Megakernels Considered Harmful: Wavefront Path Tracing on GPUs

Samuli Laine Tero Karras Timo Aila

NVIDIA*

Abstract

When programming for GPUs, simply porting a large CPU program into an equally large GPU kernel is generally not a good approach. Due to SIMT execution model on GPUs, divergence in control flow carries substantial performance penalties, as does high register usage that lessens the latency-hiding capability that is essential for the high-latency, high-bandwidth memory system of a GPU. In this paper, we implement a path tracer on a GPU using a wavefront formulation, avoiding these pitfalls that can be especially prominent when using materials that are expensive to evaluate. We compare our performance against the traditional megakernel approach, and demonstrate that the wavefront formulation is much better suited for real-world use cases where multiple complex materials are present in the scene.

CR Categories: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

Keywords: GPU, path tracing, complex materials

1 Introduction

General-purpose programming on GPUs is nowadays made easy by programming interfaces such as CUDA and OpenCL. These interfaces expose the GPU's execution units to the programmer and allow, e.g., general read/write memory accesses that were severely restricted or missing altogether from the preceding, graphics-specific shading languages. In addition, constructs that assist in parallel programming, such as atomic operations and synchronization points, are available.

The main difference between CPU and GPU programming is the number of threads required for efficient execution. On CPUs that are optimized for low-latency execution, only a handful of simultaneously executing threads are needed for fully utilizing the machine, whereas on GPUs the required number of threads runs in thousands or tens of thousands.[1] Fortunately, in many graphics-related tasks it is easy to split the work into a vast number of independent threads. For example, in path tracing [Kajiya 1986] one typically processes a very large number of paths, and assigning one thread for each path provides plenty of parallelism.

However, even when parallelism is abundant, the execution characteristics of GPUs differ considerably from CPUs. There are two main factors. The first is the SIMT (Single Instruction Multiple Threads) execution model, where many threads (typically 32) are grouped together in *warps* to always run the same instruction. In order to handle irregular control flow, some threads are masked out when executing a branch they should not participate in. This incurs a performance loss, as masked-out threads are not performing useful work.

The second factor is the high-bandwidth, high-latency memory system. The impressive memory bandwidth in modern GPUs comes at the expense of a relatively long delay between making a memory request and getting the result. To hide this latency, GPUs are designed to accommodate many more threads than can be executed in any given clock cycle, so that whenever a group of threads is waiting for a memory request to be served, other threads may be executed. The effectiveness of this mechanism, i.e., the latency-hiding capability, is determined by the threads' resource usage, the most important resource being the number of registers used. Because the register files are of limited size, the more registers a kernel uses, the fewer threads can reside in the GPU, and consequently, the worse the latency-hiding capabilities are.

On a CPU, neither of these two factors is a concern, which is why a naïvely ported large CPU program is almost certain to perform badly on a GPU. Firstly, the control flow divergence that does not harm a scalar CPU thread may cause threads to be severely underutilized when the program is run on a GPU. Secondly, even a single hot spot that uses many registers will drive the resource usage of the entire kernel up, reducing the latency-hiding capabilities. Additionally, the instruction caches on a GPU are much smaller than those on a CPU, and large kernels may easily overrun them. For these reasons, the programmer should be wary of the traditional *megakernel* formulation, where all program code is mashed into one big GPU kernel.

In this paper, we discuss the implementation of a path tracer on a GPU in a way that avoids these pitfalls. Our particular emphasis is on complex, real-world materials that are used in production rendering. These can be almost arbitrarily expensive to evaluate, as the complexity depends on material models constructed by artists who prefer to optimize for visual fidelity instead of rendering performance. This problem has received fairly little attention in the research literature so far. Our solution is a *wavefront* path tracer that keeps a large pool of paths alive at all times, which allows executing the ray casts and the material evaluations in coherent chunks over large sets of rays by splitting the path tracer into multiple specialized kernels. This reduces the control flow divergence, thereby improving SIMT thread utilization, and also prevents resource usage hot spots from dominating the latency-hiding capability for the whole program. In particular, ray casts that consume a major portion of execution time can be executed using highly optimized, lean kernels that require few registers, without being polluted by high register usage in, e.g., material evaluators.

