

Ray Accelerator: Efficient and Flexible Ray Tracing on a Heterogeneous Architecture

R. Barringer,¹ M. Andersson,^{1,2} and T. Akenine-Möller^{1,2}

¹Lund University ²Intel Corporation

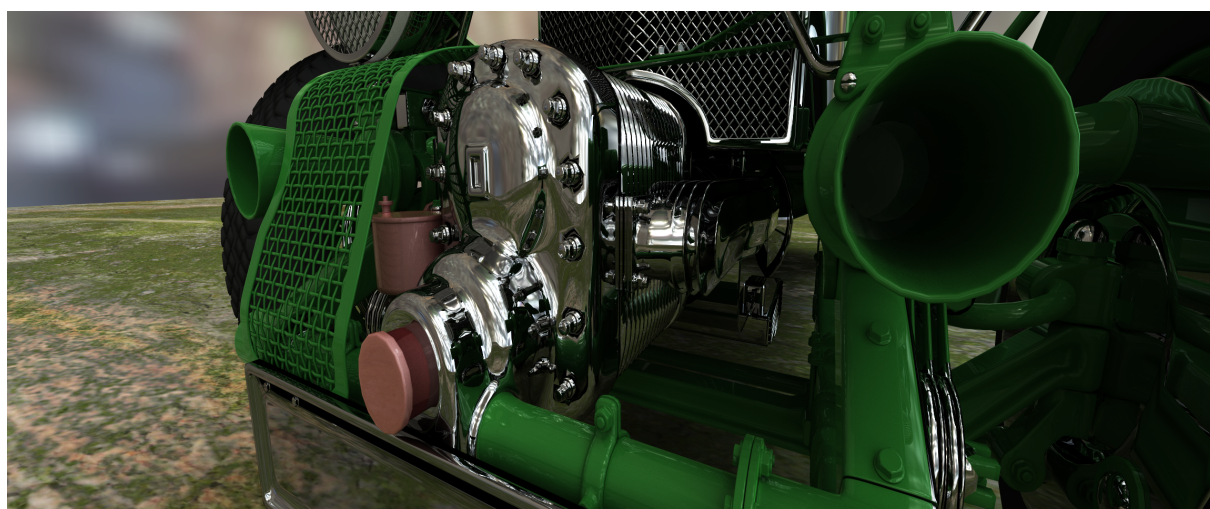


Figure 1: Our hybrid ray tracing system executes on both the CPU and the graphics processor using a novel scheduling algorithm. The system achieves high performance by utilizing the two units’ combined resources while keeping data in shared memory. The zoom-in of the Bentley car above shows that our system can handle complex, layered BRDFs with importance sampling and texturing.

Abstract

We present a hybrid ray tracing system, where the work is divided between the CPU cores and the GPU in an integrated chip, and communication occurs via shared memory. Rays are organized in large packets that can be distributed among the two units as needed. Testing visibility between rays and the scene is mostly performed using an optimized kernel on the GPU, but the CPU can help as necessary. The CPU cores typically handle most or all shading, which makes it easy to support complex appearances. For efficiency, the CPU cores shade whole batches of rays by sorting them on material and shading each material using a vectorized kernel. In addition, we introduce a method to support light paths with arbitrary recursion, such as multiple recursive Whitted-style ray tracing and adaptive sampling where the result of a ray is examined before sending the next, while still batching up rays for the benefit of GPU-accelerated traversal and vectorized shading. This allows our system to achieve high rendering performance while maintaining the flexibility to accommodate different rendering algorithms.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray tracing, I.3.2 [Computer Graphics]: Graphics Systems—

1. Introduction

Ray tracing [Whi80] and path tracing [Kaj86] are fundamental computer graphics techniques for realistic rendering. To render images quickly, one may use multi-core CPUs, Xeon Phi-architectures [WWB*14], or GPUs [CHH02, AL09, PBD*10, ALK12]. However, there are few ray tracers that exploit systems with several CPU cores and an integrated graphics processor on the same die (see Section 2). This chip configuration is common since the predominant architectures for, e.g., laptop computers are organized in this way. Our goal is to create a highly optimized rendering framework for such architectures, with the intention to exploit its particular features while maintaining support for various rendering algorithms. This includes extracting shader coherency for vectorized shading and batching rays for efficient traversal on the GPU. In order to demonstrate the flexibility of our approach, we implement both a path tracer and a Whitted-style ray tracer with multiple recursive rays at each surface, within the same rendering framework. A result from our hybrid ray tracing system can be seen in Figure 1, featuring complex BRDFs with importance sampling and texturing.

There are several difficult challenges involved in achieving high performance on architectures with a CPU and a GPU on the same die. Both the graphics processor and the CPU cores share the same memory subsystem, specifically from the last-level cache (LLC) and beyond in the Haswell architecture [HMB*14]. If the CPU cores are working on a memory-intensive application, the graphics processor may get lower performance since it competes for the same resources. In addition, Haswell and its successor Broadwell use fully integrated voltage regulators (FIVR) to manage and direct power. The FIVR can, for example, direct the majority of the package power to the subsystem (e.g., graphics processor) that needs the most power. The reason to have the FIVRs is to save power, but also because it is not possible to run the entire chip at maximum clock frequency at all times due to dark silicon [EBA*11]. Both FIVR and a shared memory system make it more of a challenge to exploit both the CPU cores and the graphics processor to work on the same task. However, they are also key to increasing performance per Watt, and we use shared memory to share data between the CPU cores and the graphics processor without any associated copy or transfer cost.

To summarize the contributions of this paper, it presents a flexible system for ray tracing-based rendering that uses a novel scheduling algorithm based on large packets of rays. Using this approach, shader coherency can be extracted to utilize SIMD instructions and packets can be intersection tested on both the CPU and the integrated graphics processor using shared memory for fast communication. We also introduce *loop shaders* (see Section 3.3) as a method to achieve flexibility similar to single-ray ray tracing, while still batching up rays for performance. In addition, we introduce the

idea to use the graphics processors texture units after traversal and route the result back to the CPU.

2. Previous Work

Weghorst et al. [WHG84] were early with combining the power of rasterization and ray tracing. Since rasterization is highly optimized for eye rays, the scene is rasterized from the camera's point of view followed by shooting secondary rays using a ray tracer. However, all rasterization and ray tracing work is done on a CPU. Since then, their approach has been implemented in several different flavors, where both CPU and GPU cores are utilized. Chen and Liu [CL07] use the GPU to generate primary visibility, and then the first-hit data is read back to the CPU, which performs tracing and shading of secondary rays. Sabino et al. [SAGC*12] use deferred shading on the GPU for primary rays and then OptiX [PBD*10] for secondary effects. Robert et al. [RSB07] generate first hits using rasterization, compute shadows using shadow mapping, and the CPU cores then handle ray tracing of secondary effects. In other work, Budge et al. [BAGJ08] create the bounding volume hierarchy (BVH) on the CPU, and in parallel, the previous frame is ray traced on the GPU. Recent work utilizes hardware rasterization and occlusion queries on the GPU to speed up coherent ray tracing of large scenes [MBV*15]. Batches of rays traverse a combined screen- and object-space hierarchy, and a generalized form of occlusion queries is used to perform occlusion culling for batches of rays with arbitrary origins and directions. In addition, temporal coherence is exploited by initiating each frame by intersection testing the closest geometry from the previous frame.

