Active Thread Compaction for GPU Path Tracing

Ingo Wald*
Intel

Abstract

Modern GPUs like NVidia's Fermi internally operate in a SIMD manner by ganging multiple (32) scalar *threads* together into SIMD *warps*; if a warp's threads diverge, the warp serially executes both branches, temporarily disabling threads that are not on that path. In this paper, we explore and thoroughly analyze the concept of *active thread compaction*—i.e., the process of taking multiple partially-filled warps and compacting them to fewer but fully utilized warps—in the context of a CUDA path tracer. Our results show that this technique can indeed lead to significant improvements in SIMD utilization, and corresponding savings in the amount of work performed; however, they also show that certain inadequacies of today's hardware wipe out most of the achieved gains, leaving bottom-up speed-ups of a mere 12–16%. We believe our analysis of *why* this is the case will provide insight to other researchers experimenting with this technique in different contexts.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

1 Introduction

Today's programmable GPUs get their compute power by running in the order of thousands of threads in parallel, and the "SIMT" programming paradigm [NVidia b] used by CUDA and OpenCL allow the programmer to treat these threads just as if they were scalar threads. The underlying hardware, however, has strong SIMD traits: each CUDA thread runs on its own scalar processor, but scalar processors are *ganged together* into multiprocessors that execute *warps* of (32) threads in SIMD fashion. If a warp's threads diverge in control flow, the warp serially executes both branches, temporarily disabling threads that are not on that path, and reconverging to the common execution path once both branches are done. During each such divergence, part of the warp's threads are in-active, which means that the underlying (SIMD-)hardware is but partially utilized for the duration of each divergent branch. Thus, re-sorting threads into warps of similar control flow should boost performance, and at least for shader executions Hoberock et al. [2009] have shown that this is indeed the case.

One particular form of this SIMD divergence—that we will concentrate on in this paper—is when a warp's threads leave a given kernel (or a sub-routine of that kernel) at different points in time: for example, by performing a varying number of traversal steps in a ray tracing kernel, by tracing paths of different lengths in a path tracer, or by needing different numbers of secondary or shadow rays per pixel in a global illumination application. In this special but important case, the warps' utilization drops every time another thread leaves that kernel, and it should be relatively straightforward to, from time to time, *compact* those threads that are still active at this point in time (hence the name *active thread compaction*), thereby forming fewer, but now fully-utilized, warps.

In this paper, we concentrate on evaluating the effectiveness—and pit-falls—of applying this concept to a global-illumination path tracer in which different paths need to compute different numbers of ray segments until the respective path is terminated. As our experiments will show, applying active path compaction to this problem is rather simple, and, at least for suitable scene, can indeed lead to significant gains in average warp utilization, as well as in savings of more then 2.3× in the number of executions of the respective path extension kernel. However, our experiments also show that certain limitations of today's hardware lead to sources of overhead that significantly affect the final outcome, eventually leading to disappointingly small speed-ups of only 12–16% for even the best-performing of our kernels. In particular, our experiments show that for the most time-consuming part of our kernel—tracing the rays—on today's hardware the cost of a fully-utilized warp is *significantly higher* than for a partly-utilized kernel, in which case producing better-utilized warps has a much smaller benefit than theory suggests. We believe that our results vindicate the validity of the underlying concept, and that, assuming improvements in future hardware architectures, it will eventually be a useful tool in writing efficient GPU programs. We also hope that our analysis will provide insight for other researchers experimenting with this idea.

2 Previous Work

We assume basic familiarity with path tracing (see, e.g., [Pharr and Humphreys 2004]), and with how to implement it. Improving SIMD efficiency in a ray tracing/path tracing context has been addressed by multiple authors on both explicit (CPU) and implicit (GPU) SIMD architectures.

On CPUs, several authors have investigated the use of active thread compaction in ray traversal, by compacting active rays at the beginning of each traversal step [Wald 2007; Gribble and Ramani 2008; Tsakok 2009], or whenever SIMD utilization falls below some threshold [Boulos et al. 2008]. Some of these authors have already pointed out that the same concept could also be applied to higher-level kernels such as shading, but have, so far, investigated them only in the context of ray traversal.

On GPUs, stream compaction is a major component in many GPGPU applications, but is usually used as a higher-level parallel programming primitive rather than merely a method to raise SIMD utilization. Several fast compaction kernels exist; for our application, we use the one that is part of the *CUDA Performance Primitives (CUDPP)* [Harris et al. ; Harris et al. 2007].

Several authors have addressed SIMD utilization in the implementation (e.g., Aila et al.'s speculative traversal steps [2009], or Kalojanov et al.'s special case handling of large cells in grid traversal [Kalojanov et al. 2011]), but usually with special-case solutions. Far closer to our approach, Hoberock at al. [2009] investigated the impact of sorting CUDA shader calls by shader type, and have shown that doing so can boost shader evaluation performance in a path tracer by up to 35%. They did, however, only consider shader execution time, and did not include traversal time. In the context of path tracing, Novak et al. [2010] proposed to re-generate new paths in lanes that got deactivated through Russian-Roulette termination. A similar approach using compaction of still-active paths was also concurrently proposed by van Antwerpen [2011].