Pre-sorting work in order to improve execution coherence is a well-known optimization for traditional feed-forward rendering, where the input geometry can be easily partitioned according to, e.g., the

*e-mail: {slaine,tkarras,taila}@nvidia.com

[1]If the CPU is programmed as a SIMT machine using, e.g., the `ispc` compiler [Pharr and Mark 2012], the number of threads is effectively multiplied by SIMD width. For example, a hyperthreading 8-core Intel processor with AVX SIMD extensions can accommodate 128 resident threads with completely vectorized code. In contrast, the NVIDIA Tesla K20 GPU used for benchmarks in this paper can accommodate up to 26624 resident threads.

fragment shader program used by each triangle. This lets each shader to be executed over a large batch of fragments, which is more efficient than changing the shader frequently. In path tracing the situation is trickier, because it cannot be known in advance which materials the path segments will hit. Similarly, before the material code has been executed it is unclear whether the path should be continued or terminated. Therefore, the sorting of work needs to happen on the fly, and we achieve this through queues that track which paths should be processed by each kernel.

We demonstrate the benefits of the wavefront formulation by comparing its performance against the traditional megakernel approach. We strive to make a fair comparison, and achieve this by having both variants thoroughly optimized and encompassing essentially the same code, so that the only differences are in the organization of the programs.

2 Previous Work

Purcell et al. [2002] examined ray tracing on early programmable graphics hardware. As the exact semantics of the hardware that was then still under development were unknown, they considered two architectures: one that allows conditional branching and loop structures, and one without support for them. In the former case, the kernels were combined into a single program which allowed for shorter overall code. In the latter case, a multipass strategy was used with multiple separate kernels for implementing the loops necessary for ray casts and path tracing. The splitting of code into multiple kernels was performed only to work around architectural limitations.

OptiX [Parker et al. 2010] is the first general-purpose GPU ray tracing engine supporting arbitrary material code supplied by the user. In the implementation presented in the paper, all of the ray cast code, material code, and other user-specified logic is compiled into a single megakernel. Each thread has a state specifying which block of code (e.g., ray-box intersection, ray-primitive intersection, etc.) it wishes to execute next, and a heuristic scheduler picks the block to be executed based on these requests [Robison 2009].

Because each task, e.g., a path in a path tracer, is permanently confined to a single thread, the scheduler cannot combine requests over a larger pool of threads than those in a single group of 32 threads. If, for example, each path wishes to evaluate a different material next, the scheduler has no other choice but to execute them sequentially with only one active thread at a time. However, as noted by Parker et al. [2010], the OptiX execution model does not prescribe an execution order of individual tasks or between pieces of code in different tasks, and it could therefore be implemented using a streaming approach with a similar rewrite pass that was used for generating the megakernel.

Van Antwerpen [2011] describes methods for efficient GPU execution of various light transport algorithms, including standard path tracing [Kajiya 1986], bi-directional path tracing [Lafortune and Willems 1993; Veach and Guibas 1994] and primary sample-space Metropolis light transport [Kelemen et al. 2002]. Similar to our work, paths are extended one segment at a time, and individual streams for paths to be extended and paths to be restarted are formed through stream compaction. In the more complex light transport algorithms, the connections between path vertices are evaluated in parallel, avoiding the control flow divergence arising from some paths having to evaluate more connections than others. In contrast to our work, the efficient handling of materials is explicitly left out of scope.

Path regeneration was first introduced by Novák et al. [2010], and further examined with the addition of stream compaction by Wald [2011], who concluded that terminated threads in a warp incur no major performance penalties due to the remaining threads executing faster. Efficient handling of materials was not considered, and only simple materials were used in the tests. Our results indicate that—at least with more complex materials—the compaction of work can have substantial performance benefits.

Hoberock et al. [2009] use stream compaction before material evaluation in order to sort the requests according to material type, and examine various scheduling heuristics for executing the material code. Splitting distinct materials into separate kernels, or separating the ray cast kernels from the rest of the path tracer is not discussed. Due to the design, performance benefits are reported to diminish as the number of materials in the scene increases. In our formulation, individual materials are separated to their own kernels, and compaction is performed implicitly through queues, making our performance practically independent of the number of materials as long as enough rays hit each material to allow efficient bulk execution.

Performing fast ray casts on GPU, and constructing efficient acceleration hierarchies for this purpose, have been studied more extensively than the execution of full light transport algorithms, but these topics are both outside the scope of our paper. Our path tracer utilizes the ray cast kernels of Aila et al. [2009; 2012] unmodified, and the acceleration hierarchies are built using the SBVH algorithm [Stich et al. 2009].

3 Complex Materials

The materials commonly used in production rendering are composed of multiple BSDF layers. The purpose of the material code, generated by the artist either programmatically or through tools, is to output a stack of BSDFs when given a surface point. The possible BSDFs are supplied by the underlying renderer, and typically cannot be directly modified. This ensures that the renderer is able to evaluate extension directions, light connection weights, sampling probabilities, etc., as required by the light transport algorithm used.

While the individual BSDFs are generally not overly complicated to evaluate, the process of producing the BSDF stack can be arbitrarily expensive. Common operations in the material code include texture coordinate calculations, texture evaluations, procedural noise evaluations, or even ray marching in a mesostructure.

