Latency considerations of depth-first GPU ray tracing

M. Guthe[1]

[1]Universitaet Bayreuth, Visual Computing

Abstract

Despite the potential divergence of depth-first ray tracing [AL09], it is nevertheless the most efficient approach on massively parallel graphics processors. Due to the use of specialized caching strategies that were originally developed for texture access, it has been shown to be compute rather than bandwidth limited. Especially with recents developments however, not only the raw bandwidth, but also the latency for both memory access and read after write register dependencies can become a limiting factor.
In this paper we will analyze the memory and instruction dependency latencies of depth first ray tracing. We will show that ray tracing is in fact latency limited on current GPUs and propose three simple strategies to better hide the latencies. This way, we come significantly closer to the maximum performance of the GPU.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.7 [Computer Graphics]: Three-DimensionalGraphics and Realism—Raytracing

1. Introduction

Most global illumination algorithms are based on ray tracing and need to compute intersections between rays and the scene geometry. Using hierarchical data structures, the runtime complexity of this is $O(\log n)$, where n is the number of primitives. On GPUs, this comes at the cost of incoherent execution paths and memory access. Both problems have been addressed using specialized traversal methods and the caching strategies of modern GPUs. This way, a depth first ray scene intersection method has been developed, that is not bandwidth limited any more. It has also been shown that implementing complete global illumination algorithms in a single kernel is less efficient than splitting the whole pipeline into a few specialized kernels including a trace kernel.

Due to the recent developments of GPUs, latency has become increasingly important for kernel performance. With the introduction of the Kepler architecture the maximum number of instructions per clock has increased from 2 to 6 per multiprocessor. This means that the number of eligible warps has to be three times as high to fully utilize the compute performance, while the maximum number of warps has only grown from 48 to 64. While latency was not a significant problem for ray tracing on the Tesla architecture, it has become the limiting factor on current GPUs.

We only discuss the performance and optimization of the trace kernel on recent NVIDIA GPUs. Although ray generation and shading are also important for a global illumination renderer, these are not covered in this paper. Nevertheless, any global illumination method that is not based on the less efficient megakernel approach can directly benefit from the proposed improvements.

The main contribution of the paper is an analysis of the memory and read after write latency and the development of strategies to reduce both. The core idea is using instruction level parellelism (ILP) to improve performance. Although GPUs only issue two instructions per warp in parallel, kernels benefit from more independent execution paths due to reduced latency and a higher instruction throughput.

2. Related Work

Real-time ray tracing has received a growing interest in the recent years. Development in this field has been especially spured by advances in general purpose programmable graphics hardware. Here the research can be divided into two fields. The first is the development of efficient traversal algorithms on massively parallel systems and the second one is the design of suitable acceleration data structures.

DOI: 10.2312/egsh.20141013

2.1. Efficient GPU Ray Tracing

Wald et al. [WBB08] proposed a parallel ray tracing method designed for 16 thread wide SIMD Intel CPUs. For NVIDIA GPUs, Aila and Laine [AL09] developed an efficient trace kernel that is not memory bound. Later, Aila et al. [ALK12] extended this work to newer GPUs. In addition, Aila and Karras conducted some considerations for designing efficient ray tarcing hardware [AK10]. Based on observations made by van Antwerpen [vA11], Laine et al. [LKA13] argued that a single mega kernel is unsuitable for ray tracing on GPUs. A consequence of this is that any efficient GPU ray tracing algorithm requires a dedicated trace kernel.

2.2. Acceleration Data Structures

An early acceleration data structure for ray tracing of dynamic scenes was the bounding interval hierarchy [WK06]. It consists of a binary tree with two split planes at each node and an efficient construction algorithm. Similarly, Zhou et al. [ZHWG08] describe an efficient KD-tree construction algorithm running on the GPU. Later, Lauterbach et al. [LGS*09] proposed optimized algorithms to construct bounding volume hierarchies on the GPU. While bounding volume hierarchies (BVHs) have shown to be the most efficient data structure on GPUs due to minimal stack access, they can be further improved using spatial splits [SFD09].

