Vectorized Production Path Tracing

Mark Lee
DreamWorks Animation

Brian Green
DreamWorks Animation

Feng Xie
DreamWorks Animation

Eric Tabellion
DreamWorks Animation

Figure 1: *This scene of Poppy singing is from the opening sequence of the movie "Trolls". We have rendered it here with MoonRay, our vectorized production path tracing system. The scene uses over 7 GB of geometry with 9.1 million curves, and an average of 20 control vertices per curve; the characters have complex shading networks and use around 1.3 GB of unique textures. The SIMD utilization provided a* 5× *speedup for the shading and a* 3× *speedup for the path integration portions of the frame.*

ABSTRACT

This paper presents *MoonRay*, a high performance production rendering architecture using Monte Carlo path tracing developed at *DreamWorks Animation*. *MoonRay* is the first production path tracer, to our knowledge, designed to fully leverage Single Instruction/Multiple Data (SIMD) vector units throughout. To achieve high SIMD efficiency, we employ Embree for tracing rays and vectorize the remaining compute intensive components of the renderer: the integrator, the shading system and shaders, and the texturing engine. Queuing is used to help keep all vector lanes full and improve data coherency. We use the ISPC programming language [Intel 2011; Pharr and Mark 2012] to achieve improved performance across SSE, AVX/AVX2 and AVX512 instruction sets. Our system includes two functionally equivalent uni-directional CPU path tracing implementations: a C++ scalar depth-first version and an ISPC vectorized breadth-first wavefront version. Using side by side performance comparisons on complex production scenes and assets we show our vectorized architecture, running on AVX2, delivers between a 1.3× to 2.3× speed-up in overall render time, and up to 3×, 6×, and 4×, speed-ups within the integration, shading, and texturing components, respectively.

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

CCS CONCEPTS

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

KEYWORDS

Computer Graphics, Monte Carlo, Path Tracing, Production Rendering, Vectorization

ACM Reference format:
Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 11 pages.
https://doi.org/10.1145/3105762.3105768

1 INTRODUCTION

Over the past decade, research in high performance path tracing has greatly reduced the cost per ray and the time to build efficient acceleration structures [Ernst and Greiner 2008; Parker et al. 2010;

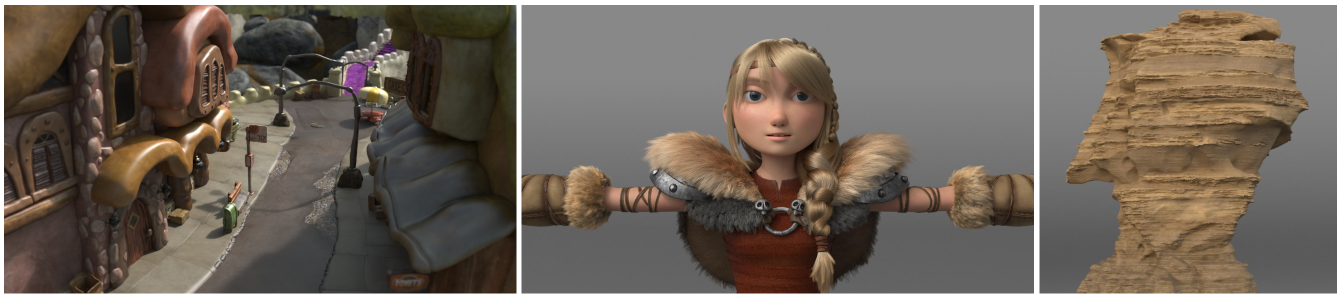

Figure 2: *"Trolls" environment Bergen Town, Astrid and an environment set piece called Hotspur from the movie "How to Train Your Dragon 2", rendered with vectorized MoonRay. Bergen Town uses over 360 MB of geometry mostly consists of subdivision meshes and 1.1 GB of textures; Astrid uses over 1.67 GB of geometry and 1.34 GB of textures; Hotspur uses 1.38 GB of geometry and 11.1 GB of textures. Astrid's hair alone has over 100K curves with an average of 45 control vertices per curve. There are over 3.5 million curves for Astrid's entire setup including hood fur, arm band fur, fur on her dress and soft skin fuzz, eyebrows and eyelashes. Vectorized MoonRay delivers a* 1.6× *to* 2.3× *speed up on total render time for these 3 scenes, with performance gains in shading, texturing and integration ranging from* 2× *to over* 4×.

Wald 2007; Wald et al. 2014]. During the same period, large improvements in hardware have vastly increased the number of CPU cores and the amount of memory available on commodity hardware. This overall increase in computational power has enabled production rendering systems to move away from the rasterization-based Reyes architecture [Cook et al. 1987], and evolve towards physically-based path tracing algorithms, the most commonly used approach being uni-directional path tracing [Kajiya 1986] with next event prediction [Pharr et al. 2016].

Path tracing is an inherently parallel algorithm and path tracing renderers have been able to exploit the growing number of CPU cores with relative ease. Each of these cores however devotes a non-trivial on-die area to supporting the vector (SIMD) processing of data. This should allow significantly higher throughput in theory, orthogonal to gains from multi-threading. However, no production renderer to date has been designed to exploit vector hardware throughout the whole system. So although path tracing itself is trivially parallelizable, unfortunately, it is not trivially vectorizable. In the context of a CPU based production renderer, the quest for full vectorization raises many questions... How do we access the vector hardware effectively? How do we map the path tracing algorithm into this domain whilst maintaining the flexibility required for production? How do we gather batches of work to keep the vector hardware busy? How do we keep memory accesses coherent and avoid scatters and gathers? How do we minimize vector code flow divergence? And if we do satisfactorily solve these problems, the question then becomes: *do the potential performance benefits gained outweigh the extra work required to harness the vector hardware?*

The main contribution of this paper is an attempt to shed light on these questions. In doing so, we introduce *MoonRay*, which to our knowledge, is the first full featured film production renderer designed to leverage SIMD vectorization throughout the whole system. We give an overview of the scalar and vectorized variants of *MoonRay* in Section 3.

In addition to being fully vectorized, *MoonRay* uses various optimization techniques commonly used in the games industry to achieve additional speedups. These include applying Data Oriented Design principles [Acton 2014] and a somewhat constrained approach to thread locking and memory allocation. It's worth clarifying that these optimizations are orthogonal to any vectorization speedups, and so benefit the both scalar and vectorized variants of the renderer. We elaborate on these details in Section 4.

In Section 5 we describe the underlying building blocks used to facilitate vectorized processing, in particular our approach to queuing and sorting the rays/samples, and the array-of-structures (AOS) to structure-of-arrays (SOA) transposition system we use to map each of these onto its own dedicated vector lane.

Embree [Wald et al. 2014] is employed for all ray intersections, which is already well vectorized internally. We have identified 3 additional components where a significant percentage of render time is spent, and are amenable to vectorization: *shading, texturing, and integration.* These are the areas we've specifically focused on, such that a single ray or sample is mapped to a single vector lane throughout. In Section 6 we cover the design and implementation specifics of each of these.