Figure 1 shows a closeup rendering of a relatively simple four-layer car paint material derived from one contained in Bunkspeed, a commercial rendering suite. The bottom layer is a Fresnel-weighted diffuse layer where the albedo depends on the angles of incoming and outgoing rays, producing a reddish tint at grazing angles. On top of the base layer there are two flake layers with procedurally generated weights and normals. The BSDF of the flakes is a standard Blinn-Phong BSDF with proper normalization to ensure energy conservation. The top layer is a Fresnel-weighted coat layer with mirror BSDF.

A major part of code related to this material is the evaluation of the procedural noise functions for the flake layers. Two noise evaluations are required per layer: the first noise perturbs the given surface position slightly, and this perturbed position is then quantized and used as an input to the second noise evaluation to obtain flake weight and normal. To produce two flake layers, four noise evaluations are therefore required in total. The proprietary noise evaluation function consists of 80 lines of C++ code compiling to 477 assembly instructions on an NVIDIA Kepler GPU. When combining the construction of the BSDF stack, evaluating the resulting BSDFs, performing importance sampling, etc., the total amount of code needed for evaluating the material amounts to approximately 4200 assembly instructions.

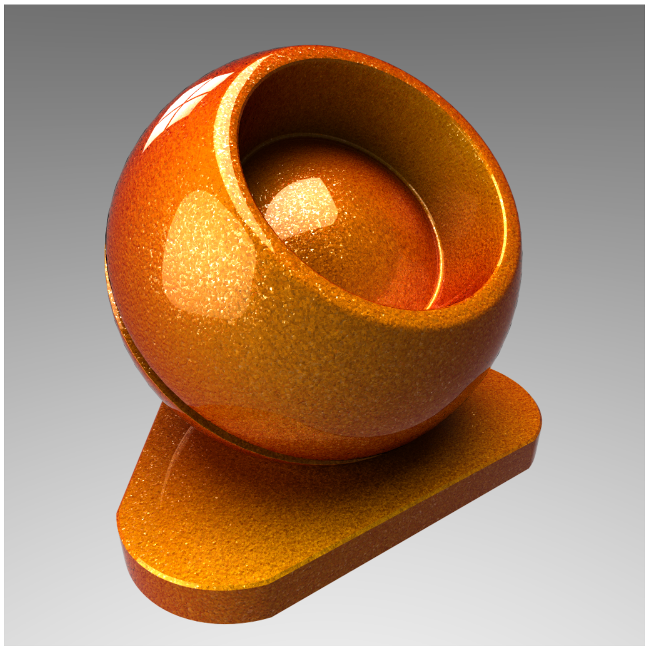


Figure 1: *A closeup of a four-layer car paint material with procedural glossy flakes, rendered using our path tracer. See Section 3 for details.*

It should be noted that four noise evaluations is a relatively modest number compared to multi-octave gradient noise required in, e.g., procedural stone materials. Also, further layers for dirt, decals, etc. could be added on top of the car paint, each with their own BSDFs. The takeaway from this example is that material evaluations can be very expensive compared to other work done during rendering, and hence executing them efficiently is highly important. For reference, casting a path extension ray in the conference room scene (Figure 3, right) executes merely 2000–3000 assembly instructions.

4 Wavefront Path Tracing

In order to avoid a lengthy discussion of preliminaries, we assume basic knowledge of the structure of a modern path tracer. Many publicly available implementations exist, including PBRT [Pharr and Humphreys 2010], Mitsuba [Jakob 2010], and Embree [Ernst and Woop 2011]. We begin by discussing some of the specifics of our path tracer and in Section 4.1 analyze the weaknesses of the megakernel variant. Our wavefront formulation is described in Section 4.2, followed by optimizations and implementation details related to memory layout of path state and queue management.

In our path tracer, light sources can be either directly sampled (e.g., area lights or distant angular light sources like the sun), or not (e.g., "bland" environment maps, extremely large area lights), as specified in the scene data. A light sample is generated out of the directly sampled light sources, and a shadow ray is cast between the path vertex and light sample. Multiple importance sampling (MIS) [Veach and Guibas 1995] with the power heuristic is used for calculating the weights of the extended path and the explicit light connection, which requires knowing the probability density of the light sampler at the extension ray direction, and vice versa, in addition to the usual sampling probabilities.

Russian roulette is employed for avoiding arbitrarily long paths. In our tests, the roulette starts after eight path segments, and the continuation probability is set to path throughput clamped to 0.95, as in the Mitsuba renderer [Jakob 2010].

