Local Shading Coherence Extraction for SIMD-Efficient Path Tracing on CPUs

Attila T. Áfra   Carsten Benthin   Ingo Wald   Jacob Munkberg

Intel Corporation

Figure 1: *The* Mazda *scene (5.7M triangles, 76 materials) path traced using our stream-based shading coherence extraction method. For this scene, our coherent shading algorithm achieves 90% utilization of 8-wide SIMD and reduces the shading time by 2× compared to simple 8-wide packet shading. This results in a total speedup of 35%. For scenes with more complex shading, we have measured speedups up to 3×.*

Abstract

Accelerating ray traversal on data-parallel hardware architectures has received widespread attention over the last few years, but much less research has focused on efficient shading for ray tracing. This is unfortunate since shading for many applications is the single most time consuming operation. To maximize rendering performance, it is therefore crucial to effectively use the processor's wide vector units not only for the ray traversal step itself, but also during shading. This is non-trivial as incoherent ray distributions cause control flow divergence, making high SIMD utilization difficult to maintain. In this paper, we propose a local shading coherence extraction algorithm for CPU-based path tracing that enables efficient SIMD shading. Each core independently traces and sorts small streams of rays that fit into the on-chip cache hierarchy, allowing to extract coherent ray batches requiring similar shading operations, with a very low overhead. We show that operating on small independent ray streams instead of a large global stream is sufficient to achieve high SIMD utilization in shading (90% on average) for complex scenes, while avoiding unnecessary memory traffic and synchronization. For a set of scenes with many different materials, our approach reduces the shading time with 1.9–3.4× compared to simple structure-of-arrays (SoA) based packet shading. The total rendering speedup varies between 1.2–3×, which is also determined by the ratio of the traversal and shading times.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Production rendering for visual effects and animation has recently undergone a paradigm shift. Traditionally, most renderers used for feature films were rasterization-based, feed-forward pipelines (e.g., Pixar's RenderMan REYES architecture [CCC87]). However, with the constant demand for increased visual fidelity, most studios are

now using ray tracing based architectures, where accurate lighting simulations can be computed more efficiently and robustly.

In previous rasterization-based architectures, each primitive is tested for visibility and shaded in a feed-forward pipeline, and shaders are evaluated coherently over sample grids by design, either in screen space or object space. In a ray tracer, the points to be
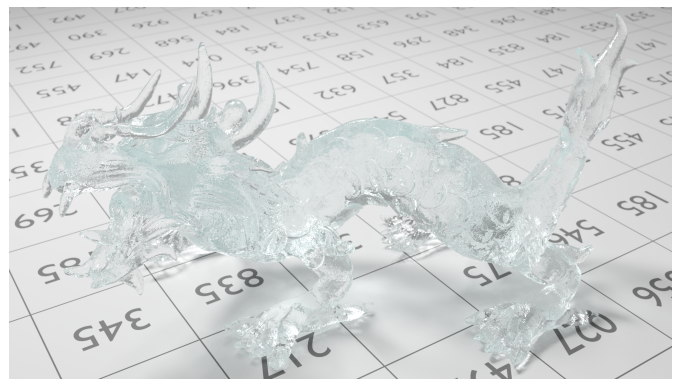
Art Deco (10.7M triangles, 111 materials)

Villa (37.7M triangles, 97 materials)

Conference (0.3M triangles, 36 materials)

Dragon (7.4M triangles, 5 materials)

Figure 2: *Four of the test scenes used for the performance evaluation of our shading coherence extraction method. Performance numbers for each scene are given in Table 2.*

shaded are not necessarily coherent. For example, in a path tracer (the most commonly used rendering method in modern production renderers), the shading step at each ray intersection point typically emits rays in pseudorandom directions over the hemisphere. This results in incoherent ray distributions, which in turn lead to incoherent shader invocations and data accesses after just one bounce.

On modern parallel architectures (either CPU or GPU), control flow divergence and incoherent data access can significantly degrade both ray traversal and, in particular, shading performance. Control flow divergence causes underutilization in the vector units during shading, and incoherent data access translates to non-cached reads from memory, disk, or even the network.

The growing importance of utilizing vector units and achieving better memory access patterns is well understood by the rendering community and has recently received much interest. However, most existing research focused on improving ray traversal performance, and as a result, many traversal algorithms for incoherent rays have been proposed for various architectures (see Section 2). Improving *shading* performance by extracting coherence, on the other hand, has received significantly less attention, despite the fact that in production path tracers usually more than half of the rendering time is spent in shading [ENSB13].

Previous approaches for improving shading coherence (see Section 2) operate on a single large, global stream, containing millions of rays, which is sorted (based on various criteria) to improve coherence during shading [LKA13, ENSB13]. Letting all processor cores work cooperatively on a global stream is a natural fit for high-throughput, high-latency architectures like GPUs, however, it is suboptimal for CPU architectures because they are more sensitive to memory accesses not served by the cache hierarchy.

In this paper, we propose a *local* shading coherence extraction algorithm optimized for modern many-core CPU architectures and vector instruction sets, which significantly improves the SIMD efficiency of shading for path tracing based renderers. To extract coherence, we trace small streams of rays on each processor thread in a breadth-first fashion and sort the ray hits by material ID before evaluating the shaders. In this sorting stage, we group the ray paths in each stream into coherent SIMD-sized batches that need to be processed with a single shader, avoiding code path divergence. The streams are independent from each other and are small enough (up to a few thousands rays) to fit into the cache hierarchy of the CPU. Also, they are always compact in the sense that no gaps are introduced due to terminating paths.

In contrast to previous approaches that work on a single large

stream, our local stream approach does not require additional synchronization between the different hardware threads and maximizes cache usage when operating on the ray path data of the local stream. Thus, the coherence extraction itself has a very small overhead, much smaller than that of global methods. Although larger streams enable potentially higher shading coherence, we demonstrate that smaller, per-thread streams are sufficient for achieving close-to-optimal coherence and thus high SIMD utilization for complex scenes with many different materials and shaders (see Figures 1 and 2).