Finally, in Section 7 we provide detailed performance comparisons between our scalar and vectorized implementations of the renderer.

2 PREVIOUS WORK

Much essential work has been devoted in the past decade to accelerating ray tracing [Ernst and Greiner 2008], to handling curve primitive ray traversal and intersection [Nakamaru and Ohno 2002; Woop et al. 2014] and to quickly building good quality acceleration structures [Wald 2007]. Many papers also focused on accelerating the tracing of incoherent rays [Aila and Karras 2010], either via SIMD single ray traversal [Wald et al. 2008], packet tracing [Benthin et al. 2012; Garanzha and Loop 2010] or ray stream tracing [Hoberock et al. 2009; Tsakok 2009]. The OptiX ray tracing engine [Parker et al. 2010] provides a high performance ray tracing API for GPU architecture, whilst *MoonRay* uses Embree [Wald et al. 2014], which has state of the art ray tracing kernels with good SIMD utilization for single ray, ray packets and ray stream tracing computation on the CPU. As a result, our performance focus on *MoonRay* is

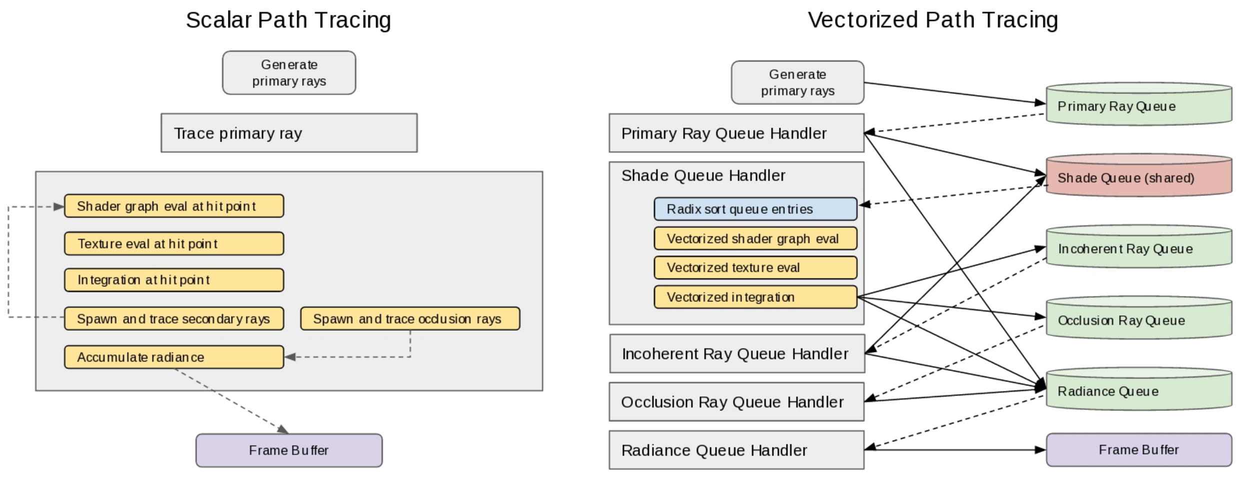

Figure 3: *Conventional path tracing on the left vs. vectorized path tracing on the right. The various queues in the vectorized implementation can be seen on the above right. All queues with the exception of the shade queue are thread local and allocated per thread. Each queue has an associated “handler”, represented by dashed lines to gray boxes, which process the entries in a queue after it fills up. Each handler can add further entries to other queues, as indicated by the solid black lines. All of the shading, texturing, and integration takes place inside the of the Shade Queue Handler.*

on accelerating all the other components needed in a production path tracing system, using coherent rendering and vectorization techniques.

Coherent rendering for ray tracing was introduced by Hanrahan [Hanrahan 1986], and further developed by Pharr et al. [Pharr et al. 1997]. It is a necessary step in attempting to achieve good memory locality of reference and enable optimizations relying on caching coherence, among others. To this end, there has been much work in hybrid rendering systems [Pantaleoni et al. 2010] that rely on spatial sorting, level of detail (LOD) selection and streaming in order to efficiently pre-compute spherical occlusion of out-of-core scenes on the GPU. Similarly, Kontkanen et al. [Kontkanen et al. 2011] performs clustering of spatial subdivision data structure nodes into coherent sub-trees, to accelerate the out-of-core traversal algorithm used in computing point based global illumination. More recently, Disney’s Hyperion [Eisenacher et al. 2013] demonstrated good geometry and texturing coherence via ray and hit point sorting, however their work does not exploit data parallelism or vectorization.

Laine et al. [Laine et al. 2013] describe a Single Instruction/Multiple Threads (SIMT) wavefront path tracing approach on the GPU, in order to reduce the divergence and register pressure encountered when naively executing large path tracing and shading kernels. Their system shows great promise when executing vectorized complex material shaders, but lacks many abilities of a production rendering system. Iray [Keller et al. 2017], which is built on top of OptiX, demonstrates what a more fully featured path tracer might look like in the GPU domain. In the work perhaps most closely related to ours, Áfra et al. [Áfra et al. 2016] study the impact of hit point sorting and ray batch sizing to reduce CPU SIMD divergence in manually vectorized shaders. Their system achieves high SIMD utilization similar to ours while rendering complex scenes. Our system extends this general type of approach to the whole ray-tracing pipeline and also supports a number of additional features which are essential for production rendering such as path splitting with Russian Roulette [Vorba and Křivánek 2016], out-of-core mip-mapped lazy texture access based on ray differentials [Gritz 2007], and arbitrary shader graphs assembled at runtime.

3 OVERVIEW

MoonRay is a production renderer that supports subdivision surfaces, polygon meshes and curve primitives. It can coherently access very large amounts of out-of-core texture data, using UDIM UV mapping [Seymour 2014] and mip-mapping driven by ray differentials tracked along each path. *MoonRay* has a complete procedural plugin API for rendering application, geometry procedural and surface shader development. We have used these APIs to develop fur/hair geometry procedurals, Alembic [Imageworks 2017] loaders and a complete set of lights, materials and texture shaders for feature animation.

Image generation in *MoonRay* is separated into two distinct phases which we will refer to in the remainder of this paper:

(1) **The preparation phase** is where we initially load the scene description, load and generate geometry assets by running procedural plug-ins, build the ray tracing acceleration structures and setup other data structures required for rendering.
(2) **The rendering phase** is where we compute pixel values by tracing paths, running shaders and executing Monte Carlo sampling and integration calculations at every ray hit along the paths. This is the compute-intensive phase, which is the focus of our vectorization architecture.

Conceptually, the rendering phase proceeds as follows. The image is partitioned into small square pixel buckets from which primary rays leaving the camera are computed and traced through the scene. Ray hits trigger the execution of a pre-assigned surface shader. The latter returns a Bsdf closure to the integrator, which contains a list of Bsdf lobes describing an arbitrarily complex surface or curve reflectance model that can be importance sampled. A pre-assigned light-set which can also be importance sampled is looked up, describing the full list of lights affecting the given ray hit.

