Efficient Multi-kernel Ray Tracing for GPUs

Thomas Schiffer[1] and Dieter W. Fellner[1,2,3]

[1]*Institut für ComputerGraphik & Wissensvisualisierung, Technische Universität Graz, Graz, Austria*

[2]*Graphisch-Interaktive Systeme, Technische Universität Darmstadt, Darmstadt, Germany*

[3]*Fraunhofer Institut für Graphische Datenverarbeitung, Fraunhofer Gesellschaft, Darmstadt, Germany*

Keywords: Ray Tracing, SIMT, Parallelism, GPU.

Abstract: Images with high visual quality are often generated by a ray tracing algorithm. Despite its conceptual simplicity, designing an efficient mapping of ray tracing computations to massively parallel hardware architectures is a challenging task.In this paper we investigate the performance of state-of-the-art ray traversal algorithms for bounding volume hierarchies on GPUs and discuss their potentials and limitations. Based on this analysis, a novel ray traversal scheme called batch tracing is proposed. It decomposes the task into multiple kernels, each of which is designed for efficient parallel execution. Our algorithm achieves comparable performance to currently prevailing approaches and represents a promising avenue for future research.

1 INTRODUCTION

Ray tracing is a widely used algorithm to compute highly realistic renderings of complex scenes. Due to its huge computational requirements massively parallel hardware architectures like modern graphics processing units (GPUs) have become attractive target platforms for implementations. We investigate high performance ray tracing on NVidia GPUs, but many of our contributions and analysis may also apply to other wide-SIMD hardware systems with similar characteristics. In this paper, we use bounding volume hierarchies (BVHs) as acceleration structure for ray traversal and we solely focus on the task of intersecting a ray with a scene containing geometric primitives only and do not include shading and other operations into our discussion.

In general, we evaluate our algorithms on ray loads generated by a path tracer with a fixed maximum path length of 3. Our test scenes contain only purely diffuse materials, from which rays bounce off to a completely random direction of the hemisphere. So, the generated ray data covers a broad spectrum of coherency ranging from coherent primary rays to highly incoherent ones after two diffuse bounces. For a diligent performance assessment, we use a diverse set of NVidia GeForce GPUs consisting of a low-end Fermi chip (GT 540M), a high-end Fermi chip (GTX 590) and a high-end Kepler-based product (GTX 680).

Our paper has the following structure: First, we provide an analysis of the state-of-the-art ray tracing algorithms and their characteristics that seem to leave significant room for improvement. We subsequently discuss our novel algorithmic approach called batch tracing that addresses the current problems. Finally, we analyze the practical implementation as well as some optimizations and point out possible directions for future research.

2 PREVIOUS WORK

This paper targets massively parallel hardware architectures like NVidia's Compute Unified Device Architecture (CUDA) that was initially presented by Lindholm et al. (Lindholm et al., 2008). CUDA hardware is based on a single-instruction, multiple-thread (SIMT) model, which extends the commonly known single-instruction, multiple-data (SIMD) paradigm. On SIMT hardware, threads are divided into small, equally-sized groups of elements called warps (current NVidia GPUs batch 32 threads in one warp). Contrary to SIMD, execution divergence of threads in the same warp is handled in hardware and thus transparent to the programmer. To avoid starvation of the numerous powerful computing cores of GPUs, L1 and L2 caches were introduced with the advent of the more recent generation of hardware called Fermi (NVIDIA, 2009). The latest architecture of

DOI: 10.5220/0004703502090217
In *Proceedings of the 9th International Conference on Computer Graphics Theory and Applications* (GRAPP-2014), pages 209-217
ISBN: 978-989-758-002-4

GPU hardware termed Kepler (NVidia, 2012) contains several performance improvements concerning atomic operations, kernel execution and scheduling.

In 2009, Aila et al. (Aila and Laine, 2009) presented efficient depth-first BVH traversal methods, which still constitute the state-of-the-art approach for GPU ray tracing. Their implementation uses a single kernel containing traversal and intersection, that is run for each input ray in parallel. More recently they presented some improvements, algorithmic variants to increase the SIMT efficiency and updated results for NVidia's Kepler architecture in (Aila et al., 2012). They also reported that ray tracing potentially generates irregular workloads, especially for incoherent rays. To handle these uneven distributions of work, compaction and reordering have been employed in the context of shading computations (Hoberock et al., 2009), ray-primitive intersection tasks (Pantaleoni et al., 2010) and GPU path tracing (Wald, 2011). Garanzha et al. propose a breadth-first BVH traversal approach in (Garanzha and Loop, 2010), which is implemented using a pipeline of multiple GPU kernels. While coherent rays can be handled very efficiently using frustra-based traversal optimizations, the performance for incoherent ray loads significantly falls below the depth-first ray tracing algorithms as reported in (Garanzha, 2010).

