Eurographics Symposium on Parallel Graphics and Visualization (2013)
F. Marton and K. Moreland (Editors)

Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays

D. Wodniok[1], A. Schulz[2], S. Widmer[1] and M. Goesele[1]

[1]Graduate School Computational Engineering, TU Darmstadt, Germany [2]TU Darmstadt, Germany

Abstract

With CPUs moving towards many-core architectures and GPUs becoming more general purpose architectures, path tracing can now be well parallelized on commodity hardware. While parallelization is trivial in theory, properties of real hardware make efficient parallelization difficult, especially when tracing incoherent rays. We investigate how different bounding volume hierarchy (BVH) and node memory layouts as well as storing the BVH in different memory areas impacts the ray tracing performance of a GPU path tracer. We optimize the BVH layout using information gathered in a pre-processing pass applying a number of different BVH reordering techniques. Depending on the memory area and scene complexity, we achieve moderate speedups.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

1. Introduction

Many applications (especially in a global illumination context) spend a substantial amount of time tracing rays through a scene. Theoretically, ray tracing is embarrassingly parallel as different rays can be traced independently. On multi-core systems it is implemented in a straightforward manner by letting each thread process its own batch of rays. Further parallelization can be achieved by taking advantage of SIMD capabilities of multi-core architectures or the massive parallelism of many-core architectures such as GPUs which is the focus of this paper. Efficient parallelization on SIMD architectures is, however, much harder due to incoherent rays whose origins and directions vary widely. Tracing incoherent rays requires traversing different paths through the acceleration structure, resulting in incoherent memory accesses since different nodes are traversed and different primitives are tested. As incoherent rays form an absolute majority they pose a serious challenge. It is thus important to carefully choose where (i.e., in which memory area) and how to layout data and to use special instructions to unlock the hardware's full potential. GPUs typically achieve their massive parallelism by a wide SIMD width (in our case 32 lanes) yielding the following challenges for an efficient implementation:

- *SIMD efficiency (ratio of active to total number of SIMD lanes)*: Especially for incoherent rays, the SIMD efficiency can be low since the number of acceleration structure nodes that a ray has to test in order to find the nearest intersection can vary significantly. Some rays terminate earlier than others, leaving a number of SIMD lanes idle.
- *SIMD divergence:* Even if all SIMD lanes have active rays, some may want to test geometry while others are still traversing the acceleration structure. In that case the execution paths of the lanes diverge and SIMD efficiency is temporarily lower until the execution paths re-converge.
- *Memory bandwidth/latency:* As incoherent rays access many different memory addresses, the number of different cache lines accessed increases, too. On current GPUs only a single cache line can be read at a time. In the worst case, each SIMD lane accesses a different cache line, resulting in serialization of the accesses and increased latency.

We focus on the memory effects of tracing incoherent rays on NVIDIA GPUs. "Real-world" incoherent rays are generated by a basic path tracer. Presumably, the cache efficiency when tracing incoherent rays is low. We analyze our GPU path tracer and the effects of rearranging the nodes of the acceleration structure (a bounding volume hierarchy (BVH))

© The Eurographics Association 2013.

DOI: 10.2312/EGPGV/EGPGV13/057-064

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

on cache efficiency using previously recorded access statistics. Our goal is to increase cache hit rates and reduce the number of cache lines read per access. Our contributions are the analysis of the cache behavior when tracing incoherent rays in real-world scenarios. In particular, we show that the commonly used depth first search memory layout performs worst and we present several alternative layouts. None of those performs, however, best in all cases.