The integrator then performs sampling decisions and calculations. We build upon robust uni-directional path tracing techniques using next event estimation of direct illumination and multiple importance sampling (MIS), combined with path splitting and Russian Roulette [Veach and Guibas 1995]. In our implementation, paths are split into sub-paths based on explicit user control and the paths with low importance are culled in a statistically unbiased way. Samples that pass the culling are traced further into the scene, either for testing occlusion or for continuing paths leading to path tracing recursion. Path splitting and Russian Roulette work hand-in-hand to semi-automatically adjust the sampling density when estimating parts of the high dimensional integral in various areas of the scene. This typically leads to a sizeable sample culling rate and consequently an increased percentage of time spent in the integrator.

MoonRay contains two fully functional implementations of the aforementioned path tracing algorithm as illustrated in Figure 3. The scalar code-path uses parallel processing of pixel buckets and each thread traces paths depth-first all the way with explicit recursion, before the next camera sample is processed. The vectorized code-path, on the other hand, processes batches of samples in wavefronts in breadth-first order, using multiple queues to track ray state. Queue entries are processed in parallel, which leads to adding entries into down-stream queues. To avoid dependencies between queues, the vectorized breadth-first implementation is a fully feed-forward pipeline and radiance contributions are decomposed as described in Pharr et al. [Pharr et al. 1997]. ISPC [Intel 2011; Pharr and Mark 2012] is a C-like language from Intel which makes it easier to write code targeting SIMD hardware. It hides the details of the underlying vector instruction set and internally generates the necessary code to handle control flow masking. Both the scalar and vectorized code paths execute the same overall computations and produce identical results. These side-by-side implementations prove essential to accurately compare the performance implications from different architectural decisions related to vectorization.

4 ARCHITECTURE AND SYSTEM DESIGN

The general philosophy underlying *MoonRay*'s implementation is to maximize the usage of available hardware resources, by keeping all vector lanes of all cores busy all the time with meaningful work. It reflects a devotion to raw performance, without sacrificing functionality or usability. This section describes some of the implementation choices which resulted from this mindset.

4.1 Data Oriented Design

Data oriented design (DOD) is a term which has been gaining popularity in recent years [Acton 2014]. Whereas an object oriented approach might focus on how to decompose the problem in terms of high level abstractions, a data oriented approach looks at how the problem can be efficiently mapped to the underlying hardware. Approaching the ray tracing problem through a data oriented lens informed many of our architectural decisions. Common properties of the platforms we target (Xeon and Xeon Phi class CPUs) include multiple cores, wide vector execution (SIMD) capabilities, and relatively long memory latencies. Under the constraints of a mismatch between high speed compute and slow memory access, DOD principles advocate focusing on how data is accessed and transferred. Consequently, we strive for careful data structure layout and controlled memory access patterns, avoiding randomly accessing memory where possible.

Another key tenet of DOD is *where there is one, there is usually more than one*, or more plainly, we should work in batches where possible. This is a common theme throughout the system implementation and has a profound impact on the overall architecture and the API choices we made.

4.2 Threading Considerations

Our approach to multi-threading during the preparation phase is to use Intel's Threading Building Blocks library [Intel 2010] (TBB), which gives us a convenient way to express task based parallelism and nested parallel loops.

For the rendering phase, we strive to achieve linear scalability with respect to core count. One of the primary causes preventing each core from running at 100% is lock contention. Therefore, aiming for 100% core utilization would imply getting rid of all potentially contentious locks. Although perhaps not entirely avoidable in practice, this serves as a valuable guiding principle.

Avoiding contentious locks has a couple of big implications. First, we aim to avoid heap allocations since they potentially cause global locks. From a development point of view, this led to curbing the use of Standard Template Library (STL) containers in the codebase unless we can ensure they are immutable in the rendering phase. By confining allocations to thread-local arenas [Pharr et al. 2016] or pre-allocated thread-friendly memory pools, we are able to avoid practically all stalls related to memory allocation.

Next, we strive to minimize thread-to-thread communication. Fortunately, the nature of path tracing allows us to treat each thread as an independent worker, so using TBB task based parallelism would be overkill here. Instead each thread simply runs in its own loop, pulling batches of primary rays from a shared work queue. To further minimize thread communication, we make heavy use of thread local storage (TLS) objects. These are structures which are only accessible by a single thread and so can be read from and written to freely without any locks. Typically, each thread will only reference read-only shared data, and its dedicated TLS object. TLS objects are assigned to each thread at the start of the render loop and passed down via the stack such that any function which needs access gets it passed in as a parameter.

5 QUEUING AND SORTING

The core idea underlying the vectorization performance improvement is that instead of processing a single work element at a time (be it a ray or shading sample), we can potentially process 4, 8, or 16 work elements simultaneously. This is accomplished by mapping each work element to its own dedicated vector lane, 4 for SSE, 8 for AVX/AVX2, or 16 for AVX-512. There are some fundamental hurdles to this style of processing however. First we need to be able to gather enough work elements together so all vector lanes are filled. This amount of work is not readily available when using depth-first path tracing, and so prompts the move to a breadth-first approach [Eisenacher et al. 2013; Hanrahan 1986; Pharr et al. 1997], as is more typical in GPU path tracing implementations [Parker et al. 2010]. Queuing is the mechanism we use to gather batches of work for vector processing.

Once we have enough work to fill the vector lanes, we want to sort these entries to maximize memory access coherence: when we fetch data from main memory we want to maximize its use before its subsequent eviction. Additionally, SIMD divergence in the code path as we process work elements is another hurdle which must be countered by actively ensuring the various inputs are relatively coherent. The queuing and sorting of data helps here also.

After sorting, we convert queue entries from array-of-structures (AOS) format to structure-of-arrays (SOA) format to facilitate this separate work element per lane model of execution whilst minimizing costly scatter and gather memory access operations.

5.1 Queuing

A queue in this context is not first-in-first-out (FIFO) but rather an area where we can gather units of work for later processing. Each queue has an associated function callback called a handler which is responsible for processing the rays or samples queued up (which we'll generically refer to as queue entries). We use the term "flush" to denote the following operations on a queue:

(1) The entries are copied into a temporary arena allocated buffer and the queue is restored to an empty or near empty state (the number of entries actually removed is a multiple of the vector lane width to maximize SIMD utilization). Other threads can continue adding entries at this point.
(2) The copied entries are sorted as described in Section 5.2.
(3) The sorted results are sent to the associated queue handler for processing.

All queues are pre-allocated before rendering starts to a known but configurable size. To minimize thread contention, all queues are thread local, except for shade queues for reasons described below.

Only rarely do we add single entries at a time to queues. To amortize queuing costs, we build up batches of entries and add them to the destination queue with a single call. This general approach applies to all queues in the system.

A queue is flushed *when a thread adding entries causes that queue to exceed its pre-configured maximum size.* That thread then becomes responsible for flushing the queue there and then before proceeding. It's possible that calling the handler to process queued entries could cause a different queue in the system to fill up, in turn causing its handler to be invoked, and so on. We can think of this chain in terms of a stack. By applying this rule, an autonomous scheduling falls out naturally and no central queue scheduler is required.