While there are other examples of previous work that utilize both the GPU and CPU cores, these differ on several points compared to our paper. For example, Nah et al. [NKL*10] use OpenGL ES to implement a ray tracer, where the CPU handles ray generation and builds the acceleration structure, while ray traversal and shading are done in shader code on the GPU. However, their results were generated using a simulator and no source code is available. Budge et al. [BBS*09] present a data management layer in software for handling out-of core path tracing on CPUs and GPUs. Their CPUs had an activity level of at most 15%, while our target is to make the CPU cores work as efficiently as possible in order to finish rendering the image earlier. Pajot et al. [PBPP11] reformulate bidirectional path tracing to exploit simultaneous execution on both the CPU and the GPU.

By using path tracing, the Brigade system [BvS13] pushes the edge of rendering quality for game engines. The system is targeting heterogeneous systems, where game logic and BVH updates are done on the CPU cores, while path tracing can be done on one or more GPUs. In theory, it is mentioned that when the CPU cores are idle, they can help with path tracing as well, but *no* successful implementation

was shown. Their ray tracers were implemented either using CUDA or OpenCL.

Aila and Laine [AL09] investigate what could possibly be reached in terms of ray tracing performance on SIMD/SIMT machines. They use a GPU software simulator, where memory accesses were assumed to return immediately, and an upper bound of performance could then be found. In addition, they propose and implement several improvements to the traversal method, and avoid scheduling inefficiencies by using a *persistent threads* approach. Their results show that the measured performance was about 90% of the simulated upper bound for diffuse rays for scenes with 80–282k triangles. Their work has been optimized for new GPUs as well [ALK12].

There are several ray tracers that run entirely on the GPU. One of these is the OptiX system [PBD*10] that runs on NVIDIA GPUs. It is highly programmable and the core of the system is a domain-specific just-in-time compiler. Many types of applications, such as Whitted ray tracing, ambient occlusion, collision detection, path tracing, etc have been shown to run at interactive rates. Laine et al. suggest that using a single large kernel for ray tracing is sub-optimal, especially when used in combination with expensive material evaluation [LKA13]. Instead, they propose to split ray queries and material evaluation into separate kernels and demonstrate a system for standard path tracing. For an overview of GPU-based progressive light transport algorithms, see the survey by Davidovič et al. [DKHS14].

LuxRender's[†] LuxMark is a ray tracing system that runs on both CPUs and GPUs, and using the combination of both. It uses Embree for tracing on the CPU and OpenCL is used for tracing on the GPU. The GPU part is using micro kernels with ray buffers. On our target platform, LuxMark runs at 4 megarays/second for a scene with 217k triangles, while our renderer runs at 14–30 megarays/s for scenes with 2.3–7.9M triangles. There is no documentation about LuxMark except for the source code, but it is highly likely that our system and theirs are quite different (e.g., LuxMark is likely more capable, etc), and as a result, it is extremely difficult to provide a fair comparison between our renderer and LuxMark. Note that LuxMark is one of the most used benchmark for OpenCL and has widespread use.

Embree is a ray tracing system, targeting CPUs and Xeon Phi, which consists of highly optimized traversal and intersection kernels for ray tracing [WWB*14]. This system includes packet traversal [WSBW01], single-ray traversal, and hybrid kernels which use packets in the beginning and switches to single-ray traversal when utilization becomes low [BWW*12]. GPU ray tracers often have binary trees for their bounding volume hierarchies (BVHs), but Embree uses multi-BVHs [WBB08, EG08, DHK08] with four or more

children per node. This improves SIMD efficiency in intersection tests. In our work, we use Embree 2.7 to trace extra rays on the CPU if any of the CPU threads are idle.

3. Heterogeneous Ray Tracing

In our approach, the majority of ray queries (BVH traversal and triangle intersection) are handled by the GPU, while the remaining parts of the renderer is implemented on the CPU. The CPU is thus responsible for creating eye rays, scheduling ray queries, and evaluating shading at surfaces. This means that the GPU is used mostly as a ray query accelerator. We argue that this is a good use of the graphics hardware since the ray query kernel can be heavily optimized for the specific GPU in the system, and since rendering systems can make use of the GPU without making the entire renderer and all BRDFs work well on the hardware. It may even be inefficient to shade on the GPU when the number of materials in the scene becomes large. A typical GPU requires thousands of data items to be shaded using the same kernel, in order to maintain high efficiency. This may be hard to accomplish, even with sophisticated scheduling. The amount of available parallelism would simply diminish as the number of rays hitting the same material decreases.

In order to exploit the fixed-function hardware available on the GPU for texture filtering [Wil83], texture sampling can also be performed on the GPU once the closest visible surface along a ray has been determined. Such texture lookups can be routed back to the CPU and utilized during BRDF evaluation. Ideally, the GPU can be used to sample both an environmental light probe, in case of a miss, and textures at surfaces, in case of a hit. However, current API limitations restrict our use of this technique in our implementation (see Section 4). To further improve efficiency, we also allow the CPU to perform some ray queries if time permits. This gives a performance boost and makes for better load balancing, as will be discussed in Section 5.

In our hybrid system, work is distributed using large buffers of *arbitrary* rays, which we call *ray streams*, similar to previous work [BAM14]. We gather a sufficient number of rays so that the resources available in both the CPU and the GPU are saturated. Our software architecture is illustrated in Figure 2. A number of *ray dispatch threads* feed the GPU with work in the form of ray streams to perform ray queries on. In addition, a number of *host threads* run on the CPU cores that perform actual CPU work, e.g., shading, eye ray generation, and ray queries. Both ray dispatch threads and host threads steal tasks from a central scheduler, which determines what to do next. As the GPU is responsible for performing most ray queries, we need to pass ray streams from the CPU to the GPU and back. On architectures with a shared memory between CPU and GPU [HMB*14, Pia12], these transactions are done without any additional work, such as copying memory. Scheduling ray queries between CPU and GPU is thus extremely efficient in this case. This

[†] <http://www.luxrender.net>

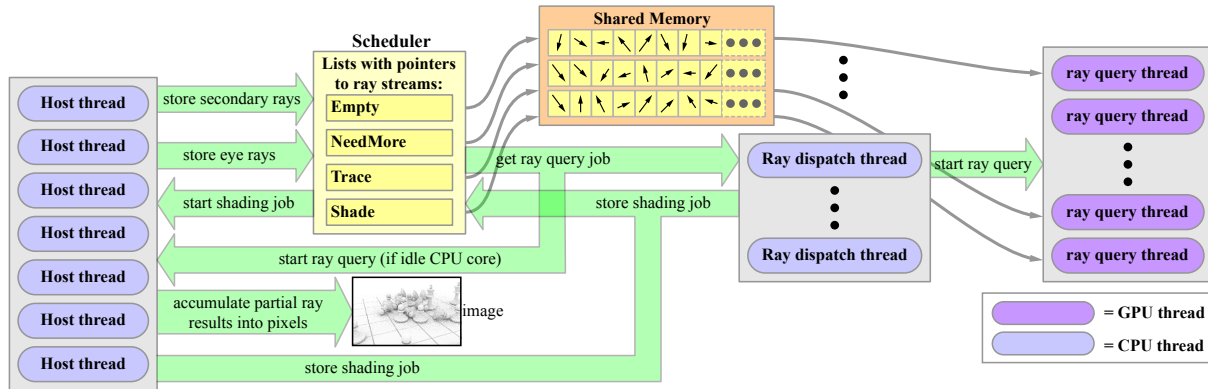


Figure 2: Our ray tracing system running on a heterogeneous architecture with shared memory. All communication happens via ray streams located in shared memory, and threads running on the CPU cores handle creation of eye rays, shading, and generation of secondary rays. The GPU threads are only responsible for ray queries (i.e., BVH traversal and intersection testing) and if the intersected surface is textured, the GPU thread can also provide a filtered texture lookup using the available hardware. Idle CPU cores will help with ray queries to maximize performance.

is exploited in our implementation described in Section 4. While we have focused on systems with shared memory in this paper, it is likely that a similar system would work even without shared memory, using asynchronous memory transfers. In this case, the system would have to tolerate even longer latencies for dispatching and tracing rays. Exploring the full ramifications of such a design is deemed out of scope of this paper and is left for future work.