The material evaluator produces the following outputs when given a surface point, outgoing direction (towards the camera), and light sample direction:

- importance sampled incoming direction,
- value of the importance sampling pdf,
- throughput between incoming and outgoing directions,
- throughput between light sample direction and outgoing direction,
- probability of producing the light sample direction when sampling incoming direction (for MIS), and
- medium identifier in the incoming direction.

For generating low-discrepancy quasirandom numbers needed in the samplers, we use Sobol sequences [Joe and Kuo 2008] for the first 32 dimensions, and after that revert to purely random numbers generated by hashing together pixel index, path index, and dimension. The Sobol sequences for the first 32 dimensions are precomputed on the CPU and shared between all pixels in the image. In addition, each pixel has an individual randomly generated scramble mask for each Sobol dimension that is XORed together with the Sobol sequence value, ensuring that each pixel's paths are well distributed in the path space but uncorrelated with other pixels. Generating a quasirandom number on the GPU therefore involves only two array lookups, one from the Sobol sequence buffer and one from the scramble value buffer, and XORing these together. Because the Sobol sequences are shared between pixels, the CPU only has to evaluate a new index in each of the 32 Sobol dimensions between every N paths, where N is the number of pixels in the image, making this cost negligible.

4.1 Analysis of a Megakernel Path Tracer

Our baseline implementation is a traditional megakernel path tracer that serves both as a correctness reference and as a performance comparison point. The megakernel always processes a batch of paths to completion, and includes path generation, light sampling, ray casters for both extension rays and shadow rays, all material evaluation code, and general path tracing logic. Path state is kept in local variables at all times.

There are three main points where control flow divergence occurs. The first and most obvious is that paths may terminate at different lengths, and terminated paths leave threads idling until all threads in the 32-wide warp have been terminated. This can be alleviated by dynamically regenerating paths in the place of terminated ones. Path regeneration is not without costs, however. Initializing path state and generating the camera ray are not completely negligible pieces of code, and if regeneration is done too often, these are run at low thread utilization. More importantly, path regeneration decreases the coherence in the paths being processed by neighboring threads. Some 1–5% improvement was obtained by regenerating paths whenever more than half of the threads in a warp are idling, and this optimization is used in the benchmarks.

The second major control flow divergence occurs at the material evaluation. When paths in a warp hit different materials, the execution is serialized over all materials involved. According to our tests, this is the main source of performance loss in scenes with multiple complex materials.

The third source of divergence is a little subtler. For materials where the composite BSDF (comprising all layers in the BSDF stack) is discrete, i.e., consists solely of Dirac functionals, it makes no sense to cast the shadow ray to the light sample because the throughput between light sample direction and outgoing direction is always zero. This happens only for materials such as glass and mirror, but in scenes with many such materials the decrease in the number of required shadow rays may be substantial.

Another drawback of the megakernel formulation is the high register usage necessitated by hot spots in the material code where many registers are consumed in, e.g., noise evaluations and math in the BSDF evaluations. This decreases the number of threads that can remain resident in the GPU, and thereby hurts the latency hiding capability. Ray casts suffer from this especially badly, as they perform relatively many memory accesses compared to math operations.

Finally, the instruction caches on a GPU, while being adequate for moderately sized or tightly looping kernels such as ray casts, cannot accommodate the entire megakernel. Because the instruction caches are shared among all warps running in the same streaming multiprocessor (SM), a highly divergent, large kernel that executes different parts of code in different warps is likely to overrun the cache.

4.2 Wavefront Formulation

Our wavefront path tracer formulation is based on keeping a pool of 1M ($= 2^{20}$) paths alive at all times. On each iteration, every path is advanced by one segment, and if a path is terminated, it is regenerated during the same iteration. Path state is stored in global memory on the GPU board (DRAM), and consumes 212 bytes per path, including extension and shadow rays and space for the results of ray casts. The total path state therefore consumes 212 MB of memory. If higher memory usage is allowed, a slight performance increase can be obtained by enlarging the pool size ($\sim$5% when going from 1M to 8M paths consuming 1.7 GB). However, as a high memory consumption is usually undesirable, all of our tests are run with the aforementioned pool size of 1M paths.

The computation is divided into three *stages*: logic stage, material stage, and ray cast stage. We chose not to split the light sampling and evaluation into a separate stage, as light sources that are complex enough to warrant having an individual stage are not as common as complex materials. However, should the need arise, such separation would be easy to carry out. Each stage consists of one or multiple individual kernels. Figure 2 illustrates the design.

Communication between stages is carried out through the path state stored in global memory, and *queues* that are similarly located in global memory. Each kernel that is not executed for all paths in the pool has an associated queue that is filled with requests by the preceding stage. The logic kernel, comprising the first stage, does not require a queue because it always operates on all paths in the pool. The queues are of fixed maximum size and they are preallocated in GPU memory. The memory consumption of each queue is 4 MB.