If a thread finds itself with a fully unwound stack, meaning it has no further work left to do, it asks a shared work queue for the next batch of primary rays to process. As the frame nears completion and there are no more primary rays left to process, each thread will enter a queue draining phase, where it manually flushes queues even though they may be only partially filled. Load balancing during this phase is achieved by having each thread flush its local queues first before moving onto flushing the shared shader queues, which may cause more entries to be inserted into local queues. This cycle is repeated until all queues are empty at which time the frame is marked as complete.

There are multiple queues in our system as illustrated in Figure 3(b). Notice that this graph contains a cycle between the "Incoherent Ray Queue" and the "Shade Queue" handlers. This cycle models the recursive aspects of path tracing in the system.

The primary ray queue and incoherent ray queue are separate since we generate primary rays based on screen space tiles, and so can assume a certain amount of coherency. Mixing incoherent rays into these queues would serve to slow down the primary ray intersections. The occlusion ray queue is also separate since it is processed in Embree using a completely separate code path. Radiance writes, although implemented via atomics, are still queued to minimize thread contention for shared frame buffers.

Shade queues are the only queue type which are shared between threads. *One queue is allocated for each individual shader instance in the scene*, which ensures a baseline coherency when shading samples. The tradeoff with introducing any shared queues into the system is that it increases the likelihood of thread contention. We decided this was worth it however due to the memory savings of only needing to allocate a single shade queue per-shader, as opposed to one per-shader, per-thread.

5.2 Sorting

A "sort key" determines how entries are sorted when flushing a queue. Most queues in the system take 64-bit entries composed of a 32-bit sort key and a 32-bit payload reference. The queues are

Table 1: *Details of how we configure shade queue sort keys. Artists may link arbitrary lights to arbitrary geometry during the content creation phase. At runtime we use this information to partition the lights in the scene into subsets which we refer to as light-sets. Only one light-set can be attached to any single hit point and so it can represented as a simple index. We sort by this light-set index first to help to minimize code flow divergence in the integrator. Next we sort by UDIM tile, typically we use 10x10 UDIM grids [Seymour 2014] so 7 bits is sufficient here. Finally, we sort based on mip level, followed by morton encoded uv coordinates. All of these separate sort criteria are encoded into a single 32-bit integer.*

Bit location	Contents	Number of bits
25-31	light-set index	7 bits = 128 light sets
18-24	UDIM tile	7 bits = 128 tiles
14-17	mip level	4 bits = 16 mip levels
0-13	uv coordinates	14 bits = 16,384 tiles

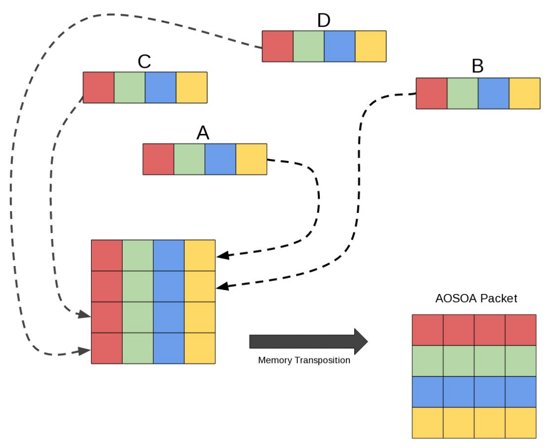

Figure 4: *Illustration of AOS to AOSOA transposition. Assume that A, B, C, and D are four 16-byte AOS structures scattered in memory. They have been prefetched by an earlier iteration of the AOS to AOSOA loop such that they are now resident in the local L1 or L2 cache. From there, they are loaded into SIMD registers (SSE in this case), where in-place transposition is performed using SIMD intrinsics. The result is an AOSOA packet ready to be consumed by an ISPC kernel.*

sorted with a radix sort algorithm using a less-than comparison of the entries' sort keys. Sorting the queue therefore only needs to access a single contiguous block of memory where these 64-bit entries reside. Radix sort inherently touches this memory in a sequential manner making the sort very fast. Furthermore, since not all queue types will use all the bits in a sort key, we can optimize by running specialized versions of the sort according to the number of significant bits actually used. Currently we support separate variations of the algorithm for 11, 22, or 32 bits.

The sort key contents are context specific to the type of queue we're sorting. Table 1 describes how we configure shade queue sort keys. As a result, when flushing a given shade queue, each required texture tile will typically need to be fetched only once as we progress through the entries.

Profiling shows that queue sorting takes on average about 1.5% of the rendering phase.

5.3 Efficient AOSOA transformation

To make best use of ISPC, inputs should be fed to it in either SOA or array-of-structures-of-arrays (AOSOA) format; please refer to [Intel 2011] for details. We chose to go with AOSOA since it provides better data locality than SOA for our use cases. The AOSOA stride is matched to that of the vector instruction set we're targeting, so for example, on AVX, we use an AOSOA stride of 8 32-bit words per component. Once data is in AOSOA format, each scalar work item maps to a separate vector lane, giving us the ability to process 4/8/16 items in parallel, depending on the targeted architecture.

Since sorting is preferably performed when inputs are in array-of-structures (AOS) format, we go through a two step process to prepare the data before handing it off to ISPC. The first step sorts the data references as described in Section 5.2. The second step takes these sorted references to AOS inputs, fetches the AOS data and transforms them in-place into AOSOA packets (Figure 4). Both of these steps are executed in C++ before invoking the associated ISPC kernel.

Accessing the elements of the AOS payload may result in fetching data from arbitrary cache lines on arbitrary memory pages. Memory fetches which need to go out to main memory can take multiple hundreds of cycles to complete. Fortunately this problem can be alleviated by using memory prefetching intrinsics. Since we are working on sizeable batches of AOS inputs, we have a priori knowledge of where in memory each structure lives. This allows us to prefetch cache lines in advance, such that by the time they are required, they will have been copied further up the processor cache hierarchy. The exact prefetch distance is configurable and chosen based on experimentation for different platforms.

Notice that the conversion from AOS into AOSOA is essentially a matrix transpose operation. This can be implemented efficiently in-place using a combination of SIMD unpack, shuffle and permute operations. A limitation of using these instructions is that they work on sets of 32-bit words, so we need to cater the contents of any C++ AOS data structures which we want to convert to AOSOA accordingly. For the most part, our basic data member types are float and (unsigned) integers which work fine. Pointers however need careful handling since they are 64-bit values and will be split up such that the upper and lower 32-bits of the original address won't be adjacent in memory anymore. Special code is needed in ISPC to reconstruct the original 64-bit address from the upper and lower 32-bit partial addresses. Likewise, similar care is needed for data types which are less than 32-bits. In practice, the reconstruction of these values is hidden behind `get()`/`set()` accessor functions on the ISPC side.

By carefully layering these transposition and prefetching building blocks, we can auto-transpose larger and more complex data structures on architectures of differing SIMD widths. Additionally these techniques can be applied in reverse order to convert from AOSOA back to AOS. Prefetching can also be used in this context to minimize the cost of the scatter operations when writing AOS structures back to memory.

