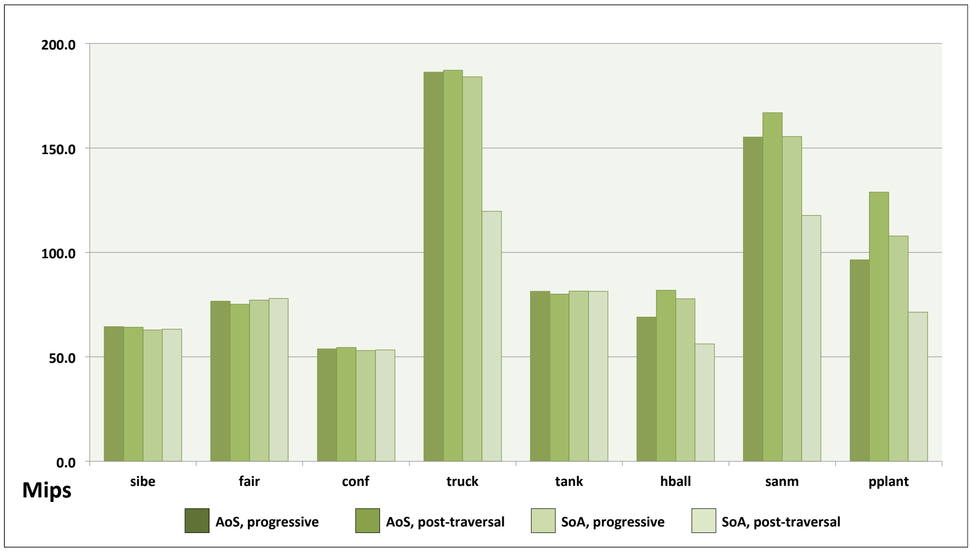

Figure 7. **Performance on a single Intel® Xeon Phi ™ 7120A coprocessor.** Here, the graph depicts performance (in Mhps) of our sorting techniques and hit-point data layouts on the coprocessor. On average, coherent post-traversal sort with AoS hit-point data performs best, despite the observed limitation of OSPRay's frame-compositing bottleneck.

capacity per ray at any given time. Thus, when rays encounter intersections, neighboring hit points that may require swapping are likely in the cache already.

Xeon Phi coprocessor performance. The Xeon Phi coprocessor results shown in Figure 7 reveal that post-traversal sort with AoS hit-point data layout generally performs best for our test scenes. The expected trend is that wider SIMD vectors benefit more from careful data layout and attention to coherence. On the contrary, we observe that memory bandwidth has a larger impact on performance in this context. Xeon Phi has less available cache per ray during traversal, which introduces shared memory bandwidth as the primary factor influencing performance. However, enough cache is available so that sorting coherently after traversal generally maximizes overall performance. We also observe that OSPRay's Xeon Phi frame-compositing operations become the limiting factor when measuring performance for the smaller test scenes.

NVIDIA GPU performance. The GPU results shown in Figure 8 reveal that the best performing technique is progressive sort with AoS hit-point data layout. On the GPU, huge numbers of rays are traced concurrently, which requires a large buffer for per-ray hit-point data. This requirement causes almost all hit-point manipulations to incur expensive cache misses, so efficient memory operations have the greatest impact on overall performance. Similar to the Intel Xeon Phi coprocessor, sorting AoS data is

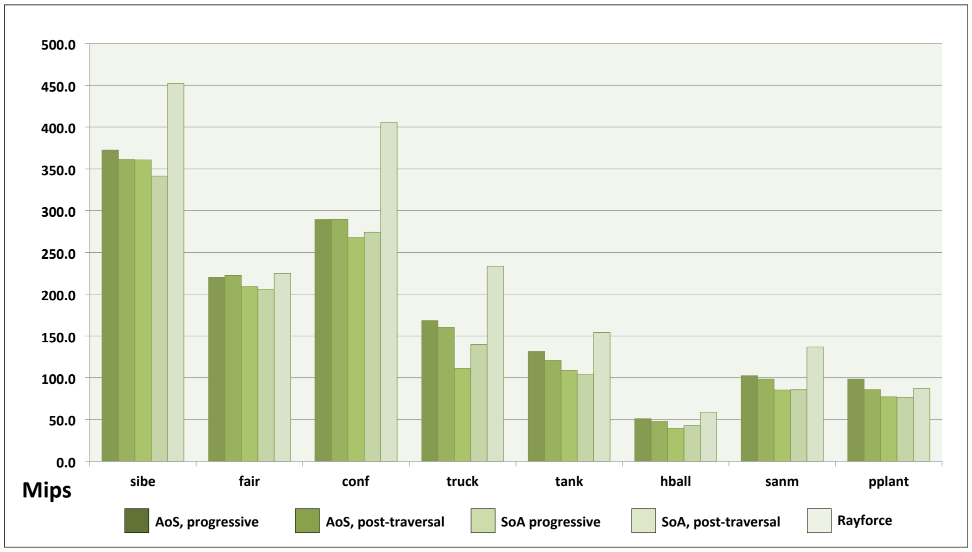

Figure 8. **Performance on a single NVIDIA GTX Titan.** Here, the graph depicts performance (in Mhps) of our sorting techniques and hit-point data layouts on the GPU. For comparison to multi-hit traversal in structures based on spatial subdivision, results for Rayforce are also shown. On average, progressive sort with AoS hit-point data performs best among our BVH techniques for these scenes.

more bandwidth-efficient on this platform. The 32-wide SIMD vectors make it more likely to store and sort incoherent hit points, which further justifies optimizing for memory bandwidth. Lastly, progressive sorting is generally the faster technique because the likelihood of neighboring hit points being in cache is highest during traversal.