The ray streams are scheduled for different kinds of work using a low overhead method that simply keeps a pointer to each ray stream in one of multiple work lists. First, there is one list for empty ray streams and one list for ray streams that need more rays before ray queries can commence. Ray streams can end up in the latter list if they do not contain enough rays to be efficiently executed on the graphics processor. In addition, there is one list for rays waiting for ray queries to be performed, and one list for ray streams that need shading. At any time, the system can have up to N rays in flight and allocates enough ray streams to store all of them. We have found that keeping about 256k rays active simultaneously leads to good utilization on our target architecture. The size of a single ray stream (and its preferred size before performing a ray query) is typically mandated by the underlying hardware. This size, together with N , determines the number of ray streams that need to be allocated in the system.

The scheduler balances work so that both CPU cores and the GPU are as active as possible. Let the different work lists be named EMPTY, NEEDMORE, TRACE, and SHADE. Initially all ray streams reside in the EMPTY list and will only transfer to another list as rays are appended to them. TRACE contains ray streams large enough to be intersection tested on the GPU. Ray streams that have fewer rays end up in the

NEEDMORE work list, which the scheduler will attempt to add more rays to when possible, in the hope that they can migrate to the TRACE work list. These ray streams will only be traced if there is no other work to do. Depending on circumstances, multiple ray streams may be in NEEDMORE at the same time. SHADE contains all ray streams that have been traced and are in need of shading. While the ray dispatch threads continuously attempt to feed the GPU with ray streams from TRACE, the host threads can perform a variety of tasks. When a host thread requests a task, it will receive a task in the following order of priority:

1. If the number of active rays is less than N , and more eye rays are still to be traced, return a task to add more eye rays to a ray stream. The ray stream is taken from NEEDMORE or EMPTY. In practice this is performed at tile granularity.
2. If SHADE has a ray stream, return a shading task for that ray stream, together with a ray stream from NEEDMORE or EMPTY to contain secondary rays, with priority to NEEDMORE. Shading tasks also take care to update the number of active rays as appropriate.
3. If all ray dispatch threads are occupied and TRACE has a ray stream, a ray query task for that ray stream is returned.
4. If NEEDMORE has a ray stream, a ray query task for that ray stream is returned.

If all the above conditions fail, the host thread will suspend until there is a change in the work lists. The rationale behind the priority of eye rays (#1 above) is that we would like to start tracing them as soon as possible to avoid long running chains of secondary rays toward the end of a frame. If there is no other work, we allow the CPU to perform ray queries (#3). However, care must be taken not to steal away tasks

from an underutilized GPU, and hence, we add high GPU activity as a precondition before starting a full ray stream on the CPU. Note that while this scheduling algorithm may sound cumbersome, it has low overhead in reality since it only amounts to popping a pointer from a handful of work lists. It is also worth mentioning that #4 results in that small ray streams may be processed towards the end of a frame, which provides better load balancing.

We have found that tracing an entire ray stream on a CPU core may occupy the CPU long enough to starve the GPU from work. In order to avoid this problem, we only allow CPU cores to process parts of a ray stream, in chunks of 1024 rays at a time. When a CPU core starts processing a ray stream, that ray stream ends up in a special job slot in the scheduler. Any other CPU core that wants to trace rays will be redirected to a subset of the same ray stream until it has been completed.

Shading may also happen in chunks. The ray stream that is used to store secondary rays can come from NEEDMORE, in which case it is already partially occupied. We may therefore have to shade a subset of the rays and return the task to the scheduler in order to get a new ray stream to output to. In our implementation, we always shade in batches of at most 8192 rays and ensure that all ray streams in NEEDMORE accommodates this number of appended rays.

3.1. Rendering Architecture

While the core of our system handles scheduling of ray streams on both CPU and GPU as well as intersection queries, a mechanism is needed for implementing a rendering algorithm on top of it. The scheduler provides callbacks to an external renderer for spawning camera rays and for shading batches of rays. The tiling mechanism and frame buffer accumulation strategy is completely up to the renderer. The scheduler simply knows that the renderer can spawn more rays until a limit is reached and that intersection tested rays can be shaded.

When a ray stream from SHADE is processed in our system, all rays execute the same program as mandated by the renderer, i.e., after a ray query has been performed. Each triangle in the scene may refer to renderer-specific material information, such as a BRDF. For efficiency, the shading program first sorts rays based on the outcome of the ray query (hit or miss) as well as the surface material information (in case of a hit). After sorting, each subset of rays can be processed using SIMD execution by processing multiple rays at once (such as 8 when using AVX2). This work includes attribute (e.g., normal) interpolation, BRDF evaluation, pixel information accumulation, and potentially new ray generation, as required by the rendering algorithm. The approach of sorting per surface material is a very simple, but efficient technique for extracting shader coherency [ENSB13].

Rays typically need some additional information to be

stored with them to support the underlying rendering algorithm. The renderer is free to allocate additional data for each ray in a ray stream, which we denote *payload*, in whichever way is most suitable. Payloads are typically mapped to ray streams using a unique index associated with each ray stream (an integer from 0 to $n - 1$, where n is the number of ray streams). In our path tracer, for example, we store the current weight of the ray (three 32-bit floating-point values) as well as the current ray depth and originating pixel encoded in a 32-bit integer. The payload may also include ray differentials [Ige99] if needed by the renderer, for example.

3.2. Resource Constraints and Recursive Rays

The scheduling algorithm is a good fit for path tracing, where each light path has at most one active ray (not counting shadow rays). If a single ray stream is dispatched, there will always be resources to continue those paths since the rays in the stream can be overwritten. A number of shadow ray emissions can trivially be supported at each hit by keeping a separate ray stream for shadow rays and always process them before continuing the light paths.[‡] It is thus straightforward to implement a path tracing renderer in our system.

Interestingly, the problem becomes more difficult with, e.g., Whitted-style ray tracing [Whi80], where a single ray can spawn multiple rays depending on the surface. Given infinite storage for ray streams, it would be possible to have a surface shader spawn any number of new rays on a hit and just append them to new ray streams. In practice, however, resources are limited and the number of rays grows exponentially. For example, if four rays are fired from each hit with a recursion depth of 16, a single initial ray can result in over 4 billion recursive rays. Storing that many rays does not seem feasible nor efficient.