Hand-in-hand with the careful access of memory is the careful alignment of memory. ISPC has an option to always generate aligned vector loads and stores, which can be safely turned on as long as memory allocations are aligned to multiples of the SIMD register width. This optimization is used throughout our shading, texturing, and integration code, and improved the vectorized code performance by an additional 7-8%.

5.4 Ray State Persistance

Since rays are queued in our breadth-first scheme, we need a place in memory to store the related state, similarly to previous work [Áfra et al. 2016; Eisenacher et al. 2013]. Recall that queue entries in the system contain 32-bit payload references. Typically these are indices into a pool of pre-allocated `RayState` structures. A `RayState` primarily consists of ray differential information, the current path throughput, and destination frame buffer specifics, among other data. Any other state which is required to persist whilst a ray is awaiting processing in a ray queue or shade queue is also contained in this structure. A lock-free memory pool allows any thread to allocate a `RayState` instance at any time which permits arbitrary ray splitting during integration. Additionally `RayState` instances can be processed and de-allocated on threads other than the allocating thread, which is necessary to support shared shade queues.

6 VECTORIZATION

We leverage Embree [Wald et al. 2014] for all ray intersections, which is already well vectorized internally. This section covers the other parts of the system which benefit from vectorization, namely shading, texturing, and integration.

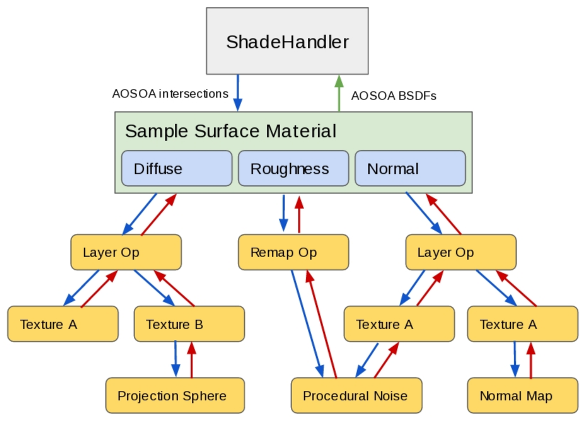

Figure 5: *An example of vectorized shading evaluation: a const array of varying* `Intersection` *objects in AOSOA format is passed from the ShadeHandler to a material implemented in ISPC, and down into a network of procedural/texture nodes as illustrated by the blue arrows. Each shade node, also implemented in ISPC, in turn returns a corresponding (varying) color as illustrated by the red arrows. Finally the material uses this information to populate the passed-in* `BSDF` *objects with the appropriate* `BsdfLobe`*s.*

6.1 Vectorized Shading

MoonRay supports a full-featured programmable shading system. For the scalar version of Moonray, the shader objects are implemented in C++ and excuted by a C++ shading framework. For vectorized *MoonRay*, the shader objects and the shading execution framework are implemented in ISPC. We support two different types of shaders which are connected together at run-time to form a shader graph as illustrated in Figure 5.

Material shaders are at the root of the shader graph hierarchy, where they output to the integrator `BSDF` objects which describe how the surface scatters light at an intersection point as described in Pharr et al. [Pharr et al. 2016]. Each `BSDF` is a list of weighted and parameterized `BsdfLobe`s. Higher `BsdfLobe` count means more expressive power for the `Bsdf` model but it also implies either an increase on variance or path integration cost. In our system, we limit each `BSDF` object to have at most 8 `BsdfLobe` objects, which in our experience is a good balance between artistic flexibility and rendering efficiency for `BSDF` modeling. Our system only allows each geometry primitive to be associated with a single material. Material layering is supported by a special material that takes two input materials to produce a `BSDF` object that is a user controllable blend of the two input `BSDF` objects.

The second type of shader is known as a shader node. A shader node produces a color value that is used to modify an input parameter of its parent shader. A texture map node or procedural noise node are some typical examples of shader nodes. Both shader types take exactly the same input shading state, also known as the `Intersection` object. The input state from the renderer is passed to the root material shader and is in turn passed down into child shader nodes without any modification, so there is no copying of input state at all during the shader graph evaluation.

Figure 5 illustrates the flow of AOSOA data up and down our shading graph during vectorized shader evaluation. From the sorted list of `RayState` references (see Table 1), the shade handler constructs a set of `Intersection` objects in AOSOA format, which are the primary inputs to a shader. Each lane of an `Intersection` contains differential geometry information about a hit point in a form that is independent of the underlying surface representation. Because we queued our `RayState` by shader instance, we only need a single ISPC function invocation to shade the entire `Intersection` bundle. The handling thread's TLS object is also passed into the shading function, allowing shaders to perform the allocations they will need in a lock-free manner.

The shading handler allocates and initializes a `BSDF` object corresponding to each varying `Intersection` object and passes these into the ISPC shading function to be populated with `BsdfLobe` objects. In ISPC terms, our `BSDF` contains a list of 8 uniform references to `varying BsdfLobe` structures. This layout easily handles the case where different parameterization and weights are needed on different lanes. Production shader code contains arbitrarily complex conditionals, often due to texturable input attributes. These conditionals not only affect the lobe parameters and weighting, but also the lobe type, and whether or not the lobe is generated in the first place. Since lobes are always varying, for each one added to the `BSDF`, we store the active ISPC `lanemask()` so we can determine at integration time whether or not a particular lobe is active for a particular lane.

The single most important role of the ISPC shader framework from a system performance perspective is to pass along coherent bundles of AOSOA data all the way through to the vectorized texture system. For production shots, texture mapping, initiated lazily by the shader code itself, is the overwhelming performance bottleneck encountered during shading. By writing Single Program/Multiple Data (SPMD) shaders in ISPC, we are able to maintain AOSOA bundles of data throughout the shading and texturing phases, and right through the integration phase.

6.2 Vectorized Texturing

We use OpenImageIO [Gritz 2007] (OIIO) as the basis for texturing in *MoonRay*. OIIO can be thought of as:

(1) An infrastructure for dealing with different types of image file formats in a format-agnostic manner, and various tools for manipulating the image data.

(2) A runtime image caching system which facilitates efficient rendering of scenes with larger texture memory footprints than could fit in physical memory.
(3) A runtime texture sampling system which layers on top of the image cache.

In our system, we make use of items 1 and 2 (with some customizations), but since all the inputs will already be in AOSOA format as textures are sampled from within the vectorized shading code, this prompted us to implement a fully vectorized ISPC replacement for item 3. OIIO provides a virtual interface class for adding new texture sampler implementations which simplified the process. We built upon the image tile cache functionality already provided in OIIO with the addition of a new set-associative micro-cache which is described later.

One impactful decision we took was to switch to a point sampling filter as our default. When sampling a texture, the appropriate adjacent mip levels are first determined using texture derivatives. Only a single non-interpolated value is looked up from each, and these values themselves are linearly interpolated based on the derivatives. This means we only have to fetch two texels for each request, resulting in a very fast, almost branchless texture lookup. Although we have not done a rigorous quantitative analysis, it works out well in practice. An informal justification for this decision can be thought of like so - we need to shoot sufficient primary rays to resolve geometric aliasing, which can be considered of infinite frequency. Intuitively, this number of rays should also be sufficient to resolve any texture aliasing from the already band-limited mip levels we sample. We do however fall back to vectorized bilinear filtering when texture magnification is required, i.e. we don't have a high enough resolution mip-map to satisfy the request.