*e-mail: Ingo.Wald@intel.com

3 Active Path Compaction

A path tracer works by tracing paths into a scene, randomly bouncing them on the surfaces they hit, and computing how much light gets transported along each path. Each surface interaction is a *node* in its respective path. A path is created by iteratively *extending* it segment by segment, where each additional ray segment involves tracing a ray to find the next path node, evaluating incident lighting at this node, and sampling a new outgoing direction at this path node. During each extension, a path can get terminated for a variety of reasons: it may not hit any geometry, it may reach a hard-coded maximum path depth, it may get absorbed, or it may get (stochastically) terminated because of having too low a pixel contribution.

If we extend many paths in parallel the number of paths active after each bounce will typically drop, with the rate of droppage depending on whether the scene (i.e., open or closed?), the surface properties, and various algorithmic parameters (Russian Roulette?). In that process, there are multiple opportunities for active path compaction: We could perform compaction at the beginning of each bounce (to eliminate paths that did not hit a surface), or at the end (to eliminate those rays that got absorbed or terminated at this node); we could possibly also perform compaction before tracing the shadow rays (not every node will find a valid light sample), or after this (some shadow rays will be occluded); etc.

For the sake of simplicity we will not consider compaction of shadow rays, and only compact once at each bounce. Since ray traversal is significantly more expensive than all of light sampling, BRDF evaluation, BRDF sampling, etc, taken together, we perform compaction right after a given path node is fully evaluated, and this respective path node's outgoing ray (and activity status) is known.

3.1 Path Data

Compacting the set of active paths requires moving a path's entire state from one CUDA thread to another, which in turns requires temporarily storing, and later, resuming this path. To keep path state small we do not store all of a path's previous path segments, and instead only track each path's most recent end-point node. This path node contains information about the current surface sample (position, normals, derivatives, BRDF type, etc), the current path's direction (incoming direction during light sampling and BRDF evaluation, and outgoing direction after BRDF sampling). Each path keeps track of its accumulated weight (pixel contribution) and depth, the state of its (quasi-)random number generator (to ensure consistent sampling even if the path moves between threads), pixel ID (to know where to write the result to), material index, etc.

To keep the memory footprint (and thus, the bandwidth required for storing/loading paths) low, we actually store only the data that has to be maintained among bounces: For example, the state of the random number generator, accumulated path weight and radiance, etc do have to be maintained, but all local surface data, material data, reciprocals of ray direction, etc, do not, and are thus used as temporary variables in the `Path::extend()` kernel, not part of the path node. We also compress memory where possible, e.g., by storing pixel coordinates as 16-bit shorts, bools as bits, etc.

3.2 Path Tracer Infrastructure

For random numbers we use *padded replication sampling* [Keller 2003], which works by *scrambling* a low-discrepancy sequence (we use Halton) using a scramble value obtained from a secondary random number generator—in our case, a linear congruence generator seeded based on pixel coordinates, and advanced for every bounce. This method ensures good samples across multiple paths per pixel, yet requires only a single state variable (for the LCG).

Our path tracer supports a variety of BRDFs including Lambertian diffuse, Blinn, Glass, Chrome, and Car Paint, and contains specialized evaluation and sampling code for each of those. Since CUDA does not yet support virtual functions sampling and evaluation of BRDFs is done through a *Uber*kernel that switches based on BRDF type. If a warp hits surfaces with different BRDFs this obviously leads to significant SIMD divergence. This would be an ideal application for Hoberock et al.'s shader sorting [2009]; however, BRDF sampling and evaluation are so cheap compared to tracing the rays that we currently simply ignore this cost (most of our experiments use all-diffuse scenes, anyway).

At each bounce, we first evaluate incident illumination by generating a light sample, testing it for visibility (if found), and accumulating its pixel contribution (if unoccluded). We then check if the path has reached its maximum depth, and terminate it if this is the case. If not, we sample the BRDF to get a new outgoing path segment, and compute this new segment's accumulated pixel contribution. If this pixel contribution falls below a threshold (we use 5%) we perform *Russian-Roulette* termination with a termination probability proportional to "weight/5%".

Illumination in our path tracer comes from an HDRI environment light source, which is sampled by (solid-angle weighted) pixel contribution. Many better HDRI light sampling techniques exist, and could be integrated without influencing the rest of our system or measurements. For the frame buffer, we use a floating point *accumulation buffer*, which allows for tracing multiple paths per pixel as well as progressively accumulating frames when the camera remains where it was in the previous frame. The ray tracing back-end uses a binary BVH with an Aila et al.-like (but not identical) traversal kernel and pre-gathered triangle data, but without spatial splits. Contrary to Aila et al [2009] for our ray distributions while-while variant has shown to be slightly faster than the speculative version.