A related problem is the existence of interdependencies between rays. Rays originating from the same initial ray usually want to accumulate information to the same frame buffer pixel. If all rays are shaded in parallel, this would likely result in race conditions or force expensive synchronization between ray streams. Also, if there is no order between rays, a surface shader cannot depend on the result of a recursive ray in the shader, before sending the next ray, which could be useful for adaptive sampling strategies.

Traditional single-ray ray tracing does not have these problems because only a depth first set of all recursive rays are active at any given time. That is, the states of surface shaders and rays leading up to the current ray are stored on the program stack. Looking at our previous example, at most 16 rays and shader states are active in memory, regardless of the number of rays spawned at each iteration. Below, we propose a method, using what we call *loop shaders*, to get

[‡] This assumes that shadow rays cannot spawn additional rays.

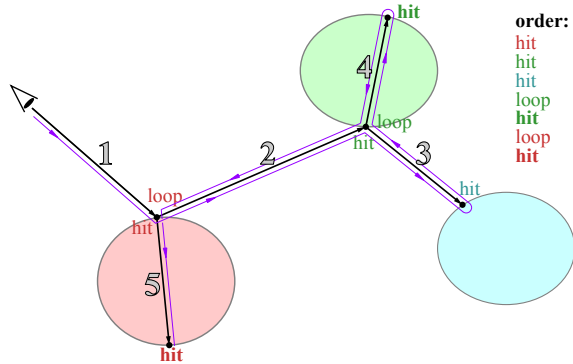


Figure 3: A hit shader is executed with the hit point information each time a ray has been tested against the scene. To support flexible rendering, our ray tracer may optionally execute a loop shader each time a light path ends, that in turn can spawn more rays from a previous hit point. The five rays are shot in the enumerated order. Note that two loop shaders are executed in this example, both after the blue object has been hit and after ray 4 has been processed. The purple path shows the order of execution for this simple ray path.

a similar behavior for ray streams. This method can be entirely implemented within the renderer and does not change the scheduling algorithm.

3.3. Loop Shaders

As previously discussed, the renderer executes a shader over a ray stream after it has been tested against the scene geometry. For the purposes of this section, we divide the shader program into two parts (or subroutines), namely the *hit shader* and the *loop shader*. The hit shader refers to the part that runs over all the rays of the intersection-tested ray stream. It can immediately spawn a single secondary output ray per shaded input ray, by outputting rays to a new ray stream. Path tracing can thus be implemented using only a hit shader, since each hit generates at most a single secondary ray. Generating more than one secondary ray could easily result in the program running out of memory in the general case, as discussed in Section 3.2.

To solve this issue, the loop shader is invoked each time a ray terminates without spawning a secondary ray (i.e., the light path reaches its end). It allows the state of the light path to “loop” back to a previous state in order to send more rays from that point. This is illustrated in Figure 3. A stack of loop shader entries is maintained where the head of the stack is allowed to spawn rays, or perform other computations, on ray termination. This is similar to how the program stack behaves for single-ray recursive ray tracing, with the notable difference that our method works for ray streams. To take Whitted-style ray tracing as an example, instead of spawning two rays at each recursion, a single secondary ray is spawned

and a loop shader entry is added that will spawn another ray on path termination. In the general case, a single loop shader entry can spawn any number of additional rays.

The data that is accessible to the loop shader is arbitrarily specified by the shader that puts the loop shader on the stack. This may consist of information about the hit point, the previous ray, or simply information about a ray to spawn. A loop shader can be initiated both from a hit shader or another loop shader.

In practice, memory for storing loop shader data is allocated using a fast bucket allocator and links between each entry make up a stack. The head of each loop shader stack is stored in the ray payload. When a ray terminates, the head field in the payload is checked to determine if any items were pushed to the loop shader stack. If a stack item exists at the head, the corresponding entry is put in a temporary buffer to be processed using a vectorized kernel. The stack items are then optionally popped. Loop shaders may also terminate a path without triggering a secondary ray, e.g., when determining that adaptive sampling is complete. This may in turn trigger additional stack items to be activated. In these cases, the entire process of executing and activating loop shaders must be performed iteratively until no new loop shaders are activated.

Note that management of loop shaders are entirely done within the renderer’s shader implementation and does not change the core scheduling algorithm. Loop shaders were developed as a way to perform flexible rendering even when working on the granularity of ray streams.

3.4. Life and Death of a Ray Stream in RayAccelerator

In this section, we exemplify our system by describing how a ray stream is first created and what happens to the ray stream over time, up to the time when it is terminated. Our description is in reference to Figure 2. As we have described, RayAccelerator pre-allocates a number of ray streams that together can keep the machine saturated with work. All ray streams are initially in the EMPTY work list, and so when a host thread requests work, it will receive a task that says that it needs to generate eye rays. The host thread thus invokes renderer-specific code that creates eye rays (typically for a rectangular tile of pixels) and puts them in the allotted ray stream (*store eye rays* in Figure 2) in the TRACE work list. At this point, a ray dispatch thread discovers that there is an available ray stream (*get ray query job*), which the thread obtains from the central scheduler and then feeds the ray stream to the graphics processor through OpenCL (*start ray query*). All the rays in the ray stream are then traced through the BVH and intersection tested against the relevant triangles in order to find the closest hit. In case of a hit, it returns the distance and barycentric coordinates. If there is no hit, the kernel instead samples the environment map using the graphics processor’s texture sampler. When all rays

are finished, the result is communicated back to the issuing ray dispatch thread, which then stores a shading job in the SHADE work list (*store shading job*). In RayAccelerator, all shading is done on the CPU cores and shading is the second most important priority (see enumerated list on page 3), so a host thread will obtain a shading job once no more eye rays can be generated (due to memory constraints or because all have been generated). Control then switches to renderer-specific code for shading. Here, the rays in the ray stream are typically sorted based on their hit information (hit/miss and surface material). After that, they are shaded with SIMD processing using AVX2. The shading result is also accumulated to the pixels that the rays belong to. Some of the rays were terminated due to hitting the environment map, and otherwise, new rays (due to reflection, path tracing, etc) may be generated and are appended to a separate ray stream holding new rays. The output ray stream may be put into the NEED-MORE work list, if it is not yet full, or otherwise into TRACE. This goes on until the ray depth threshold is met or until all rays hit the environment map. Note that if a host thread does not have any eye generation work or shading work, then it may request a subset of a trace job as well in order to keep the machine busy with work at all times.

4. Implementation

We implemented our system as an interactive renderer, which supports complex scenes and various BRDFs, including methods such as importance sampling [PH10]. The CPU part of the application was implemented in C++ and optimized using the AVX2 vector instruction set. The kernels running on the graphics processor were implemented using OpenCL 1.2. Using our framework, we have implemented both path tracing and a Whitted-style ray tracer, where the former is a rather simple scheduling problem and the latter the harder one.

All ray streams were allocated in shared memory to allow for efficient communication between CPU and GPU. The system allocates a total of 42 ray streams, each with storage for up to 27k rays and associated hit information. Each ray requires 32 bytes of memory (origin, direction, and near & far distance) and each hit point requires 16 bytes (triangle, distance, and barycentric coordinates). Empirically, we found that 11k rays is enough for efficient tracing on the graphics processor and this was used as the threshold for when a ray stream ends up in the traceable list, described in Section 3. This number likely depends on details of the GPU's scheduling system, such as how terminated threads are substituted for new work, which we have limited insight into. It is worth mentioning that the GPU used for testing (see Section 5) supports up to 8960 active threads, so it appears that a number that is a bit higher than this is beneficial.