3. Performance Analysis of GPU Ray Tracing

The two performance limiting factors of depth first ray tracing on GPUs are the SIMD efficiency and the latency caused by memory access and instruction dependencies. While several approaches have been made to improve the SIMD efficiency, all methods so far rely on the hardware scheduling to hide the latency. This however only works as long as there are enough eligible warps, i.e. warps where the next instruction can be executed. To fully utilize the computational power of the current Kepler GPUs, at least 6 warps have to be eligible for the 6 schedulers. For the previous architecture, at least two warps need to be eligible at all times.

Table 1 shows the number of executed instructions per clock cycle and the average number of eligible warps, that is an upper bound for the instructions per clock. In addition, we analyze the warp stall reasons to identify possible improvement strategies. While memory latency is inherent to any system where the memory is not located on the same die, the instruction dependency stems from the fact, that it takes several clock cycles to fetch the data from the registers to the cores and store them after the instruction has executed.

The analysis shows that the trace kernel is neither compute nor bandwidth limited on Kepler, as it only utilizes the cores by 39% to 48% and memory bandwidth by 7% to 12%. In case of relatively coherent rays, instruction dependency and memory latency each contribute to about 40% of all issue stalls on Kepler. For incoherent rays, the memory latency causes more than half of the stalls. On Fermi, instruction dependency is the main stall reason, but core utilization is between 53% and 71%. Despite the significant difference between the architectures and tracing coherent and incoherent rays, it becomes clear that depth-first tracing performance is reduced by latency on recent GPU architectures. Analysis of other kernels, i.e. ray setup and hit processing, shows that these are almost always purely bandwidth limited using well over 70% of the maximum device memory bandwidth.

GPU	ray type	IPC	el. w.	fetch	mem.	dep.
GTX	primary	1.41	3.13	32%	10%	52%
480	AO	1.37	3.60	28%	17%	49%
Fermi	diffuse	1.06	2.74	11%	16%	66%
GTX	primary	2.73	4.17	8%	42%	43%
680	AO	2.71	4.20	9%	42%	43%
Kepler	diffuse	2.34	3.37	7%	51%	37%
GTX	primary	2.85	5.22	9%	46%	37%
Titan	AO	2.79	5.15	10%	46%	37%
Kepler	diffuse	2.52	4.65	9%	52%	33%

Table 1: *Latency analysis of depth-first ray tracing using the kernel and hierarchy from [ALK12] for the conference scene (data obtained using the NVIDIA Nsight Monitor).*

As a large part of the issue stalls is caused by read after write (RAW) dependencies, we take a closer look at the latency in this case. On Fermi it typically takes 22 clock cycles [NVI13] to store the result of an operation. On Kepler, this latency was reduced to typically 11 cycles. Together with the time required for the execution and considering the number of parallel warps, this translates to a latency of 44 instructions on Fermi and 88 on Kepler, while it was 6 on Tesla. In addition, Fermi and Kepler issue pairs of instructions for each warp that need to have no RAW dependency.

While the number of independent instructions has increased from 6 to 88, the number of active warps per multiprocessor has only marginally increased. Originally it was 24 on early Tesla and 32 on later GPUs. For Fermi the number was 48 and on Kepler it is now 64. This implies that two independent instructions are theoretically just enough to saturate the warp schedulers of Fermi and Tesla GPUs. In practice however, we cannot assume an optimal warp scheduling due to other factors, like memory latency and caching issues.

4. Reducing Latency

The classical approach to reduce latency is increasing occupancy, i.e. the number of active warps per multiprocessor. The trace kernel however requires 37 registers such that increasing occupancy cannot be achieved without introducing additional local memory access. On the contrary, three more registers could be used on Kepler without reducing occupancy, while 5 would have to be saved to increase it. For Fermi, occupancy changes more fine grained but register spilling degrades the overall performance there as well.

Despite occupancy, there are some other options to reduce latency. First of all, memory latency can be reduced by loading data at once, without conditional branches that make one load dependent on the data of another one. This was already used by Aila and Laine [AL09], where the indices of the child nodes are loaded regardless of the intersection test results. While this wastes bandwidth, memory latency occurs only once for each node of the bounding volume hierarchy.

Another option is to relocate memory access, such that the number of instructions between memory access and using the result is increased. This is automatically performed by the compiler/assembler, but instructions cannot be moved between loop iterations. Loop unrolling can solve this problem and, in addition, reduces the number of branches. This is especially suited for the triangle intersection loop, where unroll counts between 2 and 4 are reasonable.