Given these building blocks, the entire path tracer is as simple as initializing a path node (→*init()* and iterating over *Path::extend()* until the path terminates; upon termination, the path's accumulated pixel contribution is then written to the frame buffer.

3.3 Naïve Kernel

The *naïve* kernel does exactly that: we create exactly one CUDA thread per pixel, and each such thread traces exactly one path, tracing it all the way until it terminates (see Appendix B.1). The advantage of this kernel is that it is trivially simple to code, and that it does not require any compaction or multiple kernel calls at all. In particular, it does not require any suspension/resumption of paths at all, meaning rays never have to be written out to, or read from, memory, saving both memory footprint and bandwidth.

On the downside, this kernel is exactly the showcase for the original problem, where each additional bounce kills some more of each warp's threads: If we assume the probability of a path to be terminating in any given bounce to be roughly 50% then SIMD utilization drops by half after every bounce.

3.4 Whole-Frame Compaction

In the *whole-frame compaction* kernel, we have the CUDA kernel perform only exactly one bounce (i.e., one path *extension*) for each path: the kernel first reads, for each pixel, the respective path node from a separate memory array (for the 0'th bounce it creates a new primary path rather than reading one from memory); it then—if the

path is still active—computes one bounce, accumulates the incremental radiance in the frame buffer, and writes the updated path node back to this array. Whether the path is still active is stored in a separate array. To render a whole frame the host then first allocates the two arrays that store one path node and active flag per pixel, and calls this kernel N times.

In-between two calls to the path extension kernel we can now use the active-flag array to perform a compaction step that extracts a list of those rays that are still active, as well as the total number of active rays(see Appendix B.2). The compaction itself is done using the stream compaction kernel in the CUDPP (CUDA Performance Primitives) Library [Harris et al. 2007; Harris et al.]. In the bounce kernel the ith thread then simply picks the ith *active* ray (or immediately terminates if $i \geq numActive$), ensuring that except for the single warp that straddles the end of the active index array, all warps will either have all active, or all inactive, threads.

On the upside, this should virtually eliminate partially inactive warps—and thus, have the same work performed by significantly fewer warps. On the downside, this kernel requires to manually split the path tracer into multiple kernels, and to move part of the control flow from the device to the host. On the performance side, we can also expect some overhead: Rather than keeping a path's state in registers over all bounces, each path not gets stored to, and read from, memory after each bounce. In particular, since compaction is done *globally* across *all* rays, there is no locality at all, since a path node written by one multiprocessor in one step will likely be handled by a completely different multiprocessor in the next step. As we cannot start compacting before even the last device thread is done bouncing we also have to perform device-wide barriers after each kernel invocation; and the CUDPP compact kernel will do the same (including a device-readback for the number of still-active path) after each compaction.

3.5 Tiled Compaction

To avoid these issues we also added a variant where we perform active thread compaction only among the threads a thread block: without having to synchronize threads from different thread blocks we can then once again do all bounces in a single kernel, without having to change the host code, and without any global synchronization (see Appendix B.3). This is particularly important in that a compiler could do *this* intra-thread block code transformation fully automatically, whereas the same is not so easy for transformations that go across different thread blocks and kernel invocations.

While we still need a global array to temporarily store path states we can keep both active bit and active index arrays in shared memory, and perform thread compaction locally in shared memory; without any global synchronization at all. Each path node is processed by only one thread block, and only by the single multiprocessor executing this thread block. In particular, because different thread blocks operate completely independent of each other we never have to perform a single global synchronization.

Multiple Paths Per Thread. One issue with this approach is that the *actual* number of CUDA threads per thread block would be far too low to find enough active paths: Because the bounce kernel is complex, CUDA requires 64 registers for this kernel, and best performance is achieved with only 64 threads (i.e., 2 warps) per thread block—which is clearly too few for our purposes. We solve this by having each thread operate on several paths at the same time: each thread block is created with dimensions 64×1, but actually operates on a *tile* of $64 \times M$ pixels, where M is a user-controllable parameter that specifies how many paths per thread we have for compaction: if too few, we will not find enough active paths to fill our two warps; if too many, we might not have enough screen tiles (i.e., thread blocks) to provide work for all multiprocessors. For the intra-tile compaction we our own code that computes a prefix sum over the active-flag array, and then writes a new list of active ray indices, all in shared memory.

(a) fairy (174K triangles) (b) dragon (922K triangles)

(c) troll (622K triangles) (d) dream home (2.63M triangles)

(e) moto (519K triangles) (f) refinery (297K triangles)

Figure 1: Test scenes used for our experiments (all materials set to gray). All models are path traced with lighting from an HDRI environment map, depth of field, up to 8 diffuse bounces, and Russian-Roulette termination for paths with weight less than 5%.

As a side note, this way of having a smaller number of actual hardware threads processing a larger number of "logical threads" is very similar to *persistent threads* [Aila and Laine 2009] (except that we do this only inside one tile, not across the entire frame), as well as to how one would implement the same strategy on CPU.