3 THE QUEST FOR EFFICIENCY

While the SIMT model is comfortable for the programmer (e.g. no tedious masking operations have to be implemented), the hardware still has to schedule the current instruction for all threads of the warp. Diverging code paths within a warp have to be serialized and lead to redundant instruction issues for non-participating threads. If code executed on SIMT hardware exhibits much intra-warp divergence, a considerable amount of computational bandwidth will be wasted. In accordance with Section 1 of (Aila and Laine, 2009), we use the term SIMT efficiency, which denotes the percentage of actually useful operations performed by the active threads related to the total amount of issued operations per warp to quantify this wastage in our experiments. As computational bandwidth continues to increase faster than memory bandwidth, SIMT efficiency is one of the key factors for high performance on current and most probably also future GPU hardware platforms.

Beside computational power, memory accesses and their efficiency are a crucial issue too. Threads of a warp should access memory in a coherent fashion, in order to get optimal performance by coalescing their requests. However, traditional ray tracing algorithms (as well as many others) are known to generate incoherent access patterns that potentially waste a substantial amount of memory bandwidth as discussed in (Aila and Karras, 2010). The caches of GPUs can help to improve the situation considerably as discussed by Aila et al. in (Aila and Laine, 2009) and (Aila et al., 2012), where they describe how recent improvements of the GPU cache hierarchy affect the overall ray tracing performance.

In general, GPUs are optimized for workloads that distribute the efforts evenly among the active threads. As reported in (Aila and Laine, 2009), ray tracing algorithms generate potentially unbalanced workloads, especially for incoherent rays. This fact poses substantial challenges to the hardware schedulers and can still be a major source for inefficiency as noted by Tzeng et al. (Tzeng et al., 2010). To support the scheduling hardware and achieve a more favorable distribution of work, compaction and task reordering steps are explicitly included in algorithms. This paradigm has been successfully applied to shading computations of scenes with strongly differing shader complexity (Hoberock et al., 2009), where shaders of the same type are grouped together to allow more coherent warp execution. In the context of GPU path tracing, Wald applied a compaction step to the rays after the construction of each path segment in order to remove inactive rays from the subsequent computations (Wald, 2011).

3.1 Depth-first Traversals

The most commonly used approach for GPU ray tracing is based on a depth-first traversal of a BVH as discussed in (Aila and Laine, 2009). Their approach is based on a monolithic design, which combines BVH traversal and intersection of geometric primitives into a single kernel. This kernel is executed for each ray of an input array in massively parallel fashion. Given two different rays contained in the same warp, potential inefficiencies now stem from the fact that either they require different operations (e.g. one ray needs to execute a traversal step, while the other ray needs to perform primitive intersection) or the sequences of required operations have a different length (e.g. one ray misses the root node of the BVH, while the other ray does not). These two fundamental problems have a negative impact on SIMT efficiency and lead to an uneven distribution of work, especially for incoherent ray loads.

To mitigate the effects of irregular work distribution, Aila et al. investigated the concept of dynamic work fetching. Given a large number of

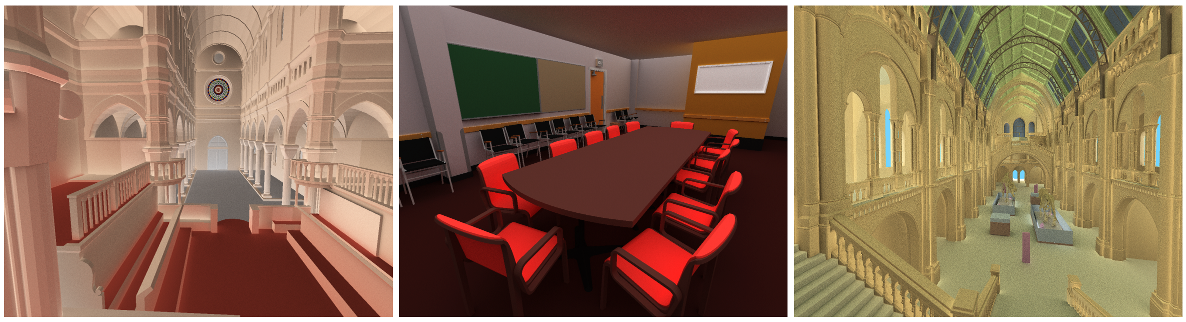
Figure 1: Ray tracing-based renderings of our test scenes approximating diffuse illumination. From left to right: Sibenik (80K triangles), Conference (282K triangles) and Museumhall (1470K triangles).