OIIO has a main texture tile cache which is shared between all threads, and a two element thread-local micro-cache layered on top. This micro-cache was deemed too small to help much with vectorized execution since each vector lane may itself be accessing a different tile. To rectify this, we replaced it with a thread-local 4-way set-associative cache [Handy 1998], configured to hold the most recent 256 tiles accessed. Tile eviction is done using a strict least recently used policy. Moving up to 256 slots significantly improved the micro-cache hit rate over the default when running vectorized code, particularly when many textures were bound to a shader. Although vectorized ISPC code generates the complete list of tiles for all lanes required to complete the operation, a C++ function is called to query the micro-cache, and main tile cache if required, in a sequential fashion.

Additional features we've implemented in the vectorized texture samplers are full UDIM support, the option to return image map derivatives (via finite differencing), and the ability to pre-load all texture data in a scene during the preparation phase if desired.

6.3 Vectorized Integration

The ISPC integration kernel directly ingests batches of `BSDF` objects in AOSOA format from the shaders. Recall from the previous sections, that each SOA `BSDF` is structurally identical on all SIMD lanes, however the parameters, weights, and dedicated lanemasks can vary per lane, as each lane corresponds to a small set of coherent ray hits using the same material shader and the same light-set.

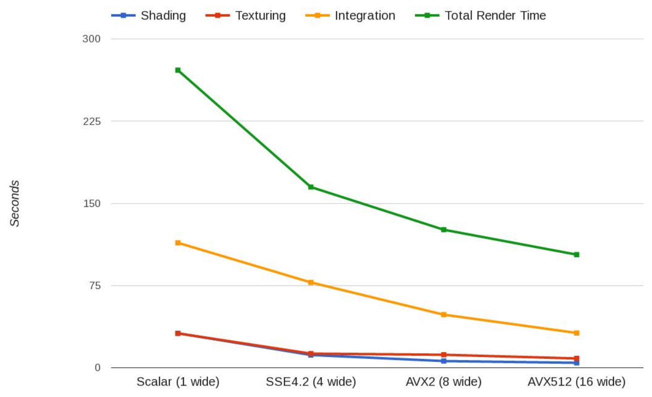

Figure 6: *This graph demonstrates how performance scales as the number of vector lanes increases. Although we still observe non-trivial gains when moving from 8 to 16 lanes, clearly we also see diminishing returns. These numbers were captured by rendering the Bergen Town scene (Figure 2(a)) on Intel's Knights Landing hardware.*

The vectorized integrator can therefore proceed in a data-parallel fashion, drawing samples and performing all Bsdf and light importance sampling, evaluation and MIS weighting calculcations in SIMD. Path splitting and Russian Roulette calculations are also computed similarly. All this math-intensive body of code is written in ISPC, which automatically handles the lane masking within its non-trivial control flow. As shown in Section 7, our vectorized integrator suffers from relatively little SIMD divergence.

A notable difficulty occurs with textured light importance sampling, which requires performing a binary search in conditional cumulative distribution function tables that are different on each lane. This causes memory gather and SIMD divergence of about 50%.

To handle spawning new rays with or without path splitting, the integrator fills arena allocated memory buffers with the required information to construct these new rays. The actual `RayState` management and subsequent queuing of spawned rays is handled with a call out to C++. In addition to the spawned rays, the integator kernel is also responsible for providing a list of occlusion rays, and a list of radiances to add to the frame buffer (see the arrows which emanate from the Shade Queue Handler in Figure 3(b)). Since these outputs are generated from ISPC, they will be in AOSOA format, so a conversion back to AOS format (see Section 5.3) is performed in C++ before the subsequent queuing of these items.

Rays from different generations are freely mixed within queues since the majority of the shading/integration code paths are the same regardless of depth. Since there isn't any concept of a one-to-one correspondence between input and output rays, there is no risk of fragmentation and therefore no ray compaction nor regeneration [Áfra et al. 2016] is required.

We put effort into ensuring that each lane executes completely independently of any other lane, since any cross communication could introduce unwanted indeterminism into the system when running on multiple threads.

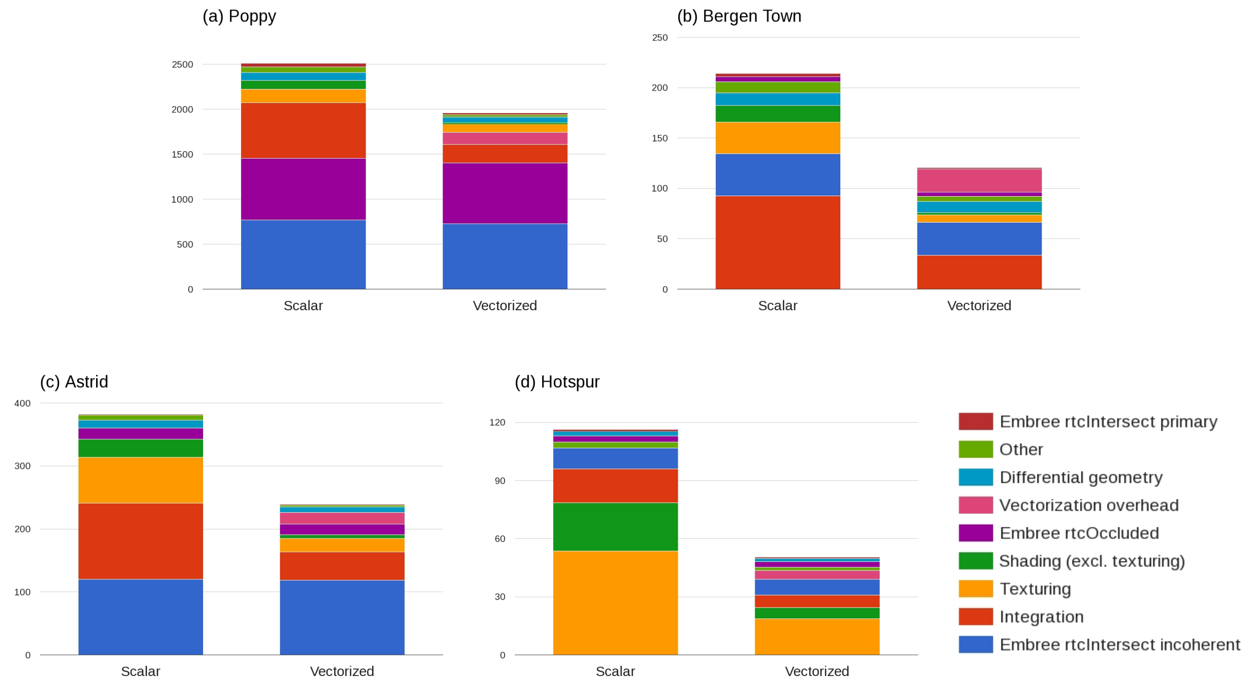

Figure 7: *Comparisons of scalar vs. vectorized execution time for the Poppy (Figure 1), Bergen Town, Astrid, and Hotspur (Figures 2(a), 2(b), 2(c)) scenes respectively. The total time spent rendering corresponds to the height of each stack, measured in seconds. A legend on the right is provided to give a feel for how time is divided up between the various subsystems. A category called “Vectorization overhead” accounts for all the time spent in queuing, sorting, and AOSOA code. In general, the speedup we observe from vectorized execution is proportional to the percentage of time spent in shading, texturing and integration code.*