Logic stage The first stage contains a single kernel, the logic kernel, whose task is to advance the path by one segment. Material evaluations and ray casts related to the previous segment have been performed during the previous iteration by the subsequent stages. In short, the logic kernel performs all tasks required for path tracing besides the material evaluations and ray casts. These include:

- calculating MIS weights for light and extension segments,
- updating throughput of extended path,
- accumulating light sample contribution in the path radiance if the shadow ray was not blocked,
- determining if path should be terminated, due to
  - extension ray leaving the scene,
  - path throughput falling to zero, or
  - Russian roulette,
- for a terminated path, accumulating pixel value,

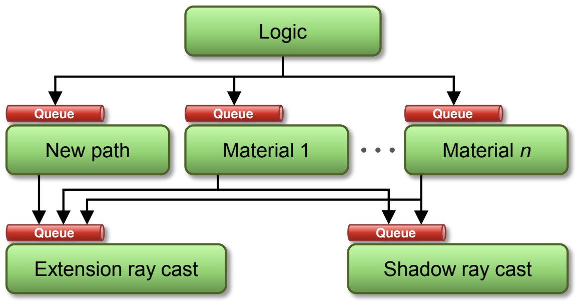


Figure 2: *The design of our wavefront path tracer. Each green rectangle represents an individual kernel, and the arrows indicate queue writes performed by kernels. See Section 4.2 for details.*

- producing a light sample for the next path segment,
- determining material at extension ray hit point, and
- placing a material evaluation request for the following stage.

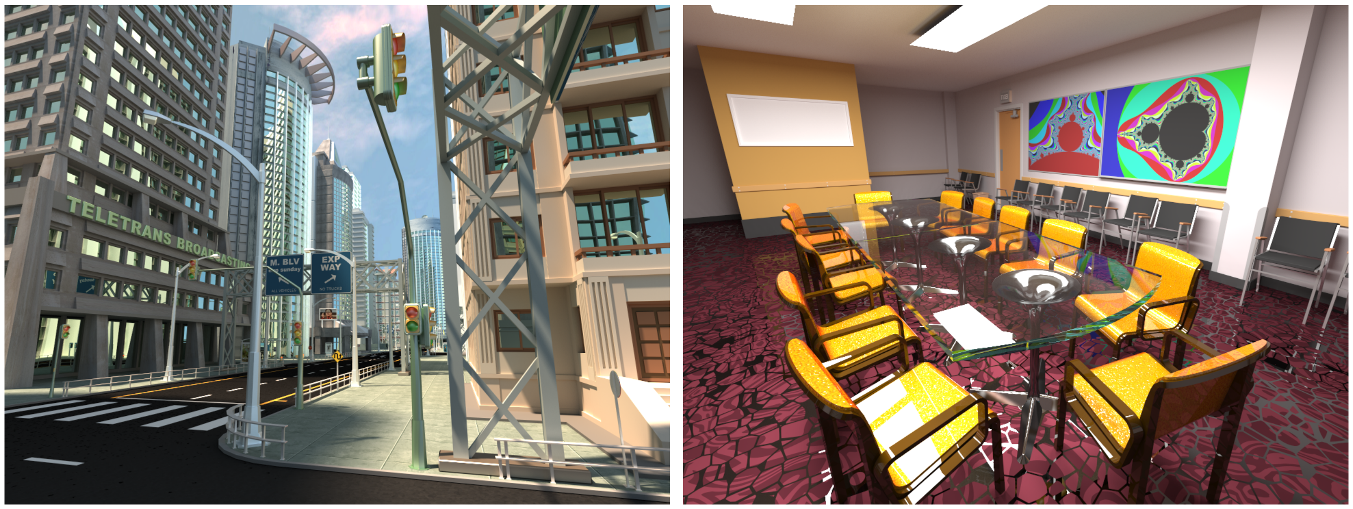
As illustrated in Figure 2, we treat the generation of a new path in the same fashion as evaluating a material. This is a natural place for this operation, because as for materials, we want to cast an extension ray (the camera ray) right afterwards, and cannot perform any other path tracing logic before this ray cast has been completed.

Material stage After the logic stage, each path in the pool is either terminated or needs to evaluate the material at extension ray hit point. For terminated paths, the logic kernel has placed a request into the queue of the new path kernel that initializes path state and generates a camera ray. This camera ray is placed into the extension ray cast queue by the new path kernel. For non-terminated paths, we have multiple material kernels whose responsibilities were listed in Section 4.