Table 1: SIMT efficiency percentages for traversal (T) and intersection (I) as well as kernel's warp execution efficiency (K) of the two best performing ray tracing kernels *fermi_speculative_while_while* (from (Aila and Laine, 2009)) and *kepler_dynamic_fetch* (from (Aila et al., 2012)) using different thresholds of active rays for dynamic fetching of work. The numbers are averages of multiple views taken of the Sibenik and the Museum scene for different generations of rays.

Scene	Rays	while-while			dyn-fetch 25%			dyn-fetch 50%			dyn-fetch 75%		
		T	I	K	T	I	K	T	I	K	T	I	K
Sibenik	1. gen	87.0	61.3	80.1	86.4	59.9	79.2	86.4	60.0	79.4	86.3	60.0	79.1
	2. gen	46.7	27.7	40.5	49.2	30.1	43.0	49.7	30.5	43.3	50.4	31.4	43.9
	3. gen	38.4	23.6	33.8	41.0	26.2	36.2	42.1	27.6	37.2	42.5	28.2	37.6
Museum	1. gen	65.4	45.8	59.0	66.2	46.2	59.4	66.8	47.1	60.3	65.9	47.7	59.8
	2. gen	32.6	19.8	28.8	37.9	28.7	34.8	39.7	31.7	36.6	40.3	33.2	37.2
	3. gen	27.8	16.6	24.5	33.6	25.8	31.2	35.3	28.7	32.9	36.3	30.3	33.9

input tasks, the idea is to process them using an (usually significantly smaller) amount of worker threads that fetch new input items until all input elements have been processed. In this paradigm, the fetching of a new task can happen immediately after the current one has been finished or depending on a more general condition (e.g. all threads of a warp have completed their tasks). This dynamic software scheduling was introduced in Section 3.3 of (Aila and Laine, 2009) called persistent threads, where they used dynamic fetching on a per-warp basis to balance the shortcomings of the hardware scheduler on Tesla generation hardware. In (Aila et al., 2012) they suggest fetching new rays on Kepler hardware, if more than 40% of the threads of a warp have finished their work. It is important to note that dynamic fetching of work attempts to increase warp utilization and thus potentially SIMT efficiency at the expense of memory bandwidth, because it tends to generate incoherent access patterns when fetching single or only a few new work items.

To have a baseline for our further experiments, we evaluated the SIMT efficiency of the two best performing ray tracing kernels *fermi_speculative_while_while* (from (Aila and Laine, 2009)) and *kepler_dynamic_fetch* (from (Aila et al., 2012)) using different thresholds for the dynamic fetching paradigm. We directly used large parts of Aila et al.'s kernels with minor adaptations for our evaluation framework. Table 1 shows SIMT efficiency percentages of the traversal and the intersection code parts and the kernel's overall warp execution efficiency as reported by the CUDA profiler obtained by averaging over multiple views of the test scenes. The SIMT efficiency for traversal and intersection have been computed by inserting additional code into the kernel, which counts the number of operations that are executed by the kernel and that are actually issued on the hardware using atomic instructions. However, the percentages in Table 1 vary from what Aila et al. reported in Table 1 of (Aila and Laine, 2009), since we use different view points and a different BVH builder. In (Aila et al., 2012), Aila et al. note that fine-grained dynamic work fetching on Kepler improves the performance by around ten percent, especially for incoherent rays. This fact can be well explained by looking at the corresponding efficiency percentages in Table 1. For coherent primary rays the percentages are already quite high and hardly change, so there is no notable overall performance gain. Incoherent ray loads, however, definitely benefit from this optimization. A newly fetched ray can either require the same sequence of operations as the others and thus increases, or require a different sequence of operations and thus decreases the SIMT efficiency of a warp. Assuming equal probabilities for traversal and intersection, it is therefore probable that low percentages of incoherent rays are increased as indicated by the measurements.

We also experimented with speculative traversal as proposed in Section 4 of (Aila and Laine, 2009) using a postpone buffer of small size of up to three elements. Comparing ordinary and speculative traversals, we observed an increase in traversal efficiency of up to 10% for coherent and up to 50% for incoherent rays, which becomes smaller for larger buffer sizes. Please note that the kernels shown in Table 1 already use a fixed size postpone buffer of one element. So, enlarging the speculative buffer yields just a slight increase of up to 10% in traversal efficiency for incoherent rays, but no overall performance improvement in our experiments. Also, we noted a slight decrease in intersection efficiency for buffer sizes larger than one for ray tracing kernels based on the while-while paradigm, which is in accordance with the results in Table 1 of (Aila and Laine, 2009). These results hint that in a monolithic design, SIMT efficiency of traversal and intersection are correlated and cannot be increased arbitrarily without adversely impacting the other. Pantaleoni et al. (in Section 4.4 of (Pantaleoni et al., 2010)) address this problem by redistributing intersection tasks of the active rays among all rays of the warp. After the intersection they perform a reduction step to obtain the closest intersection for each ray. This structural modifications of the monolithic approach result in a significantly increased SIMT efficiency of intersection (50%-60% instead of about 25%), but is used only for specialized spherical sampling rays in an out-of-core ray tracing system.