We demonstrate the effectiveness of our algorithm by comparing its performance against standard single-ray tracing and *packet tracing*, which are the most widely used techniques in current production path tracers. In the remainder of the paper, *packet tracing* refers to structure-of-arrays (SoA) based SIMD shading of small (e.g., 8-wide) packets with single-ray traversal, and *stream tracing* refers to our coherent stream shading algorithm also with single-ray traversal. For all but one test scene which has low shading complexity, our stream tracing approach significantly outperforms both single-ray and packet tracing.

2. Previous Work

Most ray tracing based renderers today trace a single ray at a time, hence a lot of research has been devoted over the years to improve performance of single-ray based traversal algorithms [WBB08, DHK08, AL09, Áfr13]. However, exceeding the single-ray traversal performance requires traversal approaches which focus on extracting coherence from multiple rays that are traced together. Ray packet based traversal algorithms [WSBW01, WBS07, ORM08, BWW*12] attempt to improve traversal efficiency by doing depth-first traversal for a small number of rays in a SIMD fashion. They achieve notable speedups for mostly coherent ray distributions (e.g., primary rays, hard shadow rays) but suffer from divergence for incoherent workloads.

Breadth-first ray tracing [Han86, LMW90] is a more robust solution, which traverses entire ray generations together and extracts coherent subsets of rays by ray queuing [PKGH97, AK10], ray reordering/sorting [BWB08, GL10], or ray stream filtering [WGBK07, GR08, RGD09]. Recently, the focus shifted to traversal algorithms that combine the advantages of stream filtering and SIMD-optimized single-ray traversal [Tsa09, BAM14, FLPE15]. These algorithms can be more efficient than corresponding single-ray or ray packet techniques if sufficient coherence can be extracted out of the ray stream.

Apart from improving coherence for ray traversal, the shading phase suffers from the same issues of low vector utilization and incoherent memory accesses when shading a set of incoherent rays. The more incoherent the rays are, the higher the probability that many different shaders have to be evaluated within a SIMD batch. Researchers addressed this issue by extending the shading framework with (multiple) pre-shading filtering/sorting steps to increase execution coherence during shader evaluation.

In the context of GPU ray tracing, Wald [Wal11] and van Antwerpen [vA11] proposed techniques that compact streams of rays to avoid inactive threads due to terminated ray paths, but SIMD

divergence from executing different shaders was completely ignored. To address the issue of low shading efficiency when using multiple materials, Hoberock et al. [HLJH09] employed different stream compaction steps in a separate shading phase, improving vector utilization. Laine et al. [LKA13] showed that splitting up a monolithic GPU path tracing kernel into smaller specialized kernels combined with global ray queuing can significantly improve vector utilization and register usage, resulting in speedups up to $3\times$. The GPU-based *OptiX* ray tracing framework [PBD*10] also supports queuing and batching of tasks (including shading) to improve vector utilization, but it cannot gather rays from different hardware vectors.

The majority of CPU-based renderers trace a single ray at a time, which directly translates to shading single rays. On wide-vector CPU architectures (where the vector width is 8 or greater) this leads to severe underutilization of SIMD units. The *Embree* kernel framework [WWB*14] supports shading of ray packets where the packet width corresponds to the vector width of the underlying CPU architecture. However, with an increasing number of ray bounces, the SIMD utilization per shader evaluation for a given ray packet can quickly drop to a single active ray.

As the number of rays within a single packet is typically not enough to extract sufficient coherence for shading, researchers focused on larger ray streams instead. For CPU-based production rendering, Eisenacher et al. [ENSB13] proposed to use a large ray stream combined with sorting to improve data access coherence during shading. The main goal was maximizing both geometry and texture access coherence, which made a costly and complex texture cache implementation obsolete and enabled coherent geometry access. Even though data access coherence is improved by sorting the rays before shading, the shader evaluation is still done sequentially for a single ray at a time.

Concurrently to our work, RenderMan RIS [Pix16] is apparently using a local coherence extraction technique similar to ours in its shipping versions, as suggested by the public API documentation and example plugin source codes. However, no paper has been published on this approach yet, thus, many important algorithmic and implementation details such as sorting and vectorization strategies are not available, neither is any data on SIMD efficiency.

3. Coherence Extraction

Our shading coherence extraction method is based on tracing small streams of rays in a breadth-first manner, traversing and shading generations of rays together. These ray streams consist of up to a few thousand rays, and each rendering thread traces separate, independent streams. In the following, we assume that a single rendering thread is mapped to a single CPU hardware thread. Therefore, the streams are kept *local* with respect to the CPU thread and the local CPU caches, and no costly cross-core communication [SBH15] is required. This simplifies the coherence extraction algorithm and also minimizes the execution overhead on high-end CPUs with tens of threads.

Our framework implements a unidirectional path tracing integrator with next event estimation. On each surface interaction, the current path is extended with one ray, and a light is sampled with one

Figure 3: *The material IDs assigned to the primitives of the* Mazda *scene, visualized with different colors. Our method sorts the ray hits by material ID before shading, which is enough to achieve high SIMD utilization.*

Figure 4: *Overview of our path tracing pipeline. The algorithm can be divided into three groups of stages: traversal (extension ray and shadow ray intersection), shading (material evaluation, accumulation, and ray generation), and sorting.*

shadow ray. The shading system consists of materials, bidirectional scattering distribution functions (BSDFs), and shaders. Each material in the scene has a unique *material ID* (see Figure 3) and is associated with a shader. The material shaders, when given a surface point, construct a composite BSDF consisting of one or more layers, which is then evaluated and sampled by the integrator. Lights (e.g., area lights or environment maps) are treated as emissive materials and can be directly sampled. Multiple importance sampling (MIS) [VG95] is used to combine the BSDF and light samples.