2. Related Work

Plunkett et al. [PB85] first implemented ray tracing on a vector processor. With the widespread availability of SIMD architectures, research on efficiently implementing ray tracing on such architectures proliferated. Wald et al. [WSBW01] presented an SIMD implementation of a ray tracer using Intel's SSE instructions. Their packet tracing technique exploits ray coherence by tracing rays in packets of SIMD width size (4 for SSE) which achieves good caching behavior and yields a speed-up of roughly half an order of magnitude. Memory bandwidth is reduced by loading a node only once for packets of 4 rays. Later, Wald et al. [WBS07] proposed a combination of packet and frustum tracing. Using a packet size larger than the native SIMD width and different optimizations, they reported 3.3-10.7× speed-ups over the native SIMD packet size. Purcell et al. [PBMH02] first presented a complete GPU ray tracing pipeline which had to map all computations to the GPU's rendering pipeline. First ray tracing implementations using NVIDIA's CUDA [NVIa] include Günther et al. [GPSS07] and Popov et al. [PGSS07].

Aila et al. [AL09] presented different trace loop organizations. The key difference to packet tracing is that essentially single ray tracing in an SIMD manner is performed using scatter/gather operations and hardware SIMD divergence handling. Rays only visit nodes which they actually intersect, but memory accesses become more incoherent. The *speculative while-while* loop organization performed best. It processes rays in one of two phases at a time: traversal or triangle intersection. During traversal, an SIMD lane traverses the tree until it finds a leaf. If some SIMD lanes have not yet found a leaf, the SIMD lane stores its found leaf and speculatively continues traversal until every SIMD lane found a leaf. Though this may result in superfluous memory accesses, the memory bandwidth overhead is generally low enough and the higher SIMD efficiency results in a 10% lower runtime.

Ray grouping and reordering Simply grouping rays into packets only works well for coherent rays. Therefore, techniques that extract hidden ray coherence using regrouping or reordering ray packets have been developed. Some of which defer ray processing at certain queue points [PKGH97, NFLM07]. Queue processing is scheduled to minimize and amortize cache misses, and reduce memory bandwidth demand when computing intersections with scene geometry. Mansson et al. [MMAM07] investigated several regrouping algorithms for secondary rays. Further strategies are regrouping by ray type [BEL*07], by hashes generated from a ray's geometry [GL10], by approximations of ray hitpoints [MBK*10], or by ray packet filtering [BWB08].

Cache efficient algorithms There are two types of cache-efficient algorithms: *Cache-aware* algorithms explicitly use prior knowledge about caches (e.g., cache-line size). *Cache-oblivious* algorithms [Pro99] only assume that a cache is present without knowing any of its properties.

Aila et al. [AK10] presented a massively parallel hardware architecture which is to some extent based on NVIDIA's Fermi GPU architecture. They developed a cache-aware traversal algorithm specifically designed for this architecture, which achieves up to a 90% reduction in total memory bandwidth for tracing incoherent rays. A major assumption of the algorithm is, that the L1 cache can access multiple cache lines per clock (otherwise L1 fetches are a serious bottleneck). To our knowledge, L1 caches of current hardware still do not have such capabilities.

Wald et al. [WSBW01] and Havran [Hav99] optimized cache efficiency by either storing just one child pointer or completely omitting them through special node arrangements, thus reducing node size. Kim et al. proposed a random-accessible compressed BVH [KMKY10] with context-based arithmetic coding. Combined with random accessible compressed triangle meshes [YL07] they achieve an average rendering time improvement of 35-54% due to increased cache efficiency and hit rate as more nodes fit into the cache. Yoon et al. [YM06] proposed a cache-oblivious BVH layout for collision detection which applied to a k-d tree for ray tracing, resulted in a 77%-180% runtime improvement. Van Emde Boas [vEB75] derived a cache-oblivious tree memory layout built by recursively subdividing the height of the tree in half yielding a number of subtrees per step. This clusters nodes and is beneficial for caches since traversing a node causes nodes of the subtree below the current node to be loaded into the cache which are likely to be traversed as well. Gil et al. [GI99] proposed a dynamic programming algorithm which allocates tree nodes to memory pages, minimizing the number of memory pages visited and the number of page faults. Bender et al. [BDFC02] present faster but approximate algorithms for solving the same problem in a cache-oblivious manner. Multi-branching BVHs [EG08,DHK08,WBB08] improve cache efficiency by simply requiring less memory thus reducing bandwidth demand and keeping more nodes in the cache. Further, contrary to packet tracing a single ray is tested against SIMD width size number of bounding boxes and triangles, which is beneficial for incoherent rays but slower for coherent rays compared to packet tracing.