3.2 Breadth-first Traversals

A radically different approach to GPU-based ray tracing was presented by Garanzha et al. in (Garanzha and Loop, 2010). They implemented a ray tracing algorithm that traverses their specialized BVH structure in a breadth-first manner including some key modifications. In a first step before the actual traversal, the input rays are sorted and partitioned into coherent groups bounded by frusta. Then, traversal starts at the root node. At the currently processed tree level, all active frustra (active means intersecting a part of the BVH) are intersected with the inner nodes and these results are propagated to the next lower level. Traversal stops at the leaf nodes and yields lists of intersected leaves for each frustum. Subsequently, all rays of each active frustum are tested for intersection with the primitives contained in each leaf to obtain the final results. As described in (Garanzha, 2010), each of these steps is implemented using multiple kernels resulting in a rather long pipeline. Although we did not provide an actual implementation of their algorithm, we still want to note several important characteristics of their approach.

First of all, we argue that large parts of their implementation can be assumed to exhibit high SIMT efficiency. Each kernel is carefully designed to perform a single task (e.g. frustum intersection) and complex operations are broken down into smaller components (e.g. ray sorting exploiting existing coherency), which can again be efficiently implemented. Secondly, this approach also tends to generate workloads that are significantly more coherent and regular than the ones of depth-first traversal. There appear to be no sources for such major inefficiencies in the traversal phase and persistent threads are used to balance the workloads in the intersection stage, since the length of the corresponding list of leaf nodes may vary significantly. We believe that the performance wins over monolithic traversals reported on rather coherent rays are not only due to potentially efficient multi-kernel implementation, but also largely due to their clever frusta-traversal based optimization.

Despite these favorable properties, the approach also possesses a major algorithmic inefficiency. In the traversal stage, a complete traversal of the acceleration structure is performed for each frustum regardless of the number of the actually necessary operations. Thus, a lot of likely redundant work per ray is carried out, since the ray may intersect a primitive that is referenced by the first leaf that is encountered in traversal. This node over-fetching potentially leads to a lot of redundant instructions and memory accesses. For coherent rays, this drawback apparently does not outweigh the benefits from the efficient traversal and intersection stages. For incoherent rays, however, the set of the traversed nodes grows considerably and the node over-fetching dominates the all aforementioned benefits resulting in a significantly reduced overall performance.

4 MULTI-KERNEL BATCH TRAVERSAL

Based on the room for improvement that is left by current state-of-the-art ray tracing algorithms, we propose a novel approach called batch traversal. It is designed to achieve the following objectives:

- Given the low percentages for incoherent rays of depth-first traversals shown in Table 1, our algorithm should notably increase SIMT efficiency.

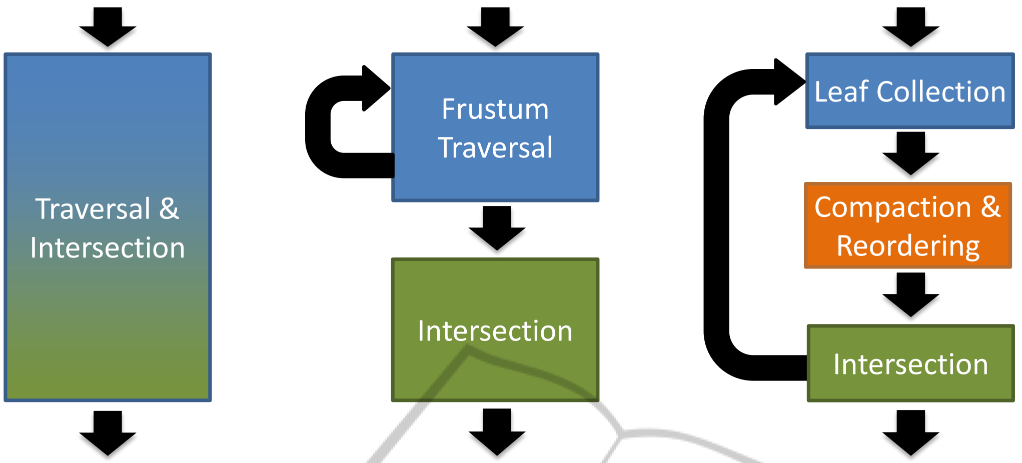

Figure 2: High-level system designs for monolithic depth-first (left), breadth-first (middle) and our batch tracing approach (right). Batch tracing consists of multiple phases, which are repeatedly executed until all rays have terminated. Blocks denote logical parts of the algorithms, black arrows denote the control flow.