Reducing the read after write latency is the most challanging task. The general problem is also relevant for sequential programs as all modern CPUs exploit instruction level parallelism. The idea is to split a sequential program into instruction sequences that can be executed in parallel. These are caracterized by not having read after write dependencies between each other. One candidate for ILP is the intersection test of the child nodes. Here, the two intersection tests for a binary tree are already independent and the number of registers is less of a problem than for the triangle test.

4.1. Shallower Hierarchy

If an n-ary tree is used instead of a binary tree, n independent instruction sequences exist. This reduces read after write latency by a factor of $n/2$ compared to binary trees. In addition, the total memory bandwith is slightly reduced, up to 4-ary trees. Interestingly, using wider trees has already been considered [WBB08, AL09], but only in the context of distributing a single ray over several threads.

The 4-ary tree can be generated from a given binary tree by pulling up the children of a child node into the current one. Starting from the root, we could simply remove every second level of the binary tree. If the tree was built using the surface area heuristic, e.g. using [SFD09], we can further optimize the construction similar to the 16-ary trees proposed by Wald et al. [WBB08]. The algorithm starts with the root node and recursively processes the constructred tree. For each node, the child with larger bounding box is first integrated into the current node. Then again that child with the largest bounding box is chosen which could be a grandchild in the original binary tree. In the original method, a special algorithm is used to accumulate 16 triangles in each leaf. As we want to preserve the number of triangles per leaf, we chose a slightly different approach close to the leaf level: If a child only has two leaf nodes, we force adding it to the current node, as this completely removes a node from the hierarchy. When two or more children only have two leaf nodes, we choose the one with the largest bounding box. If no child has two leaves and a child has three leaf nodes, we force adding it if we did not already pull up another child. If we did, we only pull it up, if it is the only non-leaf node.

After intersection testing, we can further improve ILP at the cost of a slightly higher instruction count. For depth-first tracing, we need to sort the intersected nodes according to their distance. Using a sorting network [Bat68], we need 6 compare and swap operations instead of 5 using merge sort. On the other hand, we have two independent instruction paths, which more than compensates for the increased instruction count. We implemented the parallel merge sort algorithm using conditional assignment.

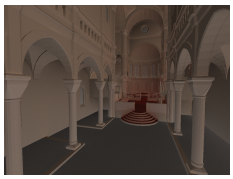
4.2. Unsorted Occlusion Tracing

Given the fact that a part of the computation for hierarchy traversal is required for the distance sorting of intersected child nodes, a further improvement is possible for rays that only require any hit and not the closest one. In this case, we can simply omit sorting and push all intersected child nodes except one onto the traversal stack. This has two advantages: First, the reduced instruction count significantly speeds up traversal and second, the more coherent child node traversal slightly improves the SIMD efficiency when tracing incoherent rays. Note, that for binary trees, sorting is negligible.

5. Results

Table 2 shows the results based on the benchmark from http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/. Note, that the results differ from those published in [ALK12], as mentioned in the source code. The reasons are the different BVH builder and possibly the newer CUDA and driver versions. In addition, we disabled ray sorting. As local memory access is slow on Fermi GPUs, not sorting the nodes for occlusion rays actually reduces the performance due to divergent access. The speedup ranges from -6.7% (primary) to 19.1% (diffuse) for these GPUs using a 4-ary tree and loop unrolling and increases with scene complexity. The overall speedup on Kepler GK104 using all optimizations is 5.1% (diffuse) to 20.1% (AO). Using 4-ary trees only, we measured a speedup of 3.4% (AO) to 11.5% (diffuse). In combination with not sorting the occlusion rays, this improves to 3.7% (primary) to 16.3% (AO). Using loop unrolling of the triangle test only results in a speedup of 1.9% (diffuse) to 4.9% (AO). On the more recent GK110, the speedup increases for incoherent rays, except for very small scenes, and decreases for primary rays. Table 3 shows the number of executed instructions and the cause of latency as for the original code. On Fermi GPUs, the main improvement is due to the improved dual instruction issuing. For Kepler, the number of instructions per clock has significantly increased in all cases. Nevertheless, RAW and memory latency are still present, possibly leaving room for further improvements.