Table 2: *A breakdown of key statisics for the scenes profiled. The “ISPC time %” shows the percentage of time in our code we’re processing a vector lane’s worth of samples simultaneously (e.g. 8 for AVX2). This is computed by taking the total time spent executing ISPC code and dividing it by the total render time less any time spent inside of Embree. It’s interesting to note that Bergen Town shows a much higher percentage of time devoted to vectorization overhead than other scenes. This can be attributed to it being one of the more simple scenes with basic geometry and shaders, and thus a higher ray throughput. The cost of queuing, sorting, and AOSOA tends to remain stable per ray and so the more complex the workload, the more the vectorization overhead is amortized.*

Scene	**Poppy**	**Bergen Town**	**Astrid**	**Hotspur**
Total scalar time (mm:ss)	41:53	3:34	6:22	1:56
Total vectorized time (mm:ss)	32:41	2:01	3:59	0:50
Millions of rays/sec scalar	3.76	10.43	2.94	5.02
Millions of rays/sec vectorized	4.75	18.47	5.01	11.57
Millions of shader evals/sec scalar	1.66	5.43	1.73	1.14
Millions of shader evals/sec vectorized	2.10	9.63	2.98	2.63
Ray intersection subsystem speedup	1.02×	1.20×	1.00×	1.14×
Shading subsystem speedup	**5.09×**	**6.19×**	**4.54×**	**4.20×**
Texturing subsystem speedup	**1.68×**	**4.24×**	**3.43×**	**2.90×**
Integration subsystem speedup	**3.00×**	**2.75×**	**2.68×**	**2.72×**
Vectorization overhead %	6.86%	18.99%	7.48%	8.82%
ISPC time % (excl. ray intersections)	58.57%	52.38%	70.47%	79.76%
Light SIMD utilization %	61.77%	51.38%	68.72%	58.22%
BSDF SIMD utilization %	71.96%	90.43%	70.39%	80.17%
Overall speedup	**1.28×**	**1.77×**	**1.60×**	**2.31×**

7 RESULTS

All tests in Figure 7 were run on a machine with dual Intel Xeon E5-2697 v3 CPUs running at 2.6GHz. Hyper-threading was disabled which gave a total of 28 hardware threads. We compiled the code to target the AVX2 instruction set which is 8 lanes wide, and we configured the texture system to lazily load textures on demand.

Great care was taken to ensure we performed the same high level computations in both scalar and vectorized modes, even though the internal architecture and implementation can differ significantly. For example, since we wrote a custom texture sampler for the vectorized code path, we also implemented a scalar version with identical functionality and outputs so that comparisons are fair. Each of the 4 images shown in Figures 1 and 2 is visually identical when run in either mode. See Table 2 for associated statistics for each scene profiled.

One important point to keep in mind when looking at the stacked graphs in Figure 7 is that *the blue and purple blocks represent time spent inside of Embree.* Time spent inside Embree is outside of the scope of our optimization work in this paper, but can nonetheless be a significant part of overall render time. In general we observe lower overall speedup numbers for scenes where a lot of time is spent intersecting rays. This can be attributed to Embree being very well optimized for both scalar and vectorized use cases.

A good example of this is in the Poppy scene (Figure 1). Every character and asset in this scene is covered in hair or fuzz, resulting in the render time being very much dominated by ray/hair intersections [Woop et al. 2014]. In fact we spend 59% of time inside of Embree for the scalar case, a portion of time which doesn't get faster with vectorized execution. As a result we only have 41% of the frame time left to speed up. Despite this, we observe over a 2× speedup for all non-intersection related work, which results in a 28% overall speedup when factoring in ray intersections.

The corollary to this however is that we observe larger speedups in scenes which are dominated by shading, texturing and/or integration. The remaining scenes show examples of this scenario.

Bergen Town (Figure 2(a)) is dominated by integration time (Figure 7(b)), which as mentioned is a good case for the vectorized code path. Interestingly, we do see a 20% speedup in ray intersection time here for vectorized execution, likely since this scene consists of more typical pre-tesselated subdivision surfaces. This, combined with a 2.21× speedup in integration and a 4.24× speedup in texturing results in a 77% overall speedup compared to scalar mode. This scene was also profiled on Intel's Knights Landing hardware using lane widths of 4, 8, and 16; see Figure 6 for details.

Astrid (Figure 2(b)) is one of the main characters in the movie "*How to Train Your Dragon*". She has long thick blond hair and her clothing is also covered in fur. Like the Poppy scene, we don't see a meaningful speedup between scalar and vectorized ray intersections in the Astrid render due to heavy hair intersection. The non-intersection related work however speeds up by a factor of 2.45×, giving an overall speedup of 1.6× for the scene.

The Hotspur asset (Figure 2(c)) represents a typical environment set piece in our films. Large environment elements have to withstand the challenge of being rendered close up from arbitrary camera angles. They often rely on high resolution images which are layered and projected via complex shader networks. In Hotspur's case there were over 6 layers of different blends and projections which can access over 11 GB of texture data. In this type of setup, texturing is the primary bottleneck, followed by shading; see Figure 7(d). In both of these vectorized code paths, we see large gains - a 3.44× speedup in shading and a 2.90× speedup in texturing, and an overall speedup of 2.31×.

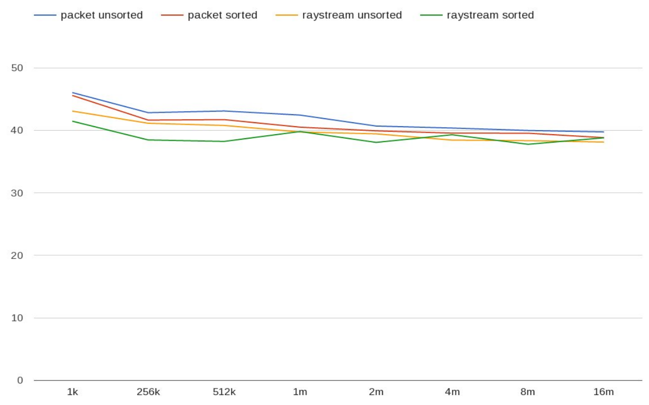

Figure 8: *A plot of Embree ray intersection time vs. incoherent queue size for the Bergen Town scene (Figure 2(a)). We tested using both packet and ray streaming APIs, and with and without ray sorting (the cost of ray sorting is not shown here). We observe only a relatively slight speed up in ray intersection time. This, combined with the extra time required to sort rays, and additional memory usage to store intermediate* `RayStates`*, ended up not being a win overall.*

Rendering cost grows logarithmically with geometric complexity, but linearly with shading and lighting complexity. Users typically add many lights and layers of shaders and textures, which causes the shading, texturing and path integration components of the rendering system to slow down proportionally. Fortunately, these are the areas of our system that benefit the most from our vectorization work.