3.1. Algorithm

The path state on each hardware thread consists of two ray streams: an *extension ray stream* for extending the paths, with one path segment at a time, and a *shadow ray stream* for direct light sampling. The path tracing algorithm is divided up into *stages*, each involving an iteration over certain ray stream values. An overview of our path tracing pipeline is shown in Figure 4.

Ray generation. First, we initialize the extension ray stream with primary rays generated from an image tile (e.g., 16×16 pixels). The rays could correspond to one or more samples per pixel.

Ray intersection. Then, we intersect all rays in the stream with the scene geometry. For this step, we can intersect the rays either with a traditional single-ray algorithm or with a faster stream-based approach [BAM14, FLPE15] that traverses the rays together in order to extract coherence. For all measurements shown in the paper, we rely on single-ray traversal to highlight the improvement on total rendering performance exclusively from improving shading efficiency. The resulting ray hit point data, including the material ID, is stored in the stream. If a ray does not hit any geometry, a special material ID (e.g., 0) is stored to indicate a ray miss.

Sorting. In the next stage, we extract shading coherence by sorting the indices of the rays in the stream by the material ID of the hit points. The ray data itself is not shuffled around during sorting. Thus, we produce an array of ray IDs where rays that must be evaluated with the same shader are referenced consecutively. Since the
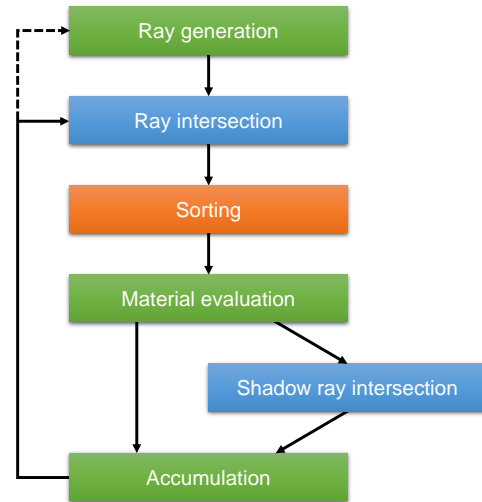
streams are local, we can use a simple sequential sorting algorithm to achieve this. We opted for the counting sort algorithm [Knu98] because the number of different material IDs is typically on the order of hundreds or, in the worst case, thousands. It can be implemented very efficiently and has $\mathcal{O}(n+k)$ complexity, where n is the stream size and k is the maximum material ID.

Material evaluation. Coherent material shading is performed by iterating over the sorted ray IDs and executing the shaders for batches of rays with the same material ID using SIMD operations. A shader is invoked for m rays in parallel at a time, where m is the SIMD width. Each instance of the shader is mapped to a different SIMD lane. If the size of a batch is a multiple of the SIMD width, the vectors passed to the shader are fully utilized, thus the shader is executed at maximum compute efficiency. However, the actual SIMD utilization can still be suboptimal due to control flow divergence in the shader itself.

The integrator can either choose to continue the path and generate an extension ray by sampling the BSDF of the material, or it can flag the path for termination. The integrator can also optionally generate a shadow ray by picking a light source, directly sampling it, and evaluating the BSDF for the generated ray direction. If there are different types of lights in the scene (e.g., both area and environment lights), light sampling could suffer from SIMD divergence. In order to avoid this, material evaluation and light sampling could be split into separate stages. We did not do this in our framework because light sampling is usually much less expensive than material evaluation, but this may not be the case in production rendering.

The extension ray, path throughput, path flags (termination and light sampling), and other variables (e.g., BSDF sample PDF, medium ID) that are computed by the integrator are appended to a new, *compact* stream, discarding the old stream. Double buffering is used for the storage of the input and output streams, switching the

buffers after each ray bounce. The shadow ray and the direct light contribution (assuming the shadow ray is not occluded) are added to the shadow ray stream, which is separate from the extension ray stream because not all materials might support shadow rays (e.g., perfectly specular materials). The direct light contribution will be added to the path radiance only in a later stage, after testing all shadow rays for occlusion.

For rays that did not hit anything, a special *miss* shader is executed, which in our implementation accumulates the pixel values corresponding to the terminated paths in the framebuffer. This shader can additionally return a specific background color or a color sample from an environment map.

Shadow ray intersection. After evaluating all materials over the stream, the shadow rays generated by the integrator are tested for occlusion. The results (either hit or miss for each ray) are written back to the stream, which will be used to compute the final direct light contributions in the next stage.

Accumulation. Finally, we iterate again over the extension and shadow rays streams, with a single loop, to accumulate the direct light contributions for the unoccluded rays in the path radiances, and to perform framebuffer accumulation for the paths flagged for termination.

Path regeneration. After the accumulation stage, we can either continue with the ray intersection stage or we can first generate more primary rays from the current tile if the number of rays in the stream drops below a threshold, due to terminated paths. This technique is known as *path regeneration* [NHD10]. Since the stream is compact, generating and appending new primary rays is trivial and allows us to operate close to the maximum SIMD efficiency.

3.2. Implementation

We have optimized our framework (both shading and ray traversal) for the AVX2 instruction set, which provides 8-wide SIMD instructions for 32-bit data types. The shaders are executed using structure-of-arrays (SoA) style SIMD processing, which is also how vectorizing compilers like the Intel® SPMD Program Compiler (ispc) [PM12] and modern GPU architectures parallelize scalar code over multiple vector unit lanes.

The streams are stored in a SoA memory layout, which is a natural fit for SIMD processing, and thus it is preferred over the simpler array-of-structures (AoS) layout. The SoA layout enables using efficient SIMD vector loads and stores for stages that operate on consecutive rays in a stream. Other stages that access rays in an arbitrary order, such as the material evaluation stage, use gather instructions to read values into vectors.