Our target architecture (see Section 5) supports 8 CPU hardware threads. It is usually most efficient to allocate one operating system thread per hardware thread because that re-

sults in less scheduling overhead. However, the ray dispatch threads are mostly idle waiting for the GPU to signal completion of work items. In this case, it makes sense to create more operating system threads than there are available hardware threads. Therefore we use 7 host threads and 4 ray dispatch threads, in order to occupy the 8 hardware threads with work. This configuration was also found to be the best performing one. Primary rays are fired from screen tiles as demanded from the scheduler described in Section 3. Each tile consists of 128×128 pixels with 1 sample per pixel, which amounts to 16k rays fired from each tile. We render a single frame at a time and wait until all outstanding rays have been processed before moving on to the next frame. This puts more stress on the scheduler to balance a small amount of work at the end of a frame. Another possibility is to decouple frame display from ray tracing and begin the next frame immediately without waiting. This is simply a matter of wrapping around to the first tile as the end is reached. We are, however, interested in the scheduler's capability to handle difficult scenarios (see Section 5) and leave frame mixing for future work.

As previously described, shading is mostly done on the CPU, while the GPU performs ray queries and additional texture lookups. While the GPU could sample a texture of the material of the closest intersection, we found no general way to pass an arbitrary number of textures with different sizes to OpenCL and access them through a pointer or an index. Texture arrays were found to be insufficient because they require all textures to be of the same size. Bindless textures [SA14] should make this a non-issue, but on our OS/architecture combination, OpenCL 2.0 was not available. Because of these limitations, we only utilize the GPU texture sampler if a ray misses the scene, in which case an environmental light probe is sampled and the resulting sample is returned with the hit information (which indicates no intersection).

Our path tracer performs efficient shading by initially sorting active rays in a ray stream based on surface material, utilizing a radix sort. The rays are then processed 8 at a time using AVX2 instructions, performing attribute interpolation, BRDF evaluation, and sending of secondary rays. The BRDF of the first ray is evaluated for all 8 rays in the batch. Then, it is determined how many rays actually had the same BRDF associated with them, and we only advance those rays in the loop. Because the rays are sorted on material, this results in little overshading.

In GPU ray tracers, each triangle is often stored using 48 bytes (B) [AL09]. However, for three vertices, only 36 B are required and the rest is padding. As a small optimization in the acceleration structure setup, we examine each leaf node and search for triangle pairs sharing an edge. We store these pairs together in 48 B, since the second triangle only requires one additional vertex (12 B). This gave a total speedup of about 4–8%. This idea is somewhat similar to the

work by Amanatides and Choi [AC97]. Much more sophisticated systems have been proposed [LYM07], but we opted for an extremely simple scheme. For triangles not sharing any edges with other triangles, the storage remains at 48 B, so there is no extra memory cost. Currently, we insert a degenerate zero-area triangle for “single” triangles in order to be able to run the same triangle intersection test for both single triangles and shared triangle pairs.

5. Results

All results presented in this section were generated on a Macbook Pro laptop running Mac OS X 10.11. The configuration includes a Haswell Core i7 (4960HQ) with four cores and eight hyperthreads. The chip features programmable thermal design power (TDP) and it ranges from 47 W to 55 W (nominal TDP/cTDP up). The CPU clock frequency is 2.6/3.6 GHz (nominal TDP/cTDP up) for multi-threaded workloads and the single-threaded turbo boost frequency is 3.8 GHz. The graphics processor is integrated on the same die and it has 40 execution units (EUs), whose tasks are all types of shading. This system is also equipped with a 6 MB L3 cache and a 128 MB EDRAM L4 cache, both which are shared between CPU cores and the graphics processor. The raw peak compute power for the graphics processor is 832 Gflops, while it is 332.8/460.8 Gflops (nominal TDP/cTDP up) for the CPU cores. Our system was compiled using Xcode 7.2.1 and Embree 2.7 [WWB*14], which we used to trace rays on the CPU, was compiled with Intel C++ Composer XE for OS X.

Our BVH trees are built using a top-down greedy surface area heuristic (SAH) [MB90] with improvements to sort only once per axis [GBDAM15]. Note that our system can handle dynamic scenes by rebuilding the entire BVH or parts of it, before ray tracing of a frame commences. While this is not the focus of our work, we note that it takes only 10 ms to build the BVH for the BATTLEFIELD scene, for example. To provide fair measurements, we have used monitoring of the CPU temperature, and we let the computer be idle until the CPU temperature is just below 45°C before starting a measurement. This ensures that approximately the same turbo effect, i.e., increasing/decreasing clock frequency, is given to all measurements.

We call our system RayAccelerator, which is described in Section 3 and 4. Four versions of the system, called RA SINGLERAY, RA PACKET, RA SIMD, and RA HYBRID, have been evaluated, in order of increasing sophistication. For RA SINGLERAY, RA PACKET, and RA SIMD, we do not use the graphics processor at all and thus let the CPU cores handle everything, including using Embree to take care of all ray queries. Since each system successively adds features, the performance impact of each improvement can be evaluated. RA SINGLERAY implements the basic scheduling algorithm but does not reap its benefits. A single ray is traced at a time and shading is scalar. This configuration is thus

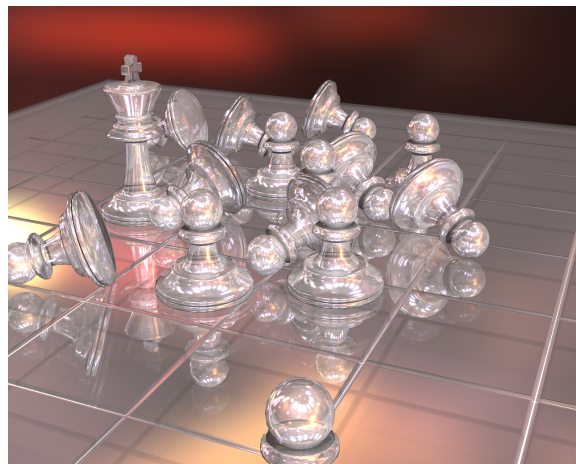


Figure 4: BATTLEFIELD rendered using Whitted-style ray tracing where all surfaces spawn one reflection and one refraction ray at each hit point. The maximum recursion depth is 8 which means that a single eye ray may spawn up to 256 secondary rays. Rendering performance is 37 Mray/s using RA HYBRID.

a good data point for determining the overhead of our system. RA PACKET improves on RA SINGLERAY by tracing 8 rays at a time against the BVH, which often improves performance. RA SIMD adds vectorized shading evaluation to RA PACKET, which improves performance further. RA HYBRID contains all previous features, but also makes use of the GPU for ray queries, but lets the CPU help as necessary as well. The scheduling algorithm for RA SINGLERAY, RA PACKET, and RA SIMD differs somewhat from RA HYBRID in that it instead focuses on shading batches immediately after traversal because there is no GPU that is at risk at starving from work. Instead, it is more beneficial to shade while the ray stream is resident in the CPU cache.