Tiled Compaction in Shared Memory. So far our tiled compaction has each tile processed by exactly one multiprocessor, but it still spills path nodes to global memory after each bounce. The obvious next step is to move those rays into the respective multiprocessor's shared memory, too (see Appendix B.3), at least in theory elimination any global memory I/O for path state (in practice, we have to take special care to prevent the compiler from spilling each path right after it got read from shared memory!).

Though we have tried to keep the state stored per path node to a minimum, we still need 88 bytes per path node (plus 16 bytes for accumulated radiance). To fit as many rays into shared memory as possible (and thus, allow a larger M), we configure our Fermi to prefer shared memory over cache, but even then can only fit a maximum tiles size of 64×8 when using shared memory.

4 Evaluation and Analysis

Given these kernels we can now evaluate their respective performance. If not stated otherwise, all experiments use a GTX480 card with 1.5GB memory, running under Linux (CUDA Toolkit 3.2, CUDPP version 1.1.1). Extensive experiments with different block

sizes and different compiler settings (in particular, max number of registers per thread) have shown that all kernels performed best for 64 threads per thread block, and 64 registers per thread.

Screen resolution is set to 1280×768 pixels. Our test scenes are depicted in Figure 1; since our path tracer does not currently support (transparency-)textures we disabled all textures and use lambertian diffuse (Kd=.7) for all originally textured surfaces (which, except some glass the DreamHome, applies to almost every surface). Our scenes are mostly "open", with many paths getting quickly lost into the environment. We also ran our path tracer on a highly realistic 4.3 million triangle car model with various different BRDFs, but cannot include pictures into this paper.

Table 1 gives the absolute performance as well as relative performance (with respect to the reference naïve kernel) for the naive, the whole-frame compaction, and the tiled shared-memory kernels (the tiled global-memory kernel performs almost identical to the compaction kernel). We include data for both two and eight bounces (2 *bounces* correspond to a path of *three* segments, plus shadow rays); other bounce counts behave similarly. From Table 1, two facts immediately catch the eye: First, that the achieved speedups—even in the best of cases—are surprisingly low across the board; second, that the kernel we had believed to hold the most promise (the shared-memory version) is actually the slowest—in fact, it is consistently $3\times$ slower than the reference *naïve* kernel.

scene	naïve	whole-frame comp.		tiled (shared mem)	
Path Tracing, max. path length=8					
fairy	3.13	3.54	+13%	1.08	-66%
moto	2.03	2.29	+13%	0.74	-64%
troll	3.19	3.63	+14%	1.10	-66%
dragon	3.61	4.06	+12%	1.21	-66%
dreamhome	3.42	3.88	+13%	1.23	-64%
refinery	4.17	4.84	+16%	1.42	-66%

Table 1: Comparison of absolute performance (in frames per second) and relative performance (compared to the monolithic kernel) for our two kernels. Results are disappointing: Not only are even the best achieved speedups rather low, the shared-memory version is actually more than $3\times$ *slower* than the reference naïve kernel.

4.1 Tiled Shared-Memory Kernel

In particular for the tiled shared-memory kernel we initially believed that *such* bad results could only be explained by an implementation artifact; this however turned out not to be the case: as NVidia's *CUDA Occupancy Calculator* [NVidia a] reveals, the occupancy of the G80 and GF100 (Fermi) architectures not only depends on how many registers a kernel needs (more registers per thread means fewer threads per multiprocessor for latency hiding), but *also* on how much shared memory that kernel uses. As can be seen in Figure 2, for a complex kernel that uses 64 registers (such as ours) the occupancy calculator predicts a maximum 33% device occupancy even if the kernel uses no shared memory at all; once the kernel uses shared memory, too, device occupancy drops even more, to as few as 8% for the 24KB of shared memory required with $M = 4$ (and 4% for $M = 8$). Clearly, our compaction scheme cannot hope to make up for a $4\times$ drop in device occupancy—in fact, it is even a good sign that performance dropped by "only" $3\times$.

Note however that this effect is not specific to our particular kernel, as exactly the same would happen for any other complex (64-register-)kernel that tried to use shared memory—essentially, it means that shared memory should be off limits for any but trivially simple kernels. This raises the hope that this performance bottleneck will eventually get fixed in future hardware revisions, in which case this kernel would again get interesting.

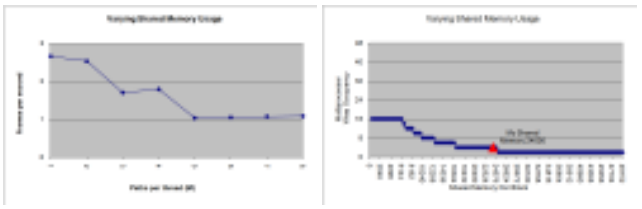

Figure 2: Left: Measured performance of our tiled shared-memory kernel as a function of active paths per thread M. Right: Warp occupancy (as predicted by NVidia's CUDA Occupancy Calculator) of a GTX480 device as a function of a kernel's shared memory use, for a complex kernel requiring 64 registers per thread (device occupancy is roughly $2\times$ that since the HW run two blocks in parallel).