Figure 8 also shows that we achieve performance competitive with multi-hit traversal in Rayforce, the GPU ray-tracing engine used in previous work [Gribble et al. 2014]. In fact, using OptiX, we achieve about 86% (on average) of performance in Rayforce for the sorted all-hit case on a single NVIDIA GTX Titan GPU. We are motivated by non-optical rendering in production environments, so we are pleased with these results, particularly given the conflict between ordered ray traversal and BVHs. Moreover, the maintenance issues alleviated by our ability to utilize existing—and continually improving—ray-tracing engines significantly outweighs the difference in absolute performance: our production rendering applications are sufficiently interactive to exceed our users' needs.

4. Conclusions and Future Work

We explore techniques for multi-hit ray traversal in a BVH, an acceleration structure common to modern, high performance ray-tracing APIs. We demonstrate a novel use of intersection callbacks in Embree and OptiX to enable correct and efficient implementation of multi-hit traversal, despite complications arising due to overlapping nodes. Importantly, intersection callbacks enable implementation of multi-hit ray traversal using finely-tuned BVH data structures, construction algorithms, and ray traversal kernels that already exist in these engines.

Specifically, we examine the various combinations of hit-point data layout and the point at which sorting occurs. Results show that deferring sort until after traversal improves efficiency by reducing memory swaps and increasing SIMD vector utilization. However, post-traversal sort does not necessarily guarantee maximum performance: memory bandwidth constraints on current hardware also impact sorting performance and can be justification alone for preferring an AoS or SoA hit-point data layout, regardless of efficiency.

We examine only the cost of multi-hit ray traversal and hit-point sorting in this work. However, production multi-hit applications in both optical and non-optical domains will also shade or otherwise process the returned hit points; these operations further complicate the problem of maintaining high SIMD utilization throughout rendering. Future work will investigate techniques to handle these operations efficiently, particularly in physics-based simulation applications. Any high performance approach will likely benefit from a wavefront formulation [Laine et al. 2013], in which shading operations are sorted by type and grouped to maximize SIMD utilization—

implying the use of SoA data. However, specific applications should carefully observe memory bandwidth, as it has a significant impact on multi-hit ray traversal.

We consider the performance of multi-hit traversal only in a BVH, and previous work did not consider SIMD utilization in spatial subdivision structures. Beyond the implications for multi-hit traversal in a BVH, the results of this work will also be instructive in any future exploration of multi-hit SIMD utilization in acceleration structures based on spatial subdivision.

Currently, our formulation in OptiX has an overly burdensome memory requirement. Because OptiX maintains a large number of in-flight rays, hit-point buffers must be very large. Future work will investigate techniques to reduce the memory requirement when tracing multi-hit rays using a BVH on the GPU.

An initial comparison of each Embree filter-based technique to hand-coded kernel versions shows that kernel-based implementations perform 25% better (on average) than intersection filters. We observe that hand-coded traversal kernels avoid unnecessary function call overhead and storage of temporary data that is not actually necessary for multi-hit ray traversal. Future work will explore the benefit of implementing unique BVH multi-hit traversal kernels for improved performance.

Additionally, an implementation using index-based storage and sorting of hit-point data has little impact on performance beyond the results presented in Section 3. Similar to linked-list fragment buffers in depth peeling [Carpenter 1984; Crassin 2010; Yang et al. 2010], the index-based technique stores full hit-point data in an unordered buffer and sorts only indices into that buffer. On average, progressive and post-traversal index sorting using each of AoS and SoA data layouts improves performance by less than one percent, and no one combination is uniformly faster across all of our test scenes. Moreover, accessing unordered hit-point data is generally less coherent than corresponding access in direct-sorting techniques, a property that may negatively impact shading or other downstream operations, particularly when combined with the additional level of indirection imposed by indexing. However, specific applications should consider hit-point data requirements in this case, as relative performance of the index-based technique may depend on this application-level constraint.

Finally, we examine only coherent multi-hit rays. Understanding the performance impact of incoherent ray distributions on multi-hit traversal should also be examined. Future work will investigate incoherent ray distributions across various multi-hit ray-tracing applications to identify techniques that maximize performance in this context.