Each material present in the scene is assigned to one of the material kernels. In a megakernel-like assignment, all materials would go into the same kernel that chooses the relevant piece of code using a switch-case statement. In the opposite extreme, every material could have its own kernel in this stage. The former option has the control flow divergence problem that we are trying to avoid, so this is clearly not viable. The latter option has overheads with materials that are cheap to evaluate, because kernel launches and managing multiple queues have nonzero costs. In practice, we place each “expensive” material into its own material kernel, and combine the “simple” materials into one kernel. This choice is currently done by hand, but automated assigment could be done, e.g., based on the amount of code in the individual material evaluators. It is not obvious that this is the best strategy, and optimizing the assignment of materials into kernels is an interesting open problem requiring more detailed analysis of the costs associated with control flow divergence versus kernel switching.

The kernels in the material stage place ray cast requests for the following ray cast stage. The new path kernel always generates an extension ray but never a shadow ray. In the common case, materials generate both an extension ray and a shadow ray, but some materials such as mirrors and dielectrics may choose not to generate the shadow ray, as mentioned above. It is also possible that extension ray generation fails (e.g., glossy reflection direction falling below horizon), in which case extension ray is not generated and the path is flagged for termination by setting its throughput to zero.

Ray cast stage In this stage, the collected extension and shadow rays are cast using the ray cast kernels from Aila et al. [2009; 2012].



CITY CONFERENCE

Figure 3: *Two of the test scenes used in evaluating the performance of the wavefront path tracer.*

The kernels place results into result buffers at indices corresponding to the requests in the input buffers. Therefore, the path state has to record the indices in the ray buffers in order to enable fetching the results in the logic stage.

4.3 Memory Layout

The main drawback of the wavefront formulation compared to the megakernel is that path state has to be kept in memory instead of local registers. However, we argue that with a suitable memory layout this is not a serious problem.

The majority of the path state is accessed in the logic kernel that always operates on all paths. Therefore, the threads in a warp in the logic kernel operate on paths with consecutive indices in the pool. By employing a structure-of-arrays (SOA) memory layout, each access to a path state variable in the logic kernel results in a contiguous read/write of 32 32-bit memory words, aligned to a 1024-bit boundary. The GPU memory architecture is extremely efficient for these kinds of memory accesses. In the other kernels, the threads do not necessarily operate on consecutive paths, but memory locality is still greatly improved by the SOA layout.

When rendering Figure 1, the SOA memory layout provides a total speedup of 80% over the simpler array-of-structures (AOS) layout. The logic kernel speedup is 147%, new path kernel speedup is a whopping 790% (presumably due to high number of memory writes), and material kernel speedup is 68%. The ray cast time is not affected by the memory layout, as the ray cast kernels do not access path state.

4.4 Queues

By producing compact queues of requests for the material and ray cast stages, we ensure that each launched kernel always has useful work to perform on all threads of a warp. Our queues are simple preallocated global memory buffers sized so that they can contain indices of every path in the pool. Each queue has an item counter in global memory that is increased atomically when writing to the queue. Clearing a queue is achieved by setting the item counter to zero.

At queue writes, it would be possible for each thread in a warp to individually perform the atomic increment and the memory write, but this has two drawbacks. First, the individual atomics are not coalesced, so increments to the same counter are serialized which hurts performance. Second, the individual atomics from different warps become intermixed in their execution order. While this does not affect the correctness of the results, it results in decreased coherence. For example, if the threads in a logic kernel warp all have paths hitting the same material, placing each of them individually in the corresponding material queue does not ensure that they end up in consecutive queue entries, as other warps can push to the queue between them.

To alleviate this, we coalesce the atomic operations programmatically within each warp prior to performing the atomic operations. This can be done efficiently using warp-wide ballot operations where each thread sets a bit in a mask based on a predicate, and this mask is communicated to every thread in the warp in one cycle. The speedup provided by atomic coalescing is 40% in the total rendering speed of Figure 1. The logic kernel speedup is 75%, new path kernel speedup is 240%, and material kernel speedup is 35%. The effect of improved coherence is witnessed by the speedup of ray cast kernels by 32%, which can be attributed entirely to improved ray coherence.

5 Results

We analyze the performance of the wavefront path tracer in three test scenes. Real-world test data is hard to integrate to an experimental renderer, so we have attempted to construct scenes and materials with workloads that could resemble actual production rendering tasks. Instead of judging the materials by their looks, we wish to focus our attention to their composition, detailed below.

The simplest test scene, CARPAINT (Figure 1), contains a geometrically simple object with the four-layer car paint material (Section 3), illuminated by an HDR environment map. This scene is included in order to illustrate that the overheads of storing the path state in GPU memory do not outweigh the benefits of having specialized kernels even in cases where just a single material is present in the scene.

The second test scene, CITY (Figure 3, left), is of moderate geometric complexity (879K triangles) and has three complex materials.