- Unlike breadth-first traversal, the algorithm should not exhibit substantial inefficiencies in handling ray loads of varying coherency.

The components of our algorithm and their interplay are shown in Figure 2 (right) together with depth-first (left) and breadth-first (middle) traversals. Like in breadth-first traversal, our approach splits the ray tracing task into several algorithmic stages implemented in multiple kernels to increase SIMT efficiency of the single code parts. During leaf collection stage, the ray traverses the BVH similar to depth-first traversal and collects a number of intersected leaf nodes called intersection candidates. In the subsequent phase, the collected intersection candidates are reorganized to achieve efficient execution of the intersection stage and the active rays are compacted for the next execution of the traversal stage. In the intersection phase the primitives contained in the intersection candidates are tested for intersection with the ray and the results are used to update the rays. These stages are executed in a loop until all input rays have terminated. Contrary to breadth-first traversal, our approach performs partial BVH traversals and intersection testing alternately in order to avoid collecting a large number of redundant intersection candidates.

4.1 Leaf Collection

As mentioned before, the leaf collection stage performs a partial depth-first traversal of the BVH similar to the monolithic ray tracing algorithms and is implemented in a single kernel. If a leaf is encountered during traversal, it is stored into the buffer of the corresponding ray and traversal continues immediately as shown in Algorithm 1. A ray terminates this stage if the whole BVH has been traversed or a certain maximum number of collected leaves has been found. The efficiency of this stage depends on the number of intersection candidates that have to be expected for a given ray. The second crucial question is how the collection of these leaves should be distributed optimally among the various iterations.

Algorithm 1 Leaf Collection Stage

```
while ray not terminated do
    while node is inner node do
        traverse to the next node
    end while
    store leaf into buffer
    if maximum number of leaves collected then
        return
    end if
end while
```

As the overall number of collected leaves for a ray is influenced by many factors (e.g. input ray distribution and scene geometry), we provide probabilistic bounds backed up by experiments. To estimate the size of the intersection candidate set, we assume a uniformly distributed ray load and that our input scene consists of N objects O_i. Each O_i is enclosed by a bounding box B_i and all objects are enclosed by a scene bounding box W. Furthermore, let $B = \cup B_i$ be the union of all bounding boxes and $p(n)$ be the probability for a random line to intersect n objects, given it intersects W. Using Proposition 5 from (Cazals and Sbert, 1997), a tight upper bound for $p(n)$ is then given by Equation 1 (SA denotes surface area).

$$p(n) \leq \frac{SA(B)}{nSA(W)} \tag{1}$$

This probabilistic bound shows that the size of a set of intersection candidates for a random ray can be expected to be reasonably small. In practice, we can hope for an even smaller candidate set, because we are only interested in intersections occurring before the first hit of the ray.

To assess the quality of these theoretical considerations in practice, we have evaluated the size of the intersection candidate set for multiple views of our test scenes. The obtained results prove our probabilistic assumption that for a large number of rays the size of the candidate is small (< 4 for 70% and < 8 for 90% of the rays) regardless of their distribution. Figure 3 shows a histogram of the size of the intersection candidate set for different ray generations for a view of the Sibenik and a view of the Museumhall scenes. Firstly, the histogram varies significantly for small set sizes (e.g. up to 8 for Figure 3) depending on the distribution of the input rays (e.g. viewpoint). However, the distribution of the larger candidate set sizes exhibits no dependence on the input ray distribution approaching zero for all scenes and viewpoints. Additionally, the overall shape of the histogram curves is also similar for all tested ray loads and input scenes. Although specific ray-geometry configurations might cause exceptionally large intersection candidate lists, these cases can still be handled by our implementation in a reasonably efficient manner.

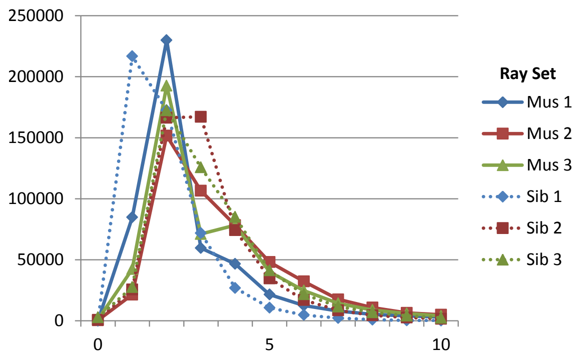

Figure 3: Histogram of size of the intersection candidate set for rays of different generations (1-3) for a view of the Sibenik (Sib) and a view of the Museumhall (Mus) scene. The intersection candidate set of a ray consists of all leaves that are pierced by this ray from its origin to its first hit point. The X axis shows the size of the intersection candidate set, the Y axis displays the corresponding number of rays. The graphs for larger candidate set sizes all are very close to the X axis, thus only set sizes up to 10 are shown.