3. GPU Hardware Details / Test Setup

We made minor modifications to Wong et al.'s microbenchmarking code [WPSAM10] and ran it on an NVIDIA

© The Eurographics Association 2013.

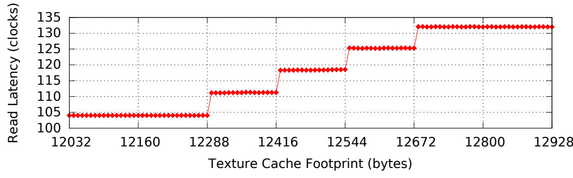

Figure 1: *Average texture memory L1 cache latency in cycles of a Geforce GTX 680 revealing the cache properties.*

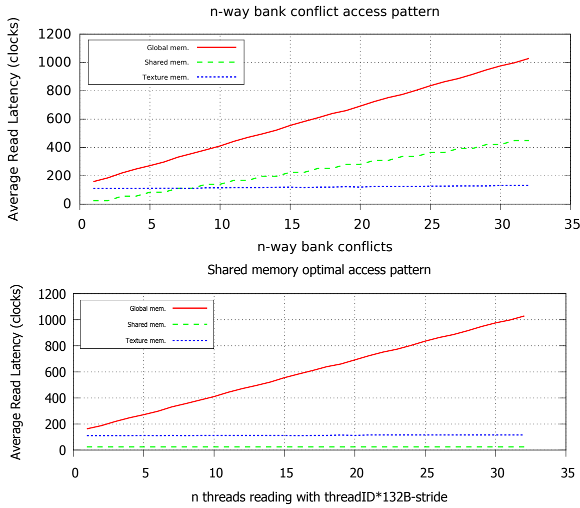

Figure 2: *Average read latency in cycles of a Geforce GTX 680 using a memory access pattern which would cause n-way bank conflicts (top) in shared memory and a shared memory optimal access pattern (bottom).*

Geforce GTX 680 to determine cache properties and access latencies. We evaluated the GPUs and performed all benchmarks using CUDA Version 5.0 [NVIa], the NVIDIA Nsight Visual Studio Edition 3.0 Beta, and the CUDA Profiler Tools Interface (CUPTI). The test system is equipped with an Intel Core i7-960, 32 GB RAM, and an NVIDIA Geforce GTX 480 (primary device) as well as a GTX 680 with 2 GB RAM (headless device). All tests were performed on the GTX 680 using driver version 306.94.

3.1. Cache properties

The Geforce GTX 680 consists of eight Streaming Multiprocessors (SMX) with 192 CUDA cores each. It provides 2048 MB of global/texture memory, 16, 32 or 48 KB of shared memory or L1 cache for local memory (depending on the runtime configuration) and 65536 registers per SMX. The fetch latency for a global memory load of 4 bytes hitting the L2 cache takes ≈160 cycles while a miss results in a latency of ≈290 cycles.

Given the average texture memory cache access latency in Figure 1 retrieved from the micro-benchmark, we can deduce that the texture cache size is 12 KB, consisting of 4 cache sets with a cache line size of 128 bytes and is 24-way set associative. L1 hit latency for reading 4 bytes is ≈105 cycles, L2 hit latency is ≈266 cycles and missing both L1 and L2 incurs a latency of ≈350 cycles. Figure 2 shows the latency of two different access patterns evaluated in three different memory areas. One causes n-way bank conflicts in shared memory and the other is optimal for shared memory. We can see that while the patterns are bad for either global or both global and shared memory, texture memory performs almost equally well with either access pattern.