As baseline, we implemented SINGLERAY, which is a traditional single-ray renderer, very much like Embree’s single-ray example path tracer [WWB*14] without using anything ray stream related. When comparing our SINGLERAY renderer with the Embree single-ray example path tracer, the performance was approximately the same (as expected). In contrast to the example renderer provided by Embree, SINGLERAY renders exactly the same images as our RA-counterparts (it shares the same shader code), which makes for better comparisons. Just like the Embree example path tracer, SINGLERAY has limited vectorization opportunities since only a single ray is traced and shaded at a time. We do, however, make use of horizontal SIMD operations whenever possible. The comparison between RA SIMD and SINGLERAY is interesting because it highlights the gains from extracting coherence from ray streams, without offloading traversal to the GPU.

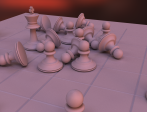





						
	BATTLEFIELD	HAIRBALL	CROWN	DRAGON	BENTLEY	SAN MIGUEL
triangles	64k	2.9M	4.9M	7.3M	2.3M	7.9M
bounces	3	3	20	4	10	4
SINGLERAY	28 Mray/s	13 Mray/s	12 Mray/s	14 Mray/s	16 Mray/s	8.1 Mray/s
RA SINGLERAY	27 Mray/s	13 Mray/s	10 Mray/s	13 Mray/s	15 Mray/s	7.5 Mray/s
RA PACKET	33 Mray/s	13 Mray/s	10 Mray/s	14 Mray/s	16 Mray/s	7.8 Mray/s
RA SIMD	36 Mray/s	14 Mray/s	13 Mray/s	16 Mray/s	18 Mray/s	8.3 Mray/s
RA HYBRID	63 Mray/s	22 Mray/s	19 Mray/s	26 Mray/s	30 Mray/s	14 Mray/s

Table 1: Results in mega rays per second (Mray/s) for our test scenes rendered at 1280×1024 using a laptop with integrated graphics processor. Note that these timings include the entire renderer, i.e., ray queries, shading, render loop, and frame buffer updates. All scenes were rendered with path tracing.

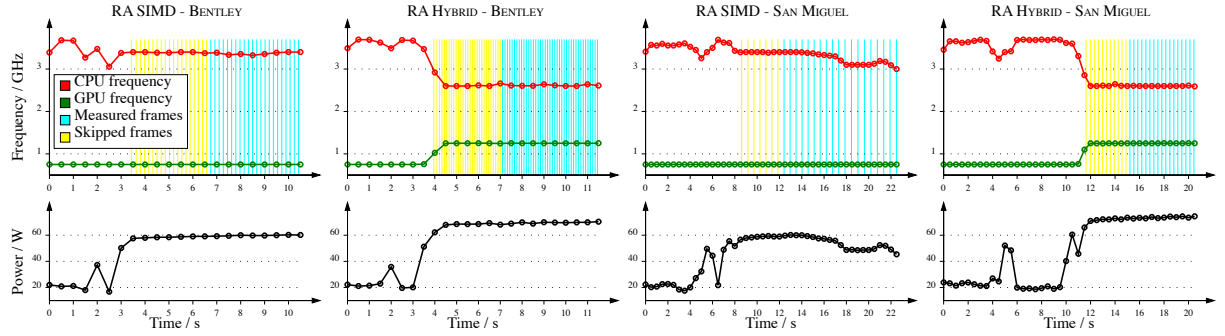


Figure 5: Frequency scaling under load. The two diagrams to the left were measured using BENTLEY, while the two diagrams to the right were measured using SAN MIGUEL. It is interesting to see that for our largest scene (SAN MIGUEL), the frequency of the CPU cores of RA SIMD is decreasing over time, which means that the frequency needs to be scaled down because of heat issues (cTDP up cannot be maintained). For RA HYBRID, the CPU cores are scaled down to about 2.6 GHz and the graphics processor scaled up to about 1.25 GHz. We also include power usage over time (CPU+GPU) as reported by the processor.

Our algorithms were evaluated using six test scenes, namely BATTLEFIELD, HAIRBALL, CROWN, DRAGON, BENTLEY, and SAN MIGUEL, of varying complexity. The major results of the evaluation are shown in Table 1. All images were rendered using *path tracing* [Kaj86], which is a Monte-Carlo technique where only one new ray may be generated at each hitpoint. This algorithm tends to generate incoherent rays after a few bounces. We refer to Table 1 for information on how many bounces were used for our test scenes.

We first note that RA SINGLERAY is fairly close to SINGLERAY, which indicates that our rendering system has low overhead. The overhead seems particularly large for CROWN. This is likely due to a small number of very long light paths within the scene (up to 20 recursions inside the gems), which reduces the ability to batch rays toward the end of the frame. This problem and a likely solution are explored in more detail below. We note that RA PACKET, which adds packet tracing to the system, gives an uneven improvement in performance, ranging from very good for BAT-

TLEFIELD, to no improvement for CROWN. RA SIMD, that also adds vectorized shading, improves all scenes by 6–30%. Interestingly, the improvement is best for CROWN, which benefited poorly from packet tracing. The improvement is the most modest for SAN MIGUEL, which is to be expected since the scene has inexpensive diffuse shading while ray queries are costly because of complex geometry. Comparing RA HYBRID to SINGLERAY, we see generous speedups for all scenes ranging from $1.7\times$ up to $2.3\times$.

Embree also ships with another example renderer that renders packets of rays, which is an alternative method to improve SIMD efficiency in ray tracing. The renderer has in this case been rewritten using ISPC, rather than C++. We setup similar scenes for Embree’s example renderers to see how much performance can be gained from small packet tracing, compared to single-ray. Running all our test scenes using the two renderers, the difference in performance ranged from no difference for the SAN MIGUEL scene to about 24% improved performance with packet tracing for the BATTLEFIELD scene. While it is a bit difficult to

compare absolute numbers because of rendering differences, Table 1 reveals that the relative speedup we get from vectorization using ray streams is similar or better than the gains compared to small packet tracing. When enabling GPU tracing, which is the real goal of our system, our approach is certainly much faster.

Since our target architecture has both CPU cores and an integrated graphics processor, both with programmable TDP, it is interesting to investigate what happens to the frequencies of the CPU and the graphics processor in different configurations of our renderer. In Figure 5, we show our measurements of the frequencies for two scenes and two renderers. Note that we always skip a number of seconds worth of frames, before starting to measure the performance of the renderer, which was done in order to avoid any type of initial turbo effect that does not last. In RA SIMD, the graphics processor is not used, which can be seen in that its frequency is kept low, while the CPU frequency is a bit over 3 GHz. However, for RA HYBRID, which uses both the CPU cores and the graphics processor, we see that the frequency of the graphics processor goes up to about 1.25 GHz, while the CPU cores are scaled down to 2.6 GHz. This makes sense, since the specification says that the max frequency of the graphics processor is 1.3 GHz and the nominal TDP frequency of the CPU cores is 2.6 GHz. These results clearly show that using both CPU and the integrated GPU is challenging since an active integrated GPU lowers overall CPU performance. Still, our system shows that there are benefits to be had when work is carefully balanced between the units.

Figure 5 also includes power usage over time, as reported by the processor. It appears that utilizing both CPU and GPU uses 15–20% more power than using only the CPU at any instance in time. Since performance is increased by more than that, it is tempting to conclude that RA HYBRID is more power efficient. However, we would like to stress that these measurements are very rough. A thorough power analysis using specialized equipment is left for future work.