Appending values in a vector to the stream is done using a *pack-store* operation [Int16], which compacts the vector based on an active mask before storing it in memory. Although AVX2 does not have a pack-store instruction, it can be efficiently emulated with a vector permute instruction followed by a masked vector store instruction. Future instruction sets such as AVX-512 [Int16] will have native pack-store instructions, which will be potentially even faster.

Extension ray stream	Double buffered	Size (bytes)
Ray origin (xyz)	Y	3×4
Ray direction (xyz)	Y	3×4
Ray t$_{far}$	Y	4
Path flags, pixel ID, sample ID	Y	4
Path throughput (RGB)	Y	3×4
Path radiance (RGB)	Y	3×4
Medium ID	Y	4
Ray ID	N	4
Hit material ID	N	4
Hit primitive ID	N	4
Hit UV	N	2×4
BSDF sample PDF	N	4
BSDF sample type	N	4

Shadow ray stream	Double buffered	Size (bytes)
Ray origin (xyz)	N	3×4
Ray direction (xyz)	N	3×4
Ray t$_{far}$	N	4
Light sample radiance (RGB)	N	3×4

Table 1: *Variables in the extension ray and shadow ray streams. The streams are stored in SoA layout, thus each variable is stored in a separate array. Some arrays are double buffered. The total storage cost per path is 188 bytes. We rely on many paths per pixel and bilinear texture lookups for anti-aliased texture filtering, and thus do not include ray differentials. These would add an additional 48 bytes per path. For our default stream size of 2048, the path state occupies 376 KB per thread and can therefore be efficiently held in the CPU cache hierarchy.*

The opposite of the pack-store is the *load-unpack*, which is used in the accumulation stage to expand the shadow ray stream, inserting inactive elements when loading into SIMD vectors, so the indices of its elements match those of the extension ray stream. This way, both streams can be processed in the same loop, iterating over the streams in lock-step. We completely avoid using scatter operations, which are more expensive on current CPU architectures.

Table 1 shows the contents of the extension ray and shadow ray streams. As previously mentioned, the extension stream has to be double buffered because it is both an input and an output stream for the material evaluation stage. However, some variables in the stream are either only inputs or only outputs (e.g., ray IDs, hit data). To minimize the size of the path state, we employ double buffering only for those variable arrays that require it. This is illustrated in Figure 5, where we provide an example for the stream data flow when executing shaders.

The path state in our implementation requires 188 bytes of storage per path, which, for example, translates to a total size of 376 KB for streams of 2048 elements. In Section 4, we will demonstrate that 2048 is a good default choice for the stream size on current CPUs. The entire path state for a thread is allocated in a single chunk, instead of separately allocating each variable array of the streams, so a single pointer and an index are sufficient for addressing any value.
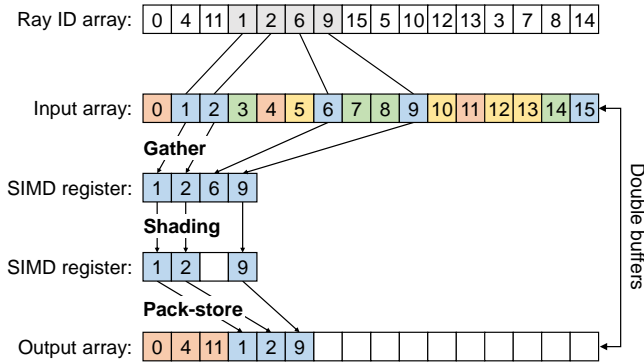
Figure 5: *Stream data flow for executing SIMD shaders. Here we use 4-wide SIMD for simplicity. The colors indicate different materials. First, values from the input variable (e.g., x coordinate of ray origin) array are gathered into a SIMD register using the sorted ray IDs as indices. This SIMD vector is then fed into the shader, which computes new values for a subset of the lanes and deactivates the rest (due to path termination). Finally, the active values are appended to the output array using a pack-store operation.*

4. Results

We compare our stream tracing algorithm to standard single-ray tracing with scalar shading and packet tracing with SoA-based SIMD shading. All algorithms use the same single-ray SIMD traversal kernel and are implemented in the same path tracing renderer having a scalar and a SIMD shading code path. The packet and stream tracers use identical SIMD shading code.

4.1. Test Setup

Path tracer. Our path tracer supports the most important BSDFs used in state-of-the-art production renderers. For diffuse surfaces, the Lambert and Oren-Nayar [ON94] models are used. Rough dielectrics and conductors are implemented with microfacet BSDFs based on the Smith microsurface model [Hei14] and the GGX normal distribution [WMLT07]. The microfacet BSDFs are importance sampled using the distribution of visible normals [HD14]. Shaders can model more complex materials (e.g., plastic, metallic paint) by combining multiple surface layers into a unified BSDF [WW07]. Each layer can feature separate shading normals, which can be either sampled from a normal map or generated procedurally. The scene can be illuminated with area lights, distant lights (e.g., the sun), ambient lights, and HDR environment maps.

The pseudorandom numbers for the samples are generated using the Sobol low-discrepancy sequence [JK08] and Cranley-Patterson rotation [CP76]. Russian roulette is used for path termination after 8 path segments, and the maximum number of path segments is limited to 48.