The second important open issue that remains is, how the collection of the leaves should be distributed among the various passes. It is challenging to provide a short and optimal solution, since choosing the number of intersection candidates per iteration is a trade-off. Collecting a large number of leaves per iteration reduces the total number of required passes and thus potentially increases overall performance. However, a lot of potentially redundant traversal and intersection operations might be introduced, since most of the rays have only got a small candidate set. To counter this effect, a small number of intersection candidates per pass can be chosen to minimize the overhead, potentially increasing the total number of iterations required. We finally came up with the following heuristic, which is backed up by numerous experiments: We start with a small number (e.g. 2 for GT 540M) in the first pass and increase the size of the intersection candidate set gradually over the next few passes. This avoids much redundant work in the first passes and also keeps the number of total passes in an acceptable range.

4.2 Compaction and Reordering

The compaction and reordering stage performs various housekeeping tasks in a single kernel. Firstly, we perform a compaction of the rays that remain active after the execution of the leaf collection stage. We simply map the active rays to consecutive slots using parallel reduction based on atomic instructions for shared and global memory. This removes already terminated rays and helps to maintain a high efficiency during the leaf collection stage. Additionally, this stage is responsible for managing the leaf collection buffer. At the start of our batch tracing algorithm, this buffer is allocated once with a size that is a fixed multiple of the number of input rays (e.g. number of rays times eight elements).

In each leaf collection pass we partition this buffer in equal parts among the remaining active rays. This layout can flexibly accommodate a wide range of thresholds for the number of collected leaves without the need to perform costly memory reallocations. Secondly, a parallel grid-wide reduction is made to compute the total number of intersection tasks and to assign these tasks to threads for the subsequent intersection stage. Again, rays, which have collected leaves during the previous stage, are mapped to consecutive slots for efficiency reasons. For storing the different mappings generated by work compaction, index arrays are allocated at the start of the algorithm. They keep track of the original indices for all active rays and intersection tasks and help to save costly movements of larger data structures (e.g. input ray data or traversal stacks).

4.3 Intersection

Our implementation of the intersection stage maps the intersection tasks of one ray to one thread, which iteratively processes its array of collected leaves. For each leaf, all contained primitives are intersected with the ray and the results are stored for use in the subsequent phases of the algorithm.

5 RESULTS AND DISCUSSION

In this section, we evaluate the previously discussed batch tracing algorithm and discuss some optimizations. To see how the total running time is distributed we profiled our implementation for different scenes and ray loads. Around 70% of the time is spent in the leaf collection, which makes this stage the primary target for optimizations. Intersection computations constitute 25% of total time and compaction and reordering 5%, while negligible overhead is caused by memory allocations and CPU-GPU communication.

5.1 Optimized Leaf Collection

In order to assess the efficiency of the leaf collection stage, we analyzed the resulting distribution of workloads. As the number of operations for different rays may vary considerably, we implemented an optimized version that uses dynamic fetching of work to counter the effects of this imbalance. If the SIMT efficiency of a warp drops below a certain threshold, already terminated rays get replaced with new input rays. We experimented with different threshold percentages and empirically determined an optimal threshold of 25% for the GTX 590. For the GT 540M dynamic fetching is only beneficial if performed on a per-warp basis, i.e. when the last active ray of a warp terminates, a whole group of new 32 work items is fetched. For a detailed comparison with monolithic traversal approaches we implemented an extended kernel based on *fermi_speculative_while_while* that uses dynamic fetching of rays in the sense of (Aila et al., 2012). Table 2 shows the ray tracing performance for the Museum scene of our monolithic dynamic fetch kernel (DF), the batch tracing baseline implementation (Batch) and its improved variant using dynamic fetching (Batch+DF) relative to the performance of the *fermi_speculative_while_while* kernel. Our optimized batch tracing algorithm (Batch+DF) achieves comparable performance on the 540 GT, but is consistently outperformed on the 590 GTX by our monolithic dynamic fetch kernel. The low-end card with few compute cores seems to benefit from our batch tracing approach especially in combination with per-warp dynamic fetching, because irregular workloads can be hardly balanced out by the hardware. On Kepler hardware batch traversal is around 70% slower than our fastest monolithic kernel regardless of the coherency of the input rays.

Table 2: Ray tracing performance for our monolithic dynamic fetch kernel (DF), the batch tracing baseline implementation (Batch) and its improved variant using dynamic fetching again (Bt+DF). Numbers are relative to the performance of our implementation of *fermi_speculative_while_while* and were averaged over multiple views of the Museum scene. The largest speed-ups are shown in bold numbers.