The asphalt is made of repurposed car paint material with adjusted flake sizes and colors. The sidewalk is a diffuse material with a tiled texture. We have added procedural noise-based texture displacement in order to make the appearance of each tile different. Finally, the windows are tinted mirrors with low-frequency noise added to normals, producing the wobbly look caused by slight nonplanarity of physical glass panes. The rest of the materials are simple diffuse or diffuse+glossy surfaces with optional textures. The scene is illuminated by a HDR environment map of the sky without sun, and an angular distant light source representing the sun.

The third test scene, CONFERENCE (Figure 3, right) has 283K triangles and also contains three expensive materials. The yellow chairs are made of the four-layer car paint material, and the floor features a procedural Voronoi cell approximation that controls the reflective coating layer. The base layer also switches between two diffuse colors based on single-octave procedural noise. The Mandelbrot fractals on the wall are calculated procedurally, acting as a proxy for a complex iterative material evaluator. A more realistic situation where such iteration might be necessary is, e.g, ray marching in a mesosurface for furry or displaced surfaces. The rest of the materials are simple dielectrics (table), or two-layer diffuse+glossy materials. The scene is illuminated by the two quadrilateral area light sources on the ceiling.

Our test images are rendered in 1024×1024 (CARPAINT) and 1024×768 resolution (CITY, CONFERENCE) on an NVIDIA Tesla K20 board containing a GK110 Kepler GPU and 5 GB of memory. For performance measurements, the wavefront path tracer was run until the execution time had stabilized due to path mixture reaching a stationary distribution—in the beginning, the performance is higher due to startup coherence. The megakernel batch size was set to 1M paths, and path regeneration was enabled as it yielded a small performance benefit. Both path tracer variants contain essentially the same code, and the differences in performance are only due to the different organization of the computation.

Table 1 shows the performance of the baseline megakernel and our wavefront path tracer. Notably, even in the otherwise very simple CARPAINT scene, we obtain a 36% speedup by employing separate logic, new path, material, and ray cast kernels. The overhead of storing the path state in GPU memory is more than compensated for by the faster ray casts enabled by running the ray cast kernels with low register counts and hence better latency hiding capability, while the entire ray cast code fits comfortably in the instruction caches. For the other two test scenes with several materials, our speedups are even higher. Especially in the CONFERENCE scene, the traditional megakernel suffers greatly from the control flow divergence in the material evaluation phase, exacerbated by highly variable evaluation costs of different materials. Analysis with NVIDIA Nsight profiler reveals that in this scene the thread utilization of the megakernel is only 23%, whereas the wavefront variant has 53% overall thread utilization (60% in logic, 99% in new path generation, 71% in materials, lowered by variable iteration count in Mandelbrot shader, and 35% in ray casts). The ray cast kernel utilization is lower than the numbers reported by Aila and Laine [2009] for two reasons. First, our rays are not sorted in any fashion, whereas in the previous work they were assigned to threads in a Morton-sorted order. Second, the rays produced during path tracing are even less coherent than the first-bounce diffuse interreflection rays used in the previous measurements.

Table 2 shows the execution time breakdown for the wavefront path tracer in each of the test scenes. It is apparent that the ray casts still constitute a major portion of the rendering time: 44% in CARPAINT, 56% in CITY, and 49% in CONFERENCE. However, in every scene approximately half of the overall rendering time is spent in path tracing related calculations and material evaluations, validating the concern that fast ray casts do not alone ensure good performance. The time spent in ray casts is largely unaffected by the materials in the scene, and conversely, the material evaluation time is independent on the geometric complexity of the scene. As the materials in the scenes are arguably still not of real-world complexity, we can expect the relative cost of materials to increase, further stressing the importance of their efficient evaluation. Another interesting finding is the relatively high cost of new path generation compared to other path tracing logic, which favors separating it into a separate kernel for compact execution so that all threads can perform useful work.

scene	#tris	performance (Mpaths/s)		**speedup**
		megakernel	wavefront	
CARPAINT	9.5K	42.99	58.38	36%
CITY	879K	5.41	9.70	79%
CONFERENCE	283K	2.71	8.71	221%

Table 1: *Path tracing performance of the megakernel path tracer and our wavefront path tracer, measured in millions of completed paths per second.*

scene	logic	new path	materials	ray cast
CARPAINT	2.40	0.86	2.31	4.31
CITY	3.42	0.86	5.47	12.53
CONFERENCE	3.01	0.79	6.37	9.62

Table 2: *Execution time breakdown for one iteration (1M path segments) of the wavefront path tracer. All timings are in milliseconds.*

6 Conclusions and Future Work

Our results show that decomposing a path tracer into multiple specialized kernels is a fruitful strategy for executing it on a GPU. While there are overheads associated with storing path data in memory between kernel launches, management of the queues, and launching the kernels, these are well outweighed by the benefits. Although all of our tests were run on NVIDIA hardware, we expect similar gains to be achievable on other vendors' GPUs as well due to architectural similarities.