The acceleration structures used for ray intersections are high-quality 8-way branching BVHs [WBB08] constructed using both object and spatial splits [SFD09]. Since we are primarily interested in shading performance, we use the same vectorized single-ray traversal kernel [Áfr13] for single-ray, packet, and stream tracing.
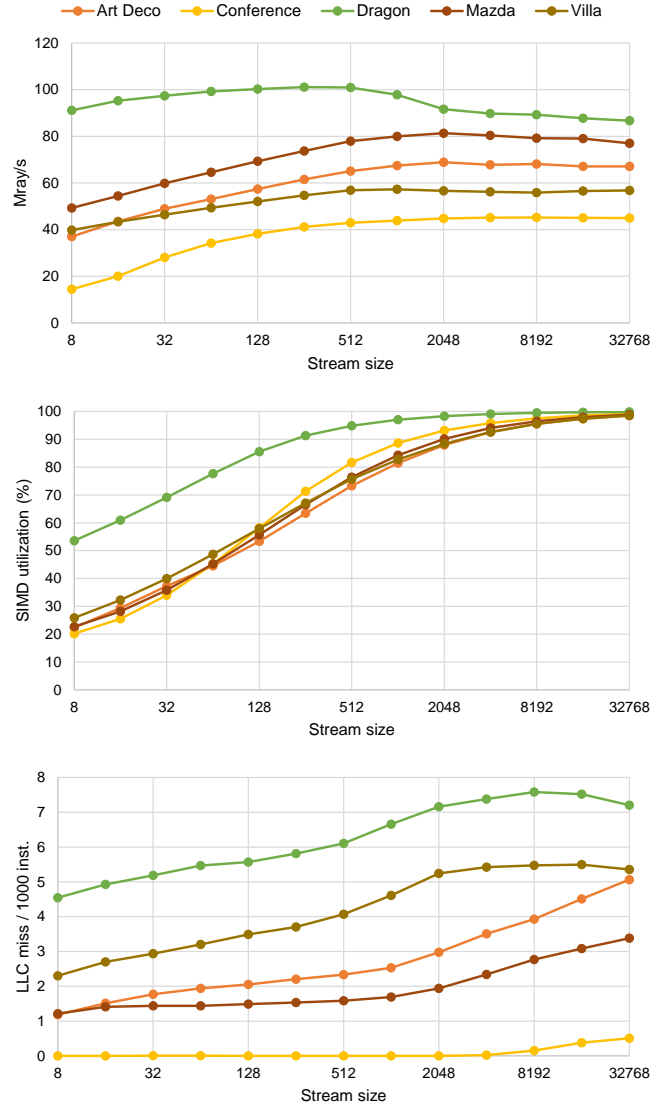

Figure 6: *Performance of our method without path regeneration in million rays per second (top) and shading SIMD utilization (middle) for 8-wide SIMD and different stream sizes. The number of LLC misses per 1000 instructions was also measured (bottom) using Intel® VTune™ Amplifier. The tile size was fixed at 16×16 pixels, and the number of samples per pixel (per stream) was varied between 1–128, depending on the stream size.*

Using a specialized stream traversal kernel could provide further speedups but analyzing this is beyond the scope of our paper.

The code, including the shaders, was written in C++ using AVX2 intrinsic functions, and it was compiled with the Intel® C++ Compiler 16.0.2 for Linux.

Hardware. Our test system is a dual-socket workstation with two Intel®Xeon®E5-2699 v3 CPUs based on the *Haswell* microarchitecture, each having 18 cores clocked at 2.3 GHz, 36 hardware threads, 8-wide SIMD, 256 KB L2 cache per core, and a 45 MB

Scene	Single	Packet			Stream				Stream w/ regeneration			
	Mray/s	Mray/s	SIMD	Speedup *Single*	Mray/s	SIMD	Speedup *Single*	*Packet*	Mray/s	SIMD	Speedup *Single*	*Packet*
Art Deco	41.7	47.0	22%	1.13×	68.6	88%	1.65×	1.46×	67.3	88%	1.61×	1.43×
Conference	14.0	15.2	20%	1.09×	44.6	93%	3.18×	2.93×	44.5	95%	3.18×	2.92×
Dragon	81.7	99.9	54%	1.22×	96.4	98%	1.18×	0.96×	92.0	99%	1.13×	0.92×
Mazda	55.4	60.5	23%	1.09×	81.6	90%	1.47×	1.35×	80.3	93%	1.45×	1.33×
Villa	42.6	46.3	26%	1.09×	57.6	88%	1.35×	1.24×	57.6	92%	1.35×	1.24×

Table 2: *Performance of the following path tracing techniques: single-ray tracing, packet tracing, and our stream tracing method without and with path regeneration. For the SIMD methods (packet and stream), we list the total performance in million rays per second (including ray traversal and shading), the SIMD utilization, and the speedup compared to single-ray and packet tracing. For all test cases we used a stream size of 2048 rays and a tile size of 16×16, and we rendered the images with 256 samples per pixel per frame. Thus, we traced a total of 65536 paths per tile.*

shared last-level cache (LLC). The system has quad-channel DDR4 memory clocked at 2133 MHz.

Scenes. We use five test scenes to evaluate the performance of our path tracer, which are depicted in Figures 1 and 2. Our assets and shaders are an attempt to approach the complexity used in visual effects, architectural visualization, and computer-aided design, but we like to point out that high-end visual effects productions may use significantly (1–2 orders of magnitude) more complex geometry, materials, and textures than presented here. Please refer to the image captions for the material and triangle counts.

Mazda, *Art Deco*, and *Villa* are large scenes with detailed geometry, many materials, high-resolution texture and normal maps (up to three layers), and realistic shaders. The most complex material in these is the car paint in the *Mazda* scene, which is a two-layer material having a microfacet conductor substrate with procedural flakes and a microfacet dielectric coating (see the closeup in Figure 1).

The *Conference* scene has simple geometry but has the highest shader complexity of all scenes. The floor is made of an expensive procedural marble material, which is evaluated using a 3D fractional Brownian motion (fBm) function with value noise. The noise function is implemented with hashing for the integer lattice points and cubic Catmull-Rom interpolation. The red chairs feature the procedural car paint material, and the table is made of rough glass. The rest of the materials are simple matte and glossy surfaces.