		Conference, 283k tris			Fairy, 174k tris			Sibenik, 80k tris			Sam Miguel, 11M tris		
	Ray type	480	680	Titan	480	680	Titan	480	680	Titan	480	680	Titan
[ALK12] measured (MRays/s)	Primary	272.1	393.5	605.7	150.9	220.3	326.2	252.7	359.8	538.6	70.2	115.1	173.0
	AO	192.4	337.5	527.2	125.5	229.8	353.2	172.9	295.3	448.8	78.8	150.0	231.8
	Diffuse	70.6	134.4	216.4	52.4	100.6	159.2	56.4	114.7	193.6	23.0	45.1	74.1
Improved measured (MRays/s)	Primary	277.4	451.8	688.6	147.0	244.0	354.1	235.8	383.6	534.6	75.1	130.9	188.2
	AO	203.7	387.0	613.3	127.2	264.9	410.1	176.4	354.6	534.1	84.5	170.0	266.2
	Diffuse	78.4	156.7	254.4	56.8	115.1	183.2	63.4	129.7	218.7	27.4	47.4	82.7
speedup	Primary	1.9%	14.8%	13.7%	-2.6%	10.8%	8.6%	-6.7%	6.6%	-0.7%	7.0%	13.7%	8.8%
	AO	5.9%	14.7%	16.3%	1.4%	15.3%	16.1%	2.0%	20.1%	19.0%	7.2%	13.3%	14.8%
	Diffuse	11.0%	16.6%	17.6%	8.4%	14.4%	15.1%	12.4%	13.1%	13.0%	19.1%	5.1%	11.6%

Table 2: *Performance comparison for Fermi (GTX480), and Kepler (GTX680, GTX Titan) using the setup of Aila et al. [ALK12].*

GPU	ray type	IPC	el. w.	fetch	mem.	dep.
GTX	primary	1.52	3.87	28%	10%	55%
480	AO	1.49	4.41	25%	16%	51%
Fermi	diffuse	1.20	3.10	11%	14%	68%
GTX	primary	3.47	6.38	10%	33%	44%
680	AO	3.31	5.92	10%	33%	46%
Kepler	diffuse	3.05	5.08	8%	42%	40%
GTX	primary	3.57	7.89	13%	35%	35%
Titan	AO	3.42	7.45	13%	35%	36%
Kepler	diffuse	3.28	6.84	12%	42%	34%

Table 3: *Latency analysis of our improved depth first ray tracing for the conference scene (cf. Table 1).*

6. Conclusion and Limitations

We have analyzed the performace of depth-first ray tracing on GPUs, specifically considering read after write and memory latency. Based on our observations, we have proposed three simple improvements that result in a speedup of up to 20% on recent GPUs, depending on scene and ray type. In addition, the nodes need about 27% less graphics memory.

The current implementation is limited to bounding volume hierarchies. In principle, it would be possible to use a similar approach for other acceleration data structures. For bounding interval hierarchies [WK06] and KD-trees, we could aggregate 6 or 3 planes respectively into one node. This way we could also increase ILP, since the plane intersection tests can be computed independently of each other. Finally, we do not attempt to solve the problem of thread divergence. Nevertheless, the thread utilization slightly improved by about 1% for all ray types on Fermi and Kepler.

References

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics 2010* (2010), pp. 113–122. 2

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 145–149. 1, 2, 3

[ALK12] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012. 2, 3, 4

[Bat68] BATCHER K. E.: Sorting networks and their applications. In *Proceedings of the 1968, spring joint computer conference* (1968), AFIPS '68, ACM, pp. 307–314. 3

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2 (2009), 375–384. 2

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of High-Performance Graphics 2013* (2013). 2

[NVI13] NVIDIA: *NVIDIA CUDA Programming Guide 5.5*. NVIDIA, 2013. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, ACM, pp. 7–13. 2, 3

[vA11] VAN ANTWERPEN D.: Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), HPG '11, ACM, pp. 41–50. 2

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting Rid of Packets – Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2008* (2008). 2, 3

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149. 2, 4

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5 (2008), 126:1–126:11. 2