8 LIMITATIONS

During development we experimented with ray sorting using an optimized 5D sort in combination with ray queues up to 16 million in size [Eisenacher et al. 2013]. Figure 8 shows the corresponding Embree ray intersection performance. Consequently we are not currently pursuing this avenue. In our tests, the sweet spot for the incoherent ray queue size is about 1024 entries per thread. Whilst this number itself is too small to manufacture any meaningful ray coherence from sorting, we do get some additional coherence as a byproduct of having a queue per shader instance, since each new batch of ray hit and continuation rays will be correlated to surfaces which share that shader.

We don't currently support lazy evaluation of procedurals for simplicity of implementation, and due to the complex thread synchronization and potential contention when loading geometry assets while tracing paths through the scene. Instead, all procedurals are expanded during the preparation phase, and we chose to place the onus on the upstream pipeline or pre-processing tools to generate appropriately trimmed scene files.

Writing vectorized code and thinking in a vectorized fashion can be a challenge in itself. It can be somewhat at odds with fast prototyping workflows required to keep a renderer fluidly evolving. A compromise we've found which has worked well so far is to allow execution of C++ scalar recursive code paths from within the vectorized framework. This allows programmers to experiment in the same manner as they would in a scalar depth first context. This new code runs at standard scalar speed since it's not vectorized, but at the same time we still get the full benefit from the portions of the code which are already vectorized. This hybrid approach has been utilized to implement subsurface scattering and volumes. As new functionality solidifies over time, the slower scalar code can be upgraded to full vectorized implementations.

9 CONCLUSION

We presented a system-wide approach to vectorized path tracing. This included an efficient queueing and sorting foundation for extracting data and computation coherence; and vectorization of the remaining compute-intensive parts of path tracing: shading, texturing and integration.

In the introduction we asked the question "Do the potential performance benefits gained outweigh the extra work required to harness the vector hardware?". In all scenes we profiled so far with *MoonRay*, the answer is a definitive yes.

We have seen that the actual speed up itself is very scene dependent. Scenes which spend the majority of time performing ray intersections benefit less than those which spend most of their time performing shading, texturing or integration computations. This is encouraging since we expect the relative shader and lighting complexity to increase as more complex production scenes are developed.

Results so far have been promising and we're excited and optimistic to discover how *MoonRay* performs on its first full feature.

ACKNOWLEDGMENTS

MoonRay is the work of two passionate teams of researchers and engineers, one working on the core renderer and the other on shaders and procedurals. This paper is dedicated to them! The authors would also like to express the most profound gratitude to our technology partners at Intel, and to the many open source contributors of Embree, ISPC, OpenImageIO and OpenSubdiv, without whom this project could not have been realized. A special thank you goes to Scott Cegielski, Conrad Egan, Krishna Mullia, and Jon Lanz for helping us with rendering tests, and to Pat Hanrahan and Matt Pharr, for providing feedback on in-progress drafts of this paper.

REFERENCES

Mike Acton. 2014. Data-Oriented Design and C++. (2014). https://www.youtube.com/watch?v=rX0ItVEVjHc.

Attila T. Áfra, Carsten Benthin, Ingo Wald, and Jacob Munkberg. 2016. Local Shading Coherence Extraction for SIMD-efficient Path Tracing on CPUs. In *Proceedings of High Performance Graphics (HPG '16)*. 119–128.

Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of the Conference on High Performance Graphics (HPG '10)*. 113–122.

Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R. Mark. 2012. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics* (2012), 1438–1448.

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*. 95–102.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted Deferred Shading for Production Path Tracing. In *Proceedings of the Eurographics Symposium on Rendering (EGSR '13)*. 125–132.

Manfred Ernst and Gunther Greiner. 2008. Multi Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing (RT '08)*.

Kirill Garanzha and Charles Loop. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum* (2010), 289–298.

Larry Gritz. 2007. *Open Image I/O*. https://github.com/OpenImageIO/oiio/.

Jim Handy. 1998. *The Cache Memory Book*. Morgan Kaufmann. https://www.elsevier.com/books/cache-memory-book-the/handy/978-0-08-051878-7.

P Hanrahan. 1986. Using Caching and Breadth-first Search to Speed Up Ray-tracing. In *Proceedings on Graphics Interface '86/Vision Interface '86*. 56–61.

Jared Hoberock, Victor Lu, Yuntao Jia, and John C. Hart. 2009. Stream Compaction for Deferred Shading. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. 173–180.

Sony Imageworks. 2017. Alembic. (2017). http://www.alembic.io/.

Intel. 2010. *Threading Building Blocks*. https://www.threadingbuildingblocks.org/.

Intel. 2011. *Intel SPMD Program Compiler*. https://ispc.github.io/.

James T. Kajiya. 1986. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. 143–150.

Alexander Keller, Carsten Wächter, Matthias Raab, Daniel Seibert, Dietger van Antwerpen, Johann Korndörfer, and Lutz Kettner. 2017. The Iray Light Transport Simulation and Rendering System. *CoRR* abs/1705.01263 (2017). http://arxiv.org/abs/1705.01263

Janne Kontkanen, Eric Tabellion, and Ryan S. Overbeck. 2011. Coherent Out-of-core Point-based Global Illumination. In *Proceedings of the Twenty-second Eurographics Conference on Rendering (EGSR '11)*. 1353–1360.

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. 137–143.

Koji Nakamaru and Yoshio Ohno. 2002. Ray Tracing for Curve Primitives. In *Proceedings of Winter School of Computer Graphics (WSCG)*.

Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. 2010. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH '10)*. 37:1–37:10.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH '10)*. 66:1–66:13.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering, 3rd ed.* Morgan Kaufmann. http://www.pbrt.org/.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering Complex Scenes with Memory-coherent Ray Tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. 101–108.

M. Pharr and W. R. Mark. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*. 1–13.

Mike Seymour. 2014. UDIM UV mapping. (2014). https://www.fxguide.com/featured/udim-uv-mapping/.

John A. Tsakok. 2009. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. 151–158.

Eric Veach and Leonidas J. Guibas. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. 419–428.

Jiří Vorba and Jaroslav Křivánek. 2016. Adjoint-Driven Russian Roulette and Splitting in Light Transport Simulation. *ACM Trans. Graph.* (2016), 1–11.

Ingo Wald. 2007. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing (RT '07)*. 33–40.

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs. In *IEEE Symposium on Interactive Ray Tracing (RT '08)*.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* (2014), 143:1–143:8.

Sven Woop, Carsten Benthin, Ingo Wald, Gregory S. Johnson, and Eric Tabellion. 2014. Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur. In *Proceedings of High Performance Graphics (HPG '14)*. 41–49.