Finally, the *Dragon* scene consists of millions of triangles but has only five simple materials (rough glass and Lambertian surfaces). This test scene was added in order to evaluate shading coherence extraction where it has low potential for improving performance.

All scenes were rendered at a resolution of 3840×2160, and the image was broken up into tiles of 16×16. The number of samples per pixel per stream was varied depending on the stream size.

4.2. Measurements

Figure 6 shows the path tracing performance, the shading SIMD utilization (in the material evaluation stage), and the number of LLC misses for our method with respect to different stream sizes. For most scenes, the highest performance is achieved using a stream size of 1024–2048, for a SIMD utilization of about 90% (using 8-wide SIMD). There is a trade-off for using larger streams:
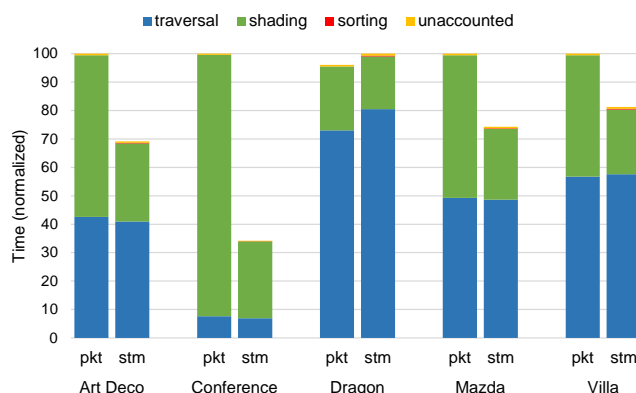

Figure 7: *Breakdown of the time spent on traversal, shading, and sorting for packet tracing (*pkt*) and our stream tracing method (*stm*). Note that sorting and the unaccounted operations have negligible costs. We have performed these measurements using Intel® VTune™ Amplifier.*

increasing the stream size improves SIMD utilization by a small amount but starts to hurt performance due to the higher number of cache misses (caused by a larger working set). We have measured the biggest performance drop for the *Dragon* scene (up to 14%), which has the lowest shading complexity, having only 5 materials. The rest of the scenes, with 36–111 materials, behave very similarly in terms of both rendering speed and shading efficiency.

Based on these measurements, we have opted for a default stream size of 2048 for our further tests, which means that the per-processor path state for 36 threads occupies about 30% (13 MB) of the shared LLC. Care must be taken not to saturate the LLC with the streams as it serves the traversal memory requests as well. Using streams larger than 16384 has practically no benefits since the SIMD utilization already exceeds 97% at this level. It is clear that using local streams is highly efficient, and global streams are not necessary for attaining almost perfectly coherent shader execution.

In Table 2, we have compared the performance of our method, both with and without path regeneration, against single-ray and packet tracing. For all scenes except the simple *Dragon* scene, stream tracing without path regeneration is the fastest approach. In those cases, it outperforms single-ray tracing by 1.35–3.18× and
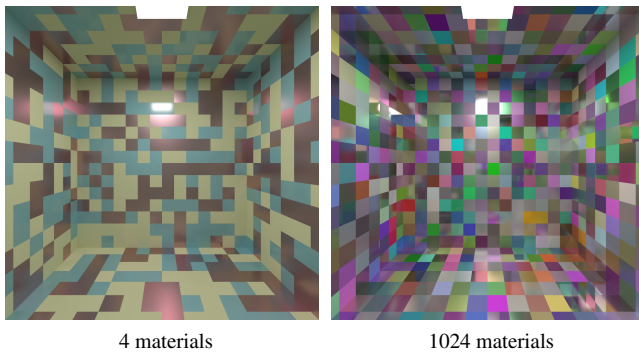
Figure 8: Tiled Box *scene with different number of randomly generated materials (with Oren-Nayar or GGX conductor substrates and up to two GGX coatings, without textures). The number of materials includes one emissive material used for the light source.*
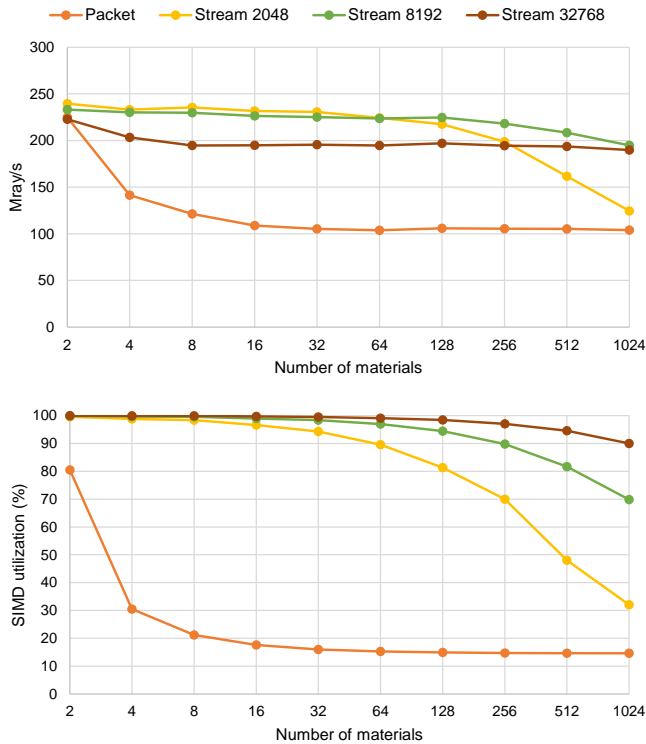

Figure 9: *Million rays per second and shading SIMD utilization for the* Tiled Box *scene (Figure 8) with varying number of materials, using packets and streams (with 2048, 8192, and 32768 rays).*

packet tracing by 1.24–2.93×. Also, it achieves significantly higher SIMD utilization for shading than packet tracing: while for packets the SIMD utilization is only 20–54%, for streams it is 88–98%. We see the highest speedup for the *Conference* scene, almost 3×, which has the most complex shaders. This number is very close to what Laine et al. [LKA13] measured for their *wavefront tracer* using a much larger global stream (1M rays) on a GPU with 32-wide vector units, for the same scene with similarly complex shaders.