4. GPU Path Tracer Implementation

The GPU path tracer implementation essentially follows van Antwerpen's streaming design [vA11]. A large batch of 2^{20} samples is processed in parallel using one thread per sample. Sample paths are iteratively extended. Samples that have finished computation due to termination via Russian Roulette or because their paths have escaped the scene are removed from the stream in each iteration and the remaining samples are compacted. To keep the amount of work constant, each finished sample is regenerated by appending a new sample to the compacted stream. As previously observed [vA11], appending regenerated samples exploits primary ray or even higher order ray coherence due to specular reflection or refraction and improves SIMD efficiency. To include such effects on performance in our analysis we use the Ashikhmin-Shirley BRDF [AS00] to model scenes with materials ranging from diffuse to highly specular as well as refractive rough surfaces [WMLT07]. For ray traversal, we use the fast speculative-while-while traversal kernel design [AL09]. As underlying ray traversal acceleration structure we use a high quality split-BVH [SFD09]. The whole path tracer is implemented in four kernels (sample initialization/regeneration, ray tracing, path extension, connection validation). As tracing is done in a designated kernel, all statistics are only affected by ray traversal and not by other computations such as shading.

5. BVH Data Structures and Layouts

For our analysis we focus on binary bounding volume hierarchies with axis aligned bounding boxes and include several memory layouts for the node data and the tree itself.

5.1. Node Layouts

The classic BVH node data structure stores a bounding volume along with pointers to its children. We follow Aila et al. [AL09], i.e., a node does not store its bounding box, but the bounding boxes of its children. Both children are fetched and tested together, which is more efficient for GPUs due to increased instruction level parallelism and allows rough front to back traversal. Depending on the data layout, the size of such a node is at least 56 bytes (2 float values for minimum/maximum per dimension and child plus pointers).

© The Eurographics Association 2013.

We implemented one array-of-structures (AoS) layout and two structure-of-arrays (SoA) layouts:

- **AoS:** 64 bytes, including 8 bytes padding (fitting 2 nodes in one 128B cache line)
- **SoA32_24:** $32+24$ bytes, min/max x/y both children, min/max z both children and pointers, plus 8 bytes padding (fitting 4 nodes across 2 128B cache lines)
- **SoA16_8:** $3\times16+8$ bytes, min/max x/y child1, min/max x/y child2, min/max z both children, pointers (fitting 8 nodes across 4 128B cache lines)

We also analyzed an SoA8 layout which fitted 16 nodes in 7 cache lines. As it consistently performed much worse than the other layouts, we excluded it from our experiments.

5.2. Tree Layouts

A tree layout describes how nodes are grouped in memory. We analyzed six different tree layouts. The first four layouts are two common layouts and two cache-efficient layouts. We further propose two more layouts. The idea behind them is to compute a path traced image at a relatively low sample rate as a pre-process, recording the number of accesses for each BVH node. We then use the access statistics to guide the two layouting methods. Layouts not using statistics are:

- **Depth-first-search (DFS):** Nodes are ordered as visited by a pre-order traversal. This layout performs best with coherent rays since a cache line is potentially filled with nodes on the path to the leaf.
- **Breadth-first-search (BFS):** Nodes are ordered as visited by a breadth-first traversal visiting the left child node first. This fits best for rays traversing neighboring branches.
- **van Emde Boas (vEB):** A cache-oblivious tree layout [vEB75] described in Section 2.
- **COLBVH (COL):** A cache-oblivious tree layout mainly used for collision detection [YM06] but also applicable to raytracing. Beginning with all n nodes in a root cluster, the tree is recursively decomposed into clusters of $\lceil\sqrt{n+1}-1\rceil$ nodes. Nodes are merged into root clusters depending on their access probability computed from the ratio of the surface areas of its grand-parent and parent.