GPU	Ray Type	DF	Batch	Bt+DF
GT 540M	1. gen	**1.13**	0.97	1.04
	2. gen	1.46	1.42	**1.54**
	3. gen	**1.55**	1.42	**1.55**
GTX 590	1. gen	**1.09**	0.78	0.79
	2. gen	**1.42**	1.10	1.17
	3. gen	**1.51**	1.12	1.21

While dynamic fetching increases the performance of the monolithic ray traversal notably, only small speedups (up to 10%) are achieved for the leaf collection stage. Contrary to the obtained results, we expected larger benefits (at least in a range similar to the difference between *fermi_speculative_while_while* and its dynamic work fetching variant) from this optimization: Since the leaf collection kernel contains no intersection code, dynamic fetching should deliver large increases in SIMT efficiency and also improve overall performance. In order to find a plausible cause for this discrepancy, we measured the warp execution efficiency of the leaf collection stage. Table 3 shows the percentages obtained by batch tracing using different variants of dynamic fetching (per-warp and using different thresholds) in comparison to the highest SIMT efficiency values for traversal achieved by state-of-the-art monolithic kernels in the Museum scene. As we expected, the efficiency of the BVH traversal is slightly increased for primary rays (up to 10%) and substantially enhanced for high order rays (up to almost 60%).

Since these improvements do not result in an overall performance win, we decided to profile our implementation comprehensively. The analysis revealed that using dynamic fetching for our leaf collection kernels results in an increased number of instruction issues due to memory replays. This fact strongly hints that the performance of this stage is limited by the caches of the GPU. As the available DRAM bandwidth is not entirely used up, we believe that the unexpected results are caused by the unfavorable memory

Table 3: SIMT efficiency percentages for traversal of monolithic kernels (Mono) and for the leaf collection stage of batch tracing using dynamic fetching per-warp (Warp DF) and with different thresholds (DF 25% and DF 50%) in the Museum scene. For the first column, we use the highest percentages achieved by any monolithic kernels, including all variants of dynamic fetching (see also Table 1).

Ray Type	Mono	Warp DF	DF 25	DF 50
1. gen	66.8	66.5	72.1	69.3
2. gen	40.3	36.7	52.3	56.8
3. gen	36.3	32.6	50.3	54.6

access pattern resulting from traversal of incoherent rays. This leads to an exhausted cache subsystem that cannot service all the memory requests on time resulting in severe latency that nullifies most of the benefits of the increased SIMT efficiency.

To verify the aforementioned assumptions experimentally and to identify primary performance limiters we use selective over-clocking of the processing cores and the memory subsystem (e.g. the performance of memory-bounded kernels will benefit from an increased memory clock rate). In fact, our leaf collection kernels hit the memory wall much earlier than we expected, given the fact that they perform no intersection computations and thus have a smaller memory working set compared to monolithic traversal. While we obtained an optimal threshold of 25% for dynamic fetching on the GTX 590, it only decreases performance on the Kepler hardware. In our opinion, this shows a drawback of separating traversal and intersection: Leaf collection kernels cannot hide the occurring memory latencies as well as a monolithic kernel, which can potentially schedule additional operations while waiting for memory fetches (e.g. execute primitive intersection while waiting for a BVH node memory access). These observations raise the question for alternative memory layouts and acceleration structures, which better suit this kind of traversal algorithm.

5.2 Traversal Performance Limiters

However, the analysis of our monolithic ray tracing kernels revealed that memory performance becomes a major challenge for this kind of algorithms, too. As discussed by Aila et al. (Aila and Laine, 2009), traditional depth-first traversal are commonly believed to be compute-bound. This is confirmed by our results and and holds for Fermi hardware regardless of whether dynamic fetching is used or not. The Kepler-based GTX 680, however, provides a significantly increased amount of computational power, while memory bandwidth has grown only moderately in comparison. Our speculative depth-first traversal kernel is apparently still compute-bound as shown in Figure 4. The *kepler_dynamic_fetch* kernel similarly

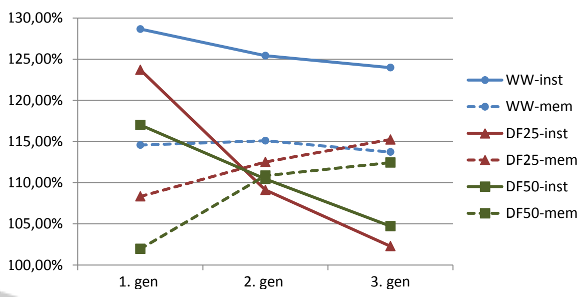

Figure 4: Kernel performance limiter analysis for monolithic ray traversals on Kepler-based 680 GTX. We evaluate our fastest monolithic kernel (WW) using different thresholds for dynamic fetching (DF25 and DF50). Series suffixed with "-instr" denote that processor clock has been increased by 20%, "-mem" means that the memory clock has been over-clocked by 20%. The graphs show obtained performance relative to the default settings of the device.

exhibits compute-bound behavior for coherent ray loads. With diminishing coherence of the input rays, dynamic fetching overcharges the cache subsystem with incoherent memory requests, which in turn becomes the dominating factor. Contrary to the speculations in (Aila et al., 2012), this shows that monolithic state-of-the-art traversal already tends to be limited by cache and memory performance, at least for incoherent rays. Since a significant decrease in the instruction-to-byte ratio in future GPUs cannot be expected, memory bandwidth and cache throughput will become the primary limiter for traditional algorithms.