Figure 10: *8-wide and 16-wide SIMD utilization for shading the* Mazda *scene using our method (without path regeneration). With 2048 rays per stream, doubling the SIMD width causes only 10% reduction in SIMD utilization.*

Using path regeneration was not beneficial for our test scenes, being slightly slower than the basic method. This can be explained by the overhead of handling rays from different generations in the same stream, and the negligible improvement in SIMD utilization, up to only 4%. However, turning on path regeneration could be worthwhile for outdoor scenes similar to *Villa*, where paths terminate at a higher rate, and there are several complex materials.

In Figure 7, we show measurements of the time spent on ray traversal, shading, and sorting by our method and packet tracing. From these results, it is apparent that our method significantly decreases shading time for all scenes by 1.2–3.4×. The ray traversal times are, in most cases, roughly the same for the two methods, which suggests that reordering ray intersections has very little effect on the overall performance. Therefore, the rendering speedups can be entirely attributed to the shading coherence extraction. The biggest difference in traversal time is for *Dragon*, where stream tracing causes a 10% traversal slowdown due to more cache misses, but shading is still faster. The chart also reveals that sorting has almost zero overhead: only up to 0.3% of the total rendering time.

The charts in Figure 9 illustrate the scaling of path tracing performance and shading SIMD utilization with the number of materials in a procedurally generated scene (see Figure 8). For 1024 materials, 2048 rays per stream are not enough to extract sufficient shading coherence. By increasing the stream size to 8192, stream tracing outperforms packet tracing by 1.87×. Streams of 32768 have smaller speedups in the tested range due to the higher cache overhead but could perform better for more complex scenes.

So far we have analyzed the coherence extraction efficiency of our method only for 8-wide SIMD. Future CPUs are switching to even wider SIMD, with the AVX-512 instruction set, which will enable 16-wide vector execution. To predict the efficiency of our method on such architectures, in Figure 10 we show the shading SIMD utilization using 16-wide SIMD for different stream sizes. As expected, the utilization is lower than using 8-wide SIMD, but the difference is small for larger streams. For example, the utilization for streams of 2048 rays is already at 80%, which can be increased to almost 90% by doubling the number of rays to 4096.

This indicates that our algorithm will scale well to wider vector architectures.

5. Discussion

Shading. From the results we can conclude that our algorithm is much more efficient than single-ray and packet tracing for scenes with moderate to high shading complexity. In such cases, packet tracing with SIMD shading is only marginally (less than 14%) faster than single-ray tracing with scalar shading, which is considerably easier to implement.

In contrast, our method makes much better use of the increasingly wider vector units of recent CPUs, which could justify implementing a SIMD shading system in a path tracing renderer using, e.g., ispc. The industry, however, is moving toward the standardized adoption of Open Shading Language (OSL) [Gri16], which currently does not have a compiler supporting SoA-based vectorization, making the transition to SIMD shading in production renderers difficult in practice. Nevertheless, OSL would be the ideal framework for implementing such a vectorized shading system.

Since our coherence extraction approach is local, it also scales well to more, potentially hundreds of CPU cores, and to bigger on-chip caches. However, for simple workloads where the number of shaders and their complexity is low, packet tracing remains a viable alternative.

Sorting. In our framework, we assign materials to surfaces using per-primitive material IDs but other shading systems could take a different approach. Although we rely on sorting material IDs to extract coherence, our method allows for other sorting criteria as well. For example, one could define only a single material which instantiates different BSDF components (BxDFs) depending on parameters fetched from textures. Thus, there would be no concept of material ID in this shading framework. Nevertheless, coherent shading could be implemented by first instantiating all BxDFs in a separate stage and then sorting by BxDF ID. The downside of this approach is that storing the instantiated BxDFs for the entire stream could substantially increase the memory requirements.

To achieve even higher performance, additional sorting steps could be introduced. After sorting by material ID, the ray hits could be further sorted by the IDs of the intersected primitives to improve geometry access coherence, reducing the amount of cache misses. Texture lookup coherence could be improved as well by sorting by the texture coordinates of the hit points. For these steps, instead of using counting sort, other sorting algorithms like radix sort would be more appropriate. Apart from temporarily storing the keys, sorting by primitives and textures would not require additional memory. Another, already mentioned, optimization would be to perform light sampling in a separate stage after sorting by light ID.

Limitations. Our framework restricts the shaders to spawn only one extension ray and one shadow ray per invocation. Shooting more than one shadow ray could be enabled by simply increasing the size of the shadow ray stream or by adding more streams. However, recursively spawning multiple extension rays would be more difficult to implement because the streams are fixed-sized and they may not have enough room to store the additional paths.

The integrator in our framework is based on unidirectional path tracing, which is often used in production renderers but in some difficult cases produces excessive amounts of noise. Extending our approach to more robust light transport algorithms like bidirectional path tracing [LW93, VG94] is a promising research direction.

Another limitation is that the control flow divergence inside the shaders is not alleviated. Therefore, sorting by material or shader ID does not work well if the shaders are so complex that their execution quickly degrades to a single SIMD lane.

6. Conclusion

We have proposed a local shading coherence extraction algorithm for CPU path tracing, which significantly improves SIMD utilization and thus performance over traditional single-ray and packet based approaches by up to 3×. We have shown that small local streams of rays map well to the CPU's cache hierarchy and sorting these streams enables nearly fully coherent shader execution with very low overhead. Our method efficiently handles complex scenes with hundreds of different materials and shaders, while being well suited for future wider-SIMD architectures. The measurements suggest that the method could perform even better with actual production shaders, which are likely to be much more expensive than ours. We are planning to release our framework as open source for the direct benefit of the rendering research community.