4.2 Full-Frame Compaction Performance

While device occupancy can explain the low performance of the *tiled/shared* kernel, occupancy for all other kernels is exactly the same. This leaves one of two explanations for the disappointingly low performance of the other two kernels: either the compaction kernels carry some significant source of overhead that mask any algorithmic savings from thread compaction; or there simply aren't enough algorithmic savings to start with.

4.2.1 Algorithmic Efficiency

Before any elaborate performance analysis, we decided to first identify the maximum algorithmic savings we could expect: In Table 2 we list, for each bounce, the number of active *paths* as well as the number of active *warps* for both the naive and the whole-frame compaction kernel (a warp is active if either one of its paths is active). Since both kernels use the same random number generators the number of active path is exactly the same for both; the number of active warps obviously is not. As Table 2 shows, by the 8th bounce the number of active paths has dropped by over 16x, but for the naive kernel still over half the warps (19k out of 31k) are still active—which corresponds each active warp having, on average, a mere 3.2 active paths (or a mere 10% warp utilization). Though earlier bounces are less extreme, even averaged across all bounces the average warp utilization is 13.8 out of 32, or 43%. In contrast, the compaction kernel always operates on full warps, and consequently has, across all bounces, $2.3\times$ fewer active warps entering the bounce kernel.

Table 2 clearly demonstrates that *conceptually* the method works as well as expected (savings obviously vary by scene, but our other scenes produce similar results). It does, however, raise the question how this algorithmic benefit of $2.3\times$ can dwindle to a real-world speedup of a mere 16%.

	active	naive		compact	
	paths	#warps	paths/warp	#warps	saved
0	983 **(983)**	30,720 **(31k)**	32 **(32)**	31k **(31k)**	**1**×
1	983 **(1,966)**	30,720 **(61k)**	32 **(32)**	31k **(61k)**	**1**×
2	528 **(2,494)**	30,717 **(92k)**	17 **(27)**	17k **(38k)**	**1.2**×
3	352 **(2,846)**	30,698 **(123k)**	11 **(23)**	11k **(89k)**	**1.4**×
4	225 **(3,071)**	30,144 **(153k)**	7.5 **(20)**	7k **(96k)**	**1.6**×
5	155 **(3,226)**	28,655 **(182k)**	5.4 **(17.8)**	4.8k **(101k)**	**1.8**×
6	108 **(3,334)**	26,383 **(208k)**	4.1 **(16)**	3.4k **(104k)**	**2**×
7	80 **(3,414)**	23,450 **(231k)**	3.4 **(14.7)**	2.5k **(107k)**	**2.2**×
8	60 **(3,474)**	16,651 **(250k)**	3.2 **(13.8)**	1.9k **(108k)**	**2.3**×

Table 2: Number of active paths (in 1,000's) and active warps for the naive and full-frame compaction kernels, respectively, across all 8 bounces, for the fairy scene (bold numbers in brackets are cumulative). The compaction kernel needs a total of $2.3\times$ fewer kernel executions (in #times a warp executes the bounce kernel).

	0	1	2	3	4	5	6	7	8
compaction disabled									
paths/warp	32	32	17	11	7.5	5.4	4.1	3.4	3.2
time/warp	1.6	2.3	1.9	1.4	1.1	1.0	0.9	0.8	0.9
compaction enabled									
paths/warp	32	32	32	32	32	32	32	32	32
time/warp	1.6	2.3	3.1	3.2	3.4	3.6	3.8	4	4.3

Table 3: Execution time (in us) per active warp for the path extension kernel and average utilization of active warps, respectively, once with compaction, once without. Contrary to expectations the time per warp execution—on a GTX480, and for our ray tracing heavy kernel—is *not* independent of warp utilization; in fact, fully utilized warps are about 5× slower than low-utilized ones.

4.2.2 Stream Compaction Overhead

One obvious suspect to explain this discrepancy in real-world vs algorithmic savings is the cost for performing the stream compaction. After measuring the time spent in the `bounce` and `compact` kernels, however, it turned out that even less than 1% of total runtime is spent in compaction. This is far less than expected, and negligible.

4.2.3 Path Store/Load Overhead

The next-biggest suspect is the cost for suspending and resuming paths: in theory, the naive kernel keeps all path data in registers all the time, but the compaction kernel must, in each bounce, first load a path from device memory, and write it back after the bounce. In practice, a look at the PTX code shows that even the naïve kernel seems to be spilling path state to device memory; but there are still a few additional memory accesses to the index and active-flag arrays that may produce some overhead.