As a proof of concept of our loop shaders, we have also used our framework to implement a Whitted-style ray tracer (using RA HYBRID) and the image in Figure 4 is a result of that renderer. Even though one reflection and refraction ray are shot at each hit point, the performance remains quite high (37 Mrays/s). The CPU time spent in loop shaders and loop shader data management amounts to about 25% of total shading time. As another proof of concept, we also implemented a procedural wood shader to test what happens when shading becomes more expensive. The result is shown in Figure 6.

We want to emphasize that it is unlikely that porting our entire renderer to the GPU alone would easily result in good performance. Our BRDFs have importance sampling, texturing, and may have several nested layers of simpler BRDFs as well. While this is feasible to port, we expect that perfor-

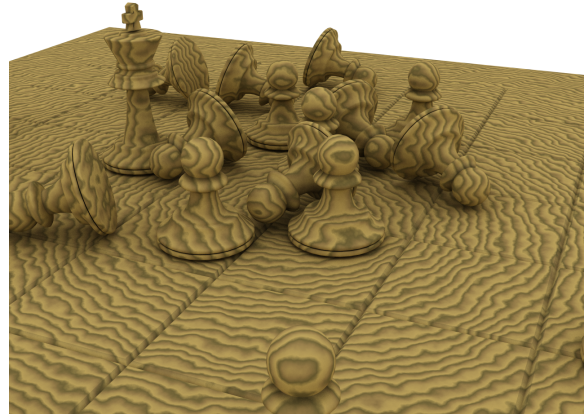


Figure 6: BATTLEFIELD rendered with path tracing and an expensive procedural wood BRDF. The majority of rendering time is spent computing the wood texture. These computations benefit greatly from vectorization. Performance is 4.6, 18, and 27 Mrays/s for SINGLERAY, RA SIMD, and RA HYBRID respectively.

mance will be rather low due to divergence and the number of different BRDFs.

In order to determine scheduling behavior and efficiency, we traced the tasks performed by each CPU thread during the rendering of a frame (using RA HYBRID). We show the results from such a trace for BATTLEFIELD, BENTLEY, and CROWN in Figure 7. As can be seen, the tasks are very tightly packed, which indicates that our system is well balanced. However, it is clear that when rendering a single frame, parallelism/efficiency is reduced towards the end of the timeline. This is especially problematic for CROWN, where long recursions of light paths linger towards the end of the frame. In practice, this can be avoided by keeping multiple frames in flight. This is something that we will explore in future work.

For completeness, we tried disabling two features of RA HYBRID to see how they contribute to overall performance. Disabling GPU light probe lookups, hence performing all lookups on the CPU, resulted in a performance degradation of 0–10%. BATTLEFIELD benefits the most from GPU light probe lookups since they represent a significant portion of the rendering time. For future work, it therefore makes a lot of sense to attempt to do *all* texture lookups on the GPU. Furthermore, we experimented with disallowing the CPU to help with intersection testing when idle. This turned out to impact all scenes with a performance reduction of 20–30%. The reason is likely that the scheduler will have difficulties building large ray streams towards the end of a frame, which makes hybrid intersection testing an important feature of our system.

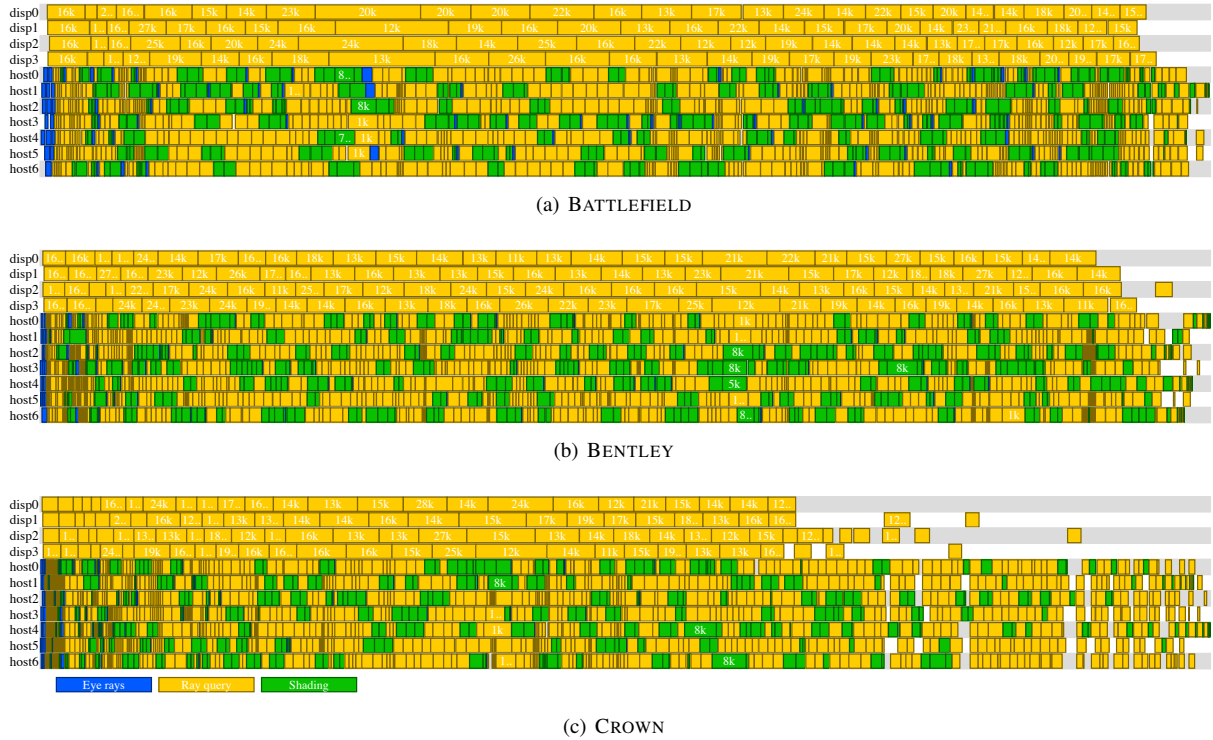


Figure 7: Distribution of work when rendering a single frame of a) BATTLEFIELD, b) BENTLEY, and c) CROWN using RA HYBRID. Tasks are distributed along a timeline for each thread, where time flows from left to right. The top 4 threads represent ray dispatch threads and the bottom 7 are host threads. Different tasks are uniquely colored (eye rays are blue, ray queries yellow, and shading is green) and some tasks indicate the number of rays being processed. It is clear that when ray queries are cheap (BATTLEFIELD), shading (including the render loop) represents a major part of the CPU workload. However, as the cost of ray queries increase, the CPU will load balance by spending more time tracing rays. The CROWN trace showcases what happens when deep light paths are started towards the end of a frame. We believe that these problems can be reduced by immediately starting tracing of the next frame.