References

[Áfr13] ÁFRA A. T.: *Faster Incoherent Ray Traversal Using 8-Wide AVX Instructions*. Tech. rep., Babeş-Bolyai University, 2013. 3, 6

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG '10, Eurographics Association, pp. 113–122. 3

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 145–149. 3

[BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic ray stream traversal. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 33*, 4 (2014), 151:1–151:9. 3, 4

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive ray packet reordering. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008* (2008), IEEE Computer Society, pp. 131–138. 3

[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 (2012), 1438–1448. 3

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. *Computer Graphics 21*, 4 (1987), 95–102. 1

[CP76] CRANLEY R., PATTERSON T. N. L.: Randomization of number theoretic methods for multiple integration. *SIAM Journal on Numerical Analysis 13*, 6 (1976), 904–914. 6

[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. 3

[ENSB13] EISENACHER C., NICHOLS G., SELLE A., BURLEY B.: Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering* (2013), EGSR '13, Eurographics Association, pp. 125–132. 2, 3

[FLPE15] FUETTERLING V., LOJEWSKI C., PFREUNDT F.-J., EBERT A.: Efficient ray tracing kernels for modern CPU architectures. *Journal of Computer Graphics Techniques (JCGT) 4*, 5 (2015), 90–111. 3, 4

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. 3

[GR08] GRIBBLE C. P., RAMANI K.: Coherent ray tracing via stream filtering. In *2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (2008), pp. 59–66. 3

[Gri16] GRITZ L.: Open Shading Language 1.7 – Language Specification, Jan. 2016. 9

[Han86] HANRAHAN P.: Using caching and breadth-first search to speed up ray-tracing. In *Proceedings of Graphics Interface and Vision Interface '86* (1986), GI '86, Canadian Man-Computer Communications Society, pp. 56–61. 3

[HD14] HEITZ E., D'EON E.: Importance sampling microfacet-based BSDFs using the distribution of visible normals. *Computer Graphics Forum 33*, 4 (2014), 103–112. 6

[Hei14] HEITZ E.: Understanding the masking-shadowing function in microfacet-based BRDFs. *Journal of Computer Graphics Techniques (JCGT) 3*, 2 (2014), 48–107. 6

[HLJH09] HOBEROCK J., LU V., JIA Y., HART J. C.: Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 173–180. 3

[Int16] INTEL CORPORATION: *Intel® Architecture Instruction Set Extensions Programming Reference*, 2016. Reference number: 319433-024. 5

[JK08] JOE S., KUO F. Y.: Constructing Sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing 30*, 5 (2008), 2635–2654. 6

[Knu98] KNUTH D. E.: *The Art of Computer Programming, Sorting and Searching*, 2 ed., vol. 3. Addison-Wesley, 1998. 4

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), HPG '13, ACM, pp. 137–143. 2, 3, 8

[LMW90] LAMPARTER B., MUELLER H., WINCKLER J.: *The Ray-z-Buffer – An Approach for Ray Tracing Arbitrarily Large Scenes*. Tech. rep., 1990. 3

[LW93] LAFORTUNE E. P., WILLEMS Y. D.: Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)* (1993), pp. 145–153. 9

[NHD10] NOVÁK J., HAVRAN V., DACHSBACHER C.: Path regeneration for interactive path tracing. In *Eurographics 2010 – Short Papers* (2010), Eurographics Association, pp. 61–64. 5

[ON94] OREN M., NAYAR S. K.: Generalization of Lambert's reflectance model. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques* (1994), SIGGRAPH '94, ACM, pp. 239–246. 6

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time Whitted ray tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008* (2008), pp. 41–48. 3

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 29*, 4 (2010), 66:1–66:13. 3

[Pix16] PIXAR: RenderMan Documentation – RIS Developers' Guide. https://renderman.pixar.com/resources/current/RenderMan/risDevGuide.html, May 2016. 3

[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* (1997), SIGGRAPH '97, ACM, pp. 101–108. 3

[PM12] PHARR M., MARK W. R.: ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar)* (2012), pp. 1–13. 5

[RGD09] RAMANI K., GRIBBLE C. P., DAVIS A.: StreamRay: A stream filtering architecture for coherent ray tracing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ASPLOS XIV, ACM, pp. 325–336. 3

[SBH15] SCHWEIZER H., BESTA M., HOEFLER T.: Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), pp. 445–456. 5

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 7–13. 6

[Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-BVH ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 151–158. 3

[vA11] VAN ANTWERPEN D.: Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), HPG '11, ACM, pp. 41–50. 3

[VG94] VEACH E., GUIBAS L.: Bidirectional estimators for light transport. In *Proceedings of Eurographics Rendering Workshop* (1994), pp. 147–162. 9

[VG95] VEACH E., GUIBAS L. J.: Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques* (1995), SIGGRAPH '95, ACM, pp. 419–428. 4

[Wal11] WALD I.: Active thread compaction for GPU path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), HPG '11, ACM, pp. 51–58. 3

[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. 3, 6

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007). 3

[WGBK07] WALD I., GRIBBLE C. P., BOULOS S., KENSLER A.: *SIMD Ray Stream Tracing – SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. Rep. UUSCI-2007-012, SCI Institute, University of Utah, 2007. 3

[WMLT07] WALTER B., MARSCHNER S. R., LI H., TORRANCE K. E.: Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (2007), EGSR'07, Eurographics Association, pp. 195–206. 6

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–165. 3

[WW07] WEIDLICH A., WILKIE A.: Arbitrarily layered micro-facet surfaces. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia* (2007), GRAPHITE '07, ACM, pp. 171–178. 6

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (2014), 143:1–143:8. 3