5.3 Intersection Stage

Table 4 shows the SIMT efficiency of our implementation of the intersection stage in comparison to the highest numbers that have been achieved by monolithic traversals. Our approach improves the efficiency of primitive intersection notably which is now in the range of what has been achieved by intersection task reordering (Pantaleoni et al., 2010). As the leaf collection pass usually submits an amount of redundantly collected leaves to the traversal stage, we tried to avoid all unnecessary primitive tests by storing the entry distance of a ray along with every intersection

Table 4: SIMT efficiency percentages of primitive intersection for monolithic kernels (Mono) and our batch tracing implementation (Batch) for the Museum scene.

Ray Type	Mono	Batch
1. gen	47.7	58.3
2. gen	33.2	47.1
3. gen	30.3	45.4

candidate. The ray always examines this value before any contained primitives are tested. However, this optimization yielded no noticeable performance improvements and might pay off only for more complex geometric primitives. Our basic mapping which lets one ray work on one intersection candidate set generates potentially uneven workloads, since the number of primitives contained in a single leaf may vary as well as the size of the intersection candidate set. Measurements hint that our intersection implementation is memory-bounded and we suppose that it would benefit significantly from a more elaborate reordering of the intersection tasks.

6 CONCLUSIONS AND FUTURE WORK

Given our comprehensive analysis of current ray traversal algorithms, we believe that state-of-the-art approaches leave a lot of room for future algorithmic improvements. A prominent example is the SIMT efficiency of depth-first traversal algorithms, which waste a considerable amount of computational bandwidth when dealing with incoherent rays. However, also the limits of traditional paradigms become obvious, which puts the research focus on alternative traversal methods like the batch tracing algorithm proposed in this paper.

Although the provided implementation cannot quite compete with mature and heavily optimized monolithic traversal, our multi-kernel method possesses appealing characteristics, which makes it an attractive direction for further research. A central point is to find an acceleration structure that permits increased traversal performance to make our approach more beneficial. Furthermore, we would like to optimize the intersection stage of our algorithm and investigate efficient handling of multiple types of geometric primitives.

ACKNOWLEDGEMENTS

We thank Marko Dabrovic for providing the Sibenik cathedral model and the University of Utah for the Fairy Scene. Many thanks also go to Timo Aila for making his testing and benchmarking framework publicly available. We also want to thank the anonymous reviewers for their valuable comments.

REFERENCES

Aila, T. and Karras, T. (2010). Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 113–122, Saarbrücken, Germany. Eurographics Association.

Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA. ACM.

Aila, T., Laine, S., and Karras, T. (2012). Understanding the efficiency of ray traversal on gpus – kepler and fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation.

Cazals, F. and Sbert, M. (1997). Some integral geometry tools to estimate the complexity of 3d scenes. Technical report, iMAGIS/GRAVIR-IMAG, Grenoble, France, Departament dInformtica i Matemtica Aplicada, Universitat de Girona, Spain.

Garanzha, K. (2010). Fast ray sorting and breadth-first packet traversal for gpu ray tracing. Oral presentation at EG2010.

Garanzha, K. and Loop, C. (2010). Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Proceedings of the Eurographics*, EG '10, pages 289–298. Eurographics Association.

Hoberock, J., Lu, V., Jia, Y., and Hart, J. C. (2009). Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 173–180, New York, NY, USA. ACM.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 –55.

NVIDIA (2009). Nvidia's next generation cuda compute architecture: Fermi.

NVidia (2012). Nvidia gk110 architecture whitepaper.

Pantaleoni, J., Fascione, L., Hill, M., and Aila, T. (2010). Pantaray: fast ray-traced occlusion caching of massive scenes. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 37:1–37:10, New York, NY, USA. ACM.

Tzeng, S., Patney, A., and Owens, J. D. (2010). Task management for irregular-parallel workloads on the gpu. In Doggett, M., Laine, S., and Hunt, W., editors, *High Performance Graphics*, pages 29–37. Eurographics Association.

Wald, I. (2011). Active thread compaction for gpu path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 51–58, New York, NY, USA. ACM.