References

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ACM, New York, NY, AFIPS '68 (Spring), 37–45. URL: http://doi.acm.org/10.1145/1468075.1468082. 73

CARPENTER, L. 1984. The A-Buffer, an antialiased hidden surface method. *SIGGRAPH Computer Graphics 18*, 3, 103–108. URL: http://doi.acm.org/10.1145/964965.808585. 87

CRASSIN, C., 2010. Icare3D Blog: Linked lists of fragment pages. Available at http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html. 87

GRIBBLE, C., NAVEROS, A., AND KERZNER, E. 2014. Multi-hit ray traversal. *Journal of Computer Graphics Techniques 3*, 1, 1–17. 73, 74, 75, 76, 78, 80, 86

INTEL CORPORATION, 2014. OSPRay: A ray tracing based rendering engine for high-fidelity visualization. Available at http://ospray.github.io. 80

KAJIYA, J. T. 1986. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, 143–150. URL: http://doi.acm.org/10.1145/15922.15902. 73

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, New York, NY, 137–143. URL: http://doi.acm.org/10.1145/2492045.2492060. 86

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, K., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGRUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2010) 29*, 4, 66:1–66:13. URL: http://doi.acm.org/10.1145/1778765.1778803. 73

WALD, I., WOOP, S., BENTHIN, C., JOHNSON, G. S., AND ERNST, M. 2014. Embree - a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (July), 143:1–143:8. URL: http://dl.acm.org/citation.cfm?id=2601199. 73

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349. URL: http://doi.acm.org/10.1145/358876.358882. 73

YANG, J. C., HENSLEY, J., GRÜN, H., AND THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum 29*, 4, 1297–1304. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2010.01725.x/abstract. 87

Index of Supplemental Materials

We provide algorithm listings for multi-hit ray traversal using both progressive insertion sort and post-traversal selection sort, as well as performance visualizations similar to those in Figure 1 for all eight scenes, in **`extra.pdf`** available online at http://jcgt.org/published/0004/04/04/extra.pdf.

We provide the OSPRay module for our Embree multi-hit implementations targeting CPUs and Xeon Phi coprocessors. The module implements naive multi-hit traversal using Embree's intersection-filter functions and can be configured to visualize efficiency metrics as heatmaps. The various sorting techniques and data layouts are selected via flags found at the top of the `ispc` renderer source file.

Similarly, we provide some of the important source files for our OptiX multi-hit implementation. We use an OptiX SDK example as the basis of our implementation, so we cannot distribute the entire project due to license restrictions. However, we provide source files implementing the key elements required to solve the multi-hit problem with OptiX.

The material is organized as follows:

- **`ospray_mh_module/`** contains code implementing the full OSPRay multi-hit module. To build and run this module, simply copy its contents to the **`modules`** directory within the OSPRay source tree.
  - **`CMakeLists.txt`** provides boilerplate build-system content for compiling OSPRay modules.
  - **`mhtk.dox`** provides Doxygen content.
  - **`multihit_kernel.ih`** provides data structure definitions for the multi-hit module.
  - **`xray_renderer.{h,cpp}`** provides boilerplate renderer code for creating the OSPRay multi-hit module.
  - **`xray_renderer.ispc`** implements our multi-hit techniques. As noted above, the various sorting techniques and data layouts are enabled via compile-time flags defined at the top of this file.
- **`optix_snippets/`** contains our OptiX implementation code fragments.
  - **`Flags.h`** contains the compile-time flags controlling which sorting techniques and data layouts are used at runtime, similar to the flags found at the top of `ospray_mh_module/xray_renderer.ispc`.
  - **`multihit.cu`** implements the OptiX any-hit program, similar to the intersection-filter implementation in `ospray_mh_module/xray_renderer.ispc`.
  - **`pinhole_camera_mh.cu`** implements ray generation and intersection, as well as hit-point post-processing (as required).
  - **`RayPayload.h`** provides data structure definitions for hit point and per-ray multi-hit data.

Additionally, the OSPRay implementation can be found on GitHub at:

https://github.com/jeffamstutz/jcgt_multihit_2015.git

Author Contact Information

Jefferson Amstutz
Intel Corporation
1300 South Mopac Expressway
Austin, TX 78746
jefferson.d.amstutz@intel.com
https://www.linkedin.com/pub/jefferson-amstutz/22/547/4ba/

Christiaan Gribble
Applied Technology Operation
SURVICE Engineering Company
6101 Penn Avenue
Pittsburgh, PA 15206
christiaan.gribble@survice.com
http://www.rtvtk.org/˜cgribble/

Johannes Günther
Intel Corporation
Dornacher Strasse 1
Munich, BY, D-8016
GERMANY
johannes.guenther@intel.com
http://www.johannes-guenther.net/

Ingo Wald
Intel Corporation
1300 South Mopac Expressway
Austin, TX 78746
ingo.wald@intel.com
http://www.sci.utah.edu/˜wald/

Jefferson Amstutz, Christiaan Gribble, Johannes Günther, Ingo Wald, An Evaluation of Multi-Hit Ray Traversal in a BVH, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 4, 72–90, 2015
http://jcgt.org/published/0004/04/03/

Received: 2015-08-10
Recommended: 2015-10-07 Corresponding Editor: Ulf Assarsson
Published: 2015-12-16 Editor-in-Chief: Marc Olano