To measure this overhead we took the compaction kernel, temporarily disabled the compaction step, measured—for the fairy model—the time per kernel execution, and compared this to the render time of the naive kernel. Independent of the number of bounces, this overhead was fairly constant at 7–8%. This is overhead is already significant, but still far from explaining where how a 2.3× algorithmic savings could almost completely vanish.

4.2.4 Impact of Warp Utilization

With both major sources of overhead ruled out, the final—and biggest—piece of the puzzle only fell into place after taking the full-frame compaction kernel, and measuring—once with, and once without active thread compaction between bounces—the time *per active warp* spent in each bounce (Table 3).

In theory, on a SIMD device this cost should be roughly constant: the multiprocessor executes this kernel even if only a single thread is active, and whether it's 4 or 32 threads at least in first approximation should not matter—after all, that was the rationale behind the entire idea of compacting the same number of threads into fewer warps. In Table 3 however, we see that this is not the case: As expected, thanks to a significant loss in ray coherence we see the time per warp to go up significantly from bounce 0 to bounce 1—and since all primary rays in this scene do find a hit point these numbers are the same for both variants. For the variant where compaction is enabled the time per warp continues to rise for future bounces: to some degree this is because the kernel call overhead can be amortized over fewer active warps (for higher resolutions the effect is less pronounced), but primarily it is due to increasing incoherence of the rays—after all, even in bounce #1 rays have coherent origins, in bounce #2 almost all rays start at the (cheap) walls, etc.

For the variant with compaction disabled, in contrast, we see that the time per warp actually *falls* (and significantly!) the fewer threads are active per warp. In fact, by bounce #8 the time per warp for the low-utilized kernel is almost 5× lower than for the compacted version where all threads are active.

Some of this slowdown could be expected: incoherent rays have a significantly higher potential for SIMD divergence than coherent ones, first because two random rays may have very different runtimes (leading to low SIMD utilization once the fast ones have terminated), and second because two different rays have a high probability of diverging into separate leaf- and inner-node code paths in each traversal step. Consequently, Aila et al. [2010] report incoherent rays to be up to twice as expensive as primary ones. However, while SIMD utilization can lead us to *some* increase in cost per warp for incoherent rays—32 threads have more potential for SIMD divergence in the ray traversal and bounce kernels than 3 threads do—this 5× cost difference between warps that are fully utilized and those that are only to 10% filled can hardly be explained by mere SIMD divergence alone. Instead, the most likely explanation is that for as incoherent memory accesses the underlying hardware is far less capable of hiding memory latencies than expected, in which case a full warp with 32 memory accesses per iteration would suffer significantly more than a low-utilized warp with only 3. In other words: this effect, too, is likely, at least to a significant degree, a side effect of the actual hardware used—if the hardware were better able to hide the ray traverser's memory latencies, our actual speedups should be much higher.

4.2.5 Impact of Shading vs Traversal Costs

If the previous section's explanation was true, a kernel that spent a significantly higher portion of its time in shading computations should produce bigger benefits. Indeed, once we add (dummy) arithmetic computations to the BRDF evaluation we do see significantly higher savings. For instance, if we add enough computations to roughly double the render time of the reference naive kernel, the full-frame compaction kernel's speedup increases to ∼80%.

This, of course, could also be explained by mere SIMD divergence in the traversal kernel (the traversal kernel *does* have significant SIMD divergence, the dummy shader does not). If, however, we change our shader to importance-sample a 2000×1000 pixel environment map light source—which, for inverting the incoming illumination's cumulative density function, requires a 21-step binary search with random memory accesses but with absolutely no SIMD divergence at all—we again observe that absolute shading (and render) time increases significantly, yet the relative speedup through compaction remains unchanged.

4.2.6 Impact of Traversal Kernel

We also experimented with a different traversal kernel, and replaced our `while-while` kernel with a `speculative` variant as proposed by Aila et al. [2009]. This traverser gets slightly better speedups through path compaction, but since the speculative traversal does perform more memory accesses its bottom-line performance is actually slightly worse. We therefore did not include these results.

That this traverser does get a—small, but measurably—better speedup than the non-speculative traverser is a indicator that memory effects are not the only explanation, and that SIMD divergence *inside* the traverser does indeed play *some* role, too. Disentangling the relative importance of SIMD divergence and memory effects would be a highly interesting avenue of future work.

4.2.7 Influence of Geometry and BRDF Types

How well active thread compaction works also depends on how likely it is that some, but not all, of a warp's threads terminate at any point in time. For example, closed scenes will lose far less paths to the environment, and most paths would go all bounces. Also, for less-scattering BRDFs like specular reflection/refraction paths would diverge significantly less, and often all threads in a warp would either all terminate or all survive. On the other hand, scenes with lots of transparency textures (e.g., foliage), materials with different scattering components (e.g., glass), and materials with high variation in albedo (i.e., not all diffuse gray), or a more aggressive use of Russian-Roulette termination [Novak et al. 2010] all increase the likelihood of individual paths dying at different times.