Following the work of van Antwerpen [2011], augmenting more complex rendering algorithms such as bi-directional path tracing [Lafortune and Willems 1993; Veach and Guibas 1994] and Metropolis light transport [Veach and Guibas 1997; Kelemen et al. 2002] with a multikernel material evaluation stage is an interesting avenue for future research. In some predictive rendering tasks, the light sources may also be very complex (e.g., [Kniep et al. 2009]), and a similar splitting into separate evaluation kernels might be warranted. Monte Carlo rendering of participating media (see, e.g., [Raab et al. 2008]) is another case where the execution, consisting of short steps in a possibly separate data structure, differs considerably from the rest of the computation, suggesting that a specialized volume marching kernel would be advantageous.

On a more general level, our results provide further validation to the notion that GPU programs should be approached differently from their CPU counterparts, where monolithic code has always been the norm. In the case of path tracing, finding a natural decomposition could be based on the easily identifiable pain points that clash with the GPU execution model, and this general approach is equally applicable to other application domains as well. Similar issues with control flow divergence and variable execution characteristics are likely to be found in any larger-scale program, and we expect our analysis to give researchers as well as production programmers valuable insights on how to deal with them and utilize more of the computational power offered by the GPUs.

Interestingly, programming CPUs as SIMT machines has recently gained traction, partially thanks to the release of Intel's `ispc` compiler [Pharr and Mark 2012] that allows easy parallelization of scalar code over multiple SIMD lanes. CPU programs written this way suffer from control flow divergence just like GPU programs do, albeit perhaps to a lesser degree due to narrower SIMD. Using our techniques for improving the execution coherence should therefore be useful in this regime as well.

Acknowledgments

We thank Carsten Wächter and Matthias Raab for illuminating discussions about materials used in professional production rendering. Original CONFERENCE scene geometry by Anat Grynberg and Greg Ward.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics*, 145–149.

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. Tech. Rep. NVR-2012-02, NVIDIA.

ERNST, M., AND WOOP, S., 2011. Embree: Photo-realistic ray tracing kernels. White paper, Intel.

HOBEROCK, J., LU, V., JIA, Y., AND HART, J. C. 2009. Stream compaction for deferred shading. In *Proc. High Performance Graphics*, 173–180.

JAKOB, W., 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

JOE, S., AND KUO, F. Y. 2008. Constructing Sobol sequences with better two-dimensional projections. *SIAM J. Sci. Comput. 30*, 2635–2654.

KAJIYA, J. T. 1986. The rendering equation. In *Proc. ACM SIGGRAPH 86*, 143–150.

KELEMEN, C., SZIRMAY-KALOS, L., ANTAL, G., AND CSONKA, F. 2002. A simple and robust mutation strategy for the Metropolis light transport algorithm. *Comput. Graph. Forum 21*, 3, 531–540.

KNIEP, S., HÄRING, S., AND MAGNOR, M. 2009. Efficient and accurate rendering of complex light sources. *Comput. Graph. Forum 28*, 4, 1073–1081.

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1993. Bi-directional path tracing. In *Proc. Compugraphics*, 145–153.

NOVÁK, J., HAVRAN, V., AND DASCHBACHER, C. 2010. Path regeneration for interactive path tracing. In *Eurographics 2007, short papers*, 61–64.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4, 66:1–66:13.

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering, 2nd ed.* Morgan Kaufmann.

PHARR, M., AND MARK, W. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Proc. InPar 2012*, 1–13.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph. 21*, 3, 703–712.

RAAB, M., SEIBERT, D., AND KELLER, A. 2008. Unbiased global illumination with participating media. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*. 591–605.

ROBISON, A. 2009. Hot3D talk: Scheduling in NVIRT. HPG '09, http://www.highperformancegraphics.org/previous/www_2009/presentations/nvidia-rt.pdf.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proc. High Performance Graphics*, 7–13.

VAN ANTWERPEN, D. 2011. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proc. High Performance Graphics*, 41–50.

VEACH, E., AND GUIBAS, L. 1994. Bidirectional estimators for light transport. In *Proc. Eurographics Rendering Workshop*, 147–162.

VEACH, E., AND GUIBAS, L. J. 1995. Optimally combining sampling techniques for Monte Carlo rendering. In *Proc. ACM SIGGRAPH 95*, 419–428.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proc. ACM SIGGRAPH 97*, 65–76.

WALD, I. 2011. Active thread compaction for GPU path tracing. In *Proc. High Performance Graphics*, 51–58.