6. Discussion and Future Work

We have presented an optimized ray tracing-based rendering system that makes use of CPUs with wide SIMD units and an integrated graphics processor on the same chip by extracting parallelism using ray streams for both traversal and shading. Our novel scheduling system, including the addition of loop shaders, allows for both fast and flexible rendering within the framework. Note that our target platform had $2\times$ more peak compute capabilities on the graphics processor compared to the peak compute power on the CPU cores. However, this does *not* imply that it should be possible to get a $3\times$ speedup when going from using only the CPU to using both the CPU and the GPU, as discussed in Section 1. We have, despite this, developed a ray tracer that is between $1.7\text{--}2.3\times$ faster than SINGLERAY running on the CPU (see Section 5). In the future, we want to explore whether the next frame can overlap with the current in order to close the idle gaps towards the end of a frame. In addition, it would be interesting to gather expensive shading jobs using the same shader and evaluate these on the GPU as well. Another important topic is that of

easy shader authoring while maintaining good performance. To this end, it would be interesting to pursue a vectorizing shader compiler that automatically splits a single recursive shader into one hit shader and possibly multiple loop shaders with minimal state footprint.

Acknowledgements

Tomas is a *Royal Swedish Academy of Sciences Research Fellow*, supported by a grant from the Knut and Alice Wallenberg Foundation. Thanks to Martin Lubich for the Crown model (www.loramel.net). Thanks to Carsten Benthin and Sven Woop for feedback and help with Embree.

References

- [AC97] AMANATIDES J., CHOI K.: Ray Tracing Triangular Meshes. In *Western Computer Graphics Symposium* (1997), pp. 43–52. 8
- [AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics* (2009), pp. 145–149. 2, 3, 7

- [ALK12] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. Tech. Rep. NVR-2012-02, NVIDIA Corporation, June 2012. 2, 3
- [BAGJ08] BUDGE B. C., ANDERSON J. C., GARTH C., JOY K. I.: *A Hybrid CPU-GPU Implementation for Interactive Ray-Tracing of Dynamic Scenes*. Tech. Rep. 9, UC Davis, 2008. 2
- [BAM14] BARRINGER R., AKENINE-MÖLLER T.: Dynamic Ray Stream Traversal. *ACM Transactions on Graphics*, 33, 4 (2014), 151:1–151:9. 3
- [BBS*09] BUDGE B., BERNARDIN T., STUART J. A., SENGUPTA S., JOY K. I., OWENS J. D.: Out-of-core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum*, 28, 2 (2009), 385–396. 2
- [BvS13] BIKKER J., VAN SCHIJNDEL J.: The Brigade Renderer: A Path Tracer for Real-Time Games. *International Journal of Computer Games Technology*, 2013 (2013), 1–14. 2
- [BW*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
- [CHH02] CARR N. A., HALL J. D., HART J. C.: The Ray Engine. In *Graphics Hardware* (2002), pp. 37–46. 2
- [CL07] CHEN C.-C., LIU D. S.-M.: Use of Hardware Z-buffered Rasterization to Accelerate Ray Tracing. In *Symposium on Applied Computing* (2007), pp. 1046–1050. 2
- [DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum*, 27, 4 (2008), 1225–1233. 3
- [DKHS14] DAVIDOVIČ T., KRIVÁNEK J., HAŠAN M., SLUSALLEK P.: Progressive Light Transport Simulation on the GPU: Survey and Improvements. *ACM Transactions on Graphics*, 33, 3 (2014), 29:1–29:19. 3
- [EBA*11] ESMAELZADEH H., BLEM E. R., AMANT R. S., SANKARALINGAM K., BURGER D.: Dark Silicon and the End of Multicore Scaling. In *38th International Symposium on Computer Architecture* (2011), pp. 365–376. 2
- [EG08] ERNST M., GREINER G.: Multi Bounding Volume Hierarchies. In *IEEE Interactive Ray Tracing* (2008), pp. 35–40. 3
- [ENSB13] EISENACHER C., NICHOLS G., SELLE A., BURLEY B.: Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum*, 32, 4 (2013), 125–132. 5
- [GBDAM15] GANESTAM P., BARRINGER R., DOGGETT M., AKENINE-MÖLLER T.: Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees. *Journal of Computer Graphics Techniques*, 4, 3 (September 2015), 23–42. 8
- [HMB*14] HAMMARLUND P., MARTINEZ A., BAJWA A., HILL D., HALLNOR E., JIANG H., DIXON M., DERR M., HUNSAKER M., KUMAR R., OSBORNE R., RAJWAR R., SINGHAL R., D'SA R., CHAPPELL R., KAUSHIK S., CHENNUPATY S., JOURDAN S., GUNTHER S., PIAZZA T., BURTON T.: Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34, 2 (2014), 6–20. 2, 3
- [Ige99] IGEHY H.: Tracing Ray Differentials. In *Proceedings of ACM SIGGRAPH 1999* (1999), pp. 179–186. 5
- [Kaj86] KAJIYA J. T.: The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)* (1986), vol. 20, pp. 143–150. 2, 9
- [LKA13] LAINE S., KARRAS T., AILA T.: Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *High-Performance Graphics* (2013), pp. 137–143. 3
- [LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing. In *IEEE Symposium on Interactive Ray Tracing* (2007), pp. 19–26. 8
- [MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer*, 6, 3 (1990), 153–166. 8
- [MBV*15] MATTAUSCH O., BITTNER J., VILLANUEVA A. J., GOBBETTI E., WIMMER M., PAJAROLA R.: CHC+RT: Coherent Hierarchical Culling for Ray Tracing. *Computer Graphics Forum*, 34, 2 (2015), 537–548. 2
- [NKL*10] NAH J.-H., KANG Y.-S., LEE K.-J., LEE S.-J., HAN T.-D., YANG S.-B.: MobiRT: An Implementation of OpenGL ES-based CPU-GPU Hybrid Ray Tracer for Mobile Devices. In *ACM SIGGRAPH ASIA 2010 Sketches* (2010), pp. 50:1–50:2. 2
- [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*, 29, 4 (2010), 66:1–66:13. 2, 3
- [PBPP11] PAJOT A., BARTHE L., PAULIN M., POULIN P.: Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Computer Graphics Forum*, 30, 2 (2011), 315–324. 2
- [PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*, 2nd ed. MKP, 2010. 7
- [Pia12] PIAZZA T.: Processor Graphics. In *High-Performance Graphics – Hot3D Talks* (June 2012). 3
- [RSB07] ROBERT P. C. D., SCHOEPKE S., BIERI H.: Hybrid Ray Tracing - Ray Tracing using GPU-Accelerated Image-Space Methods. In *GRAPP* (2007), pp. 305–311. 2
- [SA14] SEGAL M., AKELEY K.: *The OpenGL Graphics System: A Specification (version 4.4)*. Tech. rep., 2014. 7
- [SAGC*12] SABINO T., ANDRADE P., GONZALES CLUA E., MONTENEGRO A., PAGLIOSA P.: A Hybrid GPU Rasterized and Ray Traced Rendering Pipeline for Real Time Rendering of Per Pixel Effects. In *Entertainment Computing*, vol. 7522. 2012, pp. 292–305. 2
- [WBB08] WALD I., BENTHIN C., BOULOS S.: Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *IEEE Symposium on Interactive Ray Tracing* (2008), pp. 49–57. 3
- [WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, 3, 1 (1984), 52–69. 2
- [Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23, 6 (1980), 343–349. 2, 5
- [Wil83] WILLIAMS L.: Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)* (1983), vol. 17, pp. 1–11. 3
- [WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20, 3 (2001), 153–164. 3
- [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. 2, 3, 8