Consequently the algorithmic benefits of our approach depend very much on the actual scene, and even on the viewpoint in this scene. For example, when running our ray tracer on a realistic car model we see hardly any speedups for outside views (where most paths are immediately reflected off into the environment), while seeing savings similar to the ones above for interior views.

5 Summary and Conclusion

We have evaluated the concept of *active thread compaction* in the context of a path tracer. Active thread compaction improves the underlying hardware's SIMD utilization by compacting partially-active warps' threads into fewer but fully utilized warps, thus reducing the number of times the respective kernel is executed. Statistical side, our experiments show that active path compaction can result in algorithmic savings of, for path tracing, up to $3\times$ fewer executions of the core path extension kernel (which includes tracing a ray, plus possibly tracing a shadow ray, and evaluating and sampling the BRDF).

Unfortunately, thanks to some other-than-expected behavior of the underlying hardware those algorithmic gains eventually dwindled into a mere 12–16% speedup even for our best kernel, and even into a $3\times$ *slowdown* for the shared memory-based kernel. We have investigated this somewhat surprising outcome, and identified that it is not because of some intrinsic overheads of our method (e.g., having to suspend and resume logical threads), but rather because of certain hardware limitation when running non-trivial kernels: in particular, for non-trivial kernels needing all 64 registers the device no longer has effective latency hiding for incoherent memory accesses (making memory access, rather than SIMD efficiency, the true limiting factor for such kernels), and the device's shared memory is virtually off limits if a $4\times$ drop in occupancy is to be avoided.

We particularly hope that this analysis of *why* these algorithmic benefits disappeared will prove helpful to other researchers investigating similar ideas. We also hope that those results will help hardware vendors identify where the true bottlenecks for non-trivial kernels lie: for example, in our application it is the memory system and lack of (effective) local "spilling" space that is limiting performance; actual flops and SIMD efficiency apparently are not an issue at all. This is particularly interesting in that both these issues are not specific to our particular kernel, and would apply similarly to any other "complex" kernel (our kernel is not, in fact, all *that* complex) that uses 64 registers while doing incoherent memory accesses and/or using shared memory.

Future Work

In terms of future work, it would be very interesting to do a far deeper investigation into the relative influence of insufficient memory/latency hiding on one side, and SIMD divergence on the other, on the performance observed for our kernels—the better these issues (and their interplay) are understood, the easier it is to find remedies. It would also be interesting to re-do our experiments on other hardware with more readily available "shared" memory/caches, and possibly on future hardware with hopefully improved memory latency hiding for fewer but high-register-count threads. Encouraged by a potential $3\times$ savings for our application, it also seems worthwhile to evaluate this concept for other applications like bidirectional path tracing, photon mapping, and even non-graphics applications (also see [van Antwerpen 2011]). To simplify such experiments, it also looks appealing to experiment with a `_compactThreads()`-like language primitive in a CUDA- or OpenCL-like vectorizing compiler (e.g., [Karrenberg et al. 2010]).

Acknowledgements

Our traversal code was adapted from earlier code written by Sven Woop, and also influenced by Timo Aila's downloadable ray tracing code; the path tracer's light source and BRDF sampling/evaluation code was adapted from earlier code written by Manfred Ernst. Finally, the author would like to explicitly thank the anonymous HPG reviewers for their very detailed and helpful comments, as well as all those that have provided feedback for earlier drafts of this paper.

References

AILA, T., AND KARRAS, T. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proc. High-Performance Graphics 2010*.

AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics 2009*.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive Ray Packet Reordering. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 131–138.

GRIBBLE, C. P., AND RAMANI, K. 2008. Coherent Ray Tracing via Stream Filtering. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 59–66.

HARRIS, M., OWENS, J. D., SENGUPTA, S., TZENG, S., ZHANG, Y., AND DAVIDSON, A. CUDA Data Parallel Primitives Library (v1.1.1). available from http://cudpp.googlecode.com.

HARRIS, M., SENGUPTA, S., AND OWENS, J. D. 2007. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*. Aug.

HOBEROCK, J., LU, V., JIA, Y., AND HART, J. 2009. Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 173–180.

KALOJANOV, J., BILLETER, M., AND SLUSALLEK, P. 2011. Two-Level Grids for Ray Tracing on GPUs. *Computer Graphics Forum (Proceedings of Eurographics '11)*.

KARRENBERG, R., RUBINSTEIN, D., SLUSALLEK, P., AND HACK, S. 2010. AnySL: Efficient and Portable Shading for Ray Tracing. In *High Performance Graphics*.

KELLER, A. 2003. Monte Carlo & Beyond - Course Material. Tech. Rep. 320/02, University of Kaiserslautern. Published in Eurographics 2003 Tutorial Notes.

NOVAK, J., HAVRAN, V., AND DACHSBACHER, C. 2010. Path Regeneration for Interactive Path Tracing. In *Proc. EUROGRAPHICS Short Papers*, 61–64.

NVIDIA. CUDA Occupancy Calculator. Available from http://developer.nvidia.com.

NVIDIA. CUDA Programming Guide (V3.0). Available from http://developer.nvidia.com.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufman.

TSAKOK, J. 2009. Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *Proceedings of High Performance Graphics 2009*, 151–158.

VAN ANTWERPEN, D. 2011. Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *Proceedings of High Performance Graphics 2011*. (to appear).

WALD, I. 2007. SIMD Stream Tracing — SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Reordering. Tech. rep., SCI Institute, University of Utah.

A Path Tracer Infrastructure

A.1 Path::illuminate()

```
void illuminate(Path path, Scene scene) {
  Ray lRay;
  Sample<vec3f> Li; /* incident radiance */
  if (scene.sample_light(lRay,Li,path)))
    if (!scene.occluded(shadow)) {
      vec3f brdf=evalBRDF(path,lRay);
      path.rad += path.wght*brdf / Li.pdf; }
}
```

A.2 Path::extend()

```
bool Path::extend(path, scene) {
  if (path.trace() == false) {
    path.rad+=path.wght*envShade(*this);
    return; }
  illuminate(path,scene);
  return path.bounce(scene);
}
```

B Path Tracing Kernels

B.1 Naïve

Device-side kernel code:

```
_global_ _naive(Scene scene,FrameBuffer fb) {
  Path path;
  path.init(scene.camera,pixeLID,...);
  while (path.extend())
    /* iterate */ ;
  fb.accumulate(path.pixelID,path.L);
}
```

Host-side kernel invocation:

```
void naive(Scene scene, FrameBuffer fb) {
  dim3 block(8,8,1);
  dim3 grid(fb.size.x/8,fb.size.y/8,1);
  _naive<<<grid,block>>>(scene,fb);
}
```

B.2 Path-Front Compaction

Device-side kernel code:

```
_global_ _oneBounce(...,Path pathMem[],int bounce,
                    int isActive[], int numActive) {
  int threadID=pixelID.x+pixelID.y*fb.size.x;
  if (threadID >= numActive) return;
  Path path; int path;
  if (bounce==0)  {
    pathID = threadID;
    path.init(...); }
  else {
    pathID = activeID[threadID];
    path = pathMem[pathID]; }
  bool alive = isActive[pathID] = path.extend();
  if (alive) path[pathID] = path;
  else
     fb.accumulate(path.pixel,path.L);
}
```

Host-side kernel invocation:

```
void wavefront(Scene scene, FrameBuffer fb) {
  dim3 block(8,8,1);
  dim3 grid(fb.size.x/8,fb.size.y/8,1);
  int numPixels  = fb.size.x*fb.size.y;
  int *activeBit = new int[numPixels];
  int *activeID  = new int[numPixels]
  Path *paths    = new Path[numPixels];
  int numActive  = numPixels;
  while (numActive != 0) {
    _wavefront<<<grid,block>>>(...,numActive);
    numActive = doCompaction(activeID,activeBit); }
  ... /* free memory */
}
```

B.3 Tiled Wave-Front (global memory)

Shared-memory version identical except for `pathMem` being a local array with `_shared_` storage specifiers.

Device-side kernel code:

```
#define TILE_X 64
#define TILE_Y 8 /* #initl paths/thread */
_global_ _tiled(...,Path pathMem[],int bounce,
            int isActive[], int numActive) {
  int tileID=blockIdx.x+blockIdx.y*blockDim.x;
  Path *tileMem=&pathMem[tileID*TILE_X*TILE_Y];
  _shared_ uint activeID[TILE_X*TILE_Y];
  _shared_ bool activeBit[TILE_X*TILE_Y];
  _shared_ helperStuffForPrefixSum;
  // first pass: initialize all rays
  for (int y=0..TILE_Y) {
    int pathID = y*TILE_Y+threadIdx.x;
    vec2i real_pixelID=...;
    tileMem[pathID].init(...);
    activeID[pathID]=pathID;
    activeBit[pathID]=isAvalidPixelID(...); }
  // all rays initialized: bounce and compact
  while (any active) {
    for (int y=0..TILE_Y) {
      int pathID=pathID[y*TILE_Y;
      it (!activeBit[pathID]) continue;
      path=tileMem[pathID];
      bool alive=path.doOneBounce(...);
      if (alive) {
        tileMem[pathID]=path; }
      else {
        fb.accumulate(...);
        activeBit[pathID]=false; }}
    _syncthreads();
    do_local_compaction(activeBit,activeID);
    _syncthreads();  }
}
```

Host-side kernel invocation:

```
void wavefront(Scene scene, FrameBuffer fb) {
  dim3 block(TILE_X,1,1);
  dim3 grid(fb.size.x/TILE_X,fb.size.y/TILE_Y,1);
  Path *paths    = new Path[fb.size.x*fb.size.y];
  _tiled<<<grid,block>>>(...,numActive);
  ... /* free memory */
}
```