Next we describe our two proposed layouts depending on node access statistics which use a threshold p:

- **Swapped subtrees (SWST):** Swap the sub-trees of a node in a depth-first layout if the fraction of left child accesses compared to all child accesses is below $p\in[0,0.5]$. Left children of the nodes form a path whose nodes are accessed the most and are spread over fewer cache lines.
- **Treelet based DFS/BFS (TDFS/TBFS):** A treelet is a connected sub-tree of a BVH. For this layout treelets of nodes that were accessed above a certain threshold are built. This decomposes the BVH into treelets whose nodes are accessed the most. The algorithm works with two queues: a merge queue and a deferred queue. The merge queue contains nodes which will be added to the current treelet and the other queue contains nodes which are deferred for creating additional treelets. Initially the current treelet and deferred queue are empty, and the merge queue contains the BVH root. Nodes are removed from the merge queue and added to the current treelet as long as the merge queue is not empty. When a node is removed its children are added to one of the queues. If the percentage of rays that continued to descend to a child node is larger than $p\in[0,1]$ the child is added to the merge queue, otherwise to the deferred queue. If the merge queue is empty, a new treelet is created by moving a node from the deferred queue to the merge queue and repeating the process. Once no more nodes are present in either queue the algorithm is done. The internal memory layout of a treelet can be chosen freely. By always adding nodes just to the front or the back of the merge queue we automatically obtain a treelet in DFS or BFS order. Finally the node order of the whole tree is obtained by lining up the nodes of all treelets. Thus treelets are only used as a means for grouping nodes and are not stored explicitly.

Scenes	Triangles	Nodes	Size (MB)
crytek-sponza	262269	99127	6.05
kitchen	425504	150219	9.17
hairball	2880012	1021548	62.35
san-miguel	7880512	2723017	166.2

Table 1: *Scenes used for benchmarking.*

Note that there are other possible treelet construction algorithms such as the construction algorithm described by Aila et al. [AK10]. As mentioned previously, this approach is to our knowledge not yet supported by current hardware and therefore not included in our analysis.

6. Evaluation

We evaluated the performance of the BVH and node layouts on four different scenes of varying complexity and materials (see Table 1). The Kitchen scene consists of a mixture of diffuse, glossy and translucent materials. The well known hairball scene uses a nearly specular refractive material.

Several different metrics are computed using the event counters from CUPTI. Some of them can be found in the CUPTI User's Guide, others were deduced from values in Nsight Visual Studio Edition and reconstructed with events from CUPTI. A short explanation of each metric follows (see [SWWG13] for more details):

- **Runtime**, trace kernel runtime in milliseconds, measured using CUPTI's activity API.
- **GM/L1 $\leftarrow$ L2 cache load hit rate**, percentage of global memory (GM) loads that hit in L2 cache. It is slightly diluted as the event counters include both global and local memory loads. (Local memory makes up $\sim$10% of the sum of global and local memory traffic.)

© The Eurographics Association 2013.

- **Tex cache hit rate**, percentage of texture memory loads that hit the texture memory cache.
- **Instruction replay overhead**, percentage of instructions that were issued due to replaying memory accesses, such as cache misses (lower is better).
- **SIMD efficiency (warp execution efficiency)**, percentage of average active to total number of threads per warp.
- **Branch efficiency**, ratio of non-divergent branches to all branches (SIMD divergence).
- **Load efficiency**, the ratio of requested memory load throughput to actual memory load throughput.

6.1. Baseline performance analysis

The baseline BVH is laid out in DFS order and stores nodes in AoS format. The AoS node format was chosen because Aila et al. [AL09] are using it in their GPU ray traversal routines which are one of the fastest. Tree nodes are accessed via global memory and geometry via texture memory.

Figure 3 gives an overview of the baseline performance for each scene and render loop iteration. We can see the effect of incoherent rays immediately after the first iteration. Runtime increases by 5× from 2.85 ms to 14.36 ms for the kitchen scene. The number of primary rays per batch drops to 20%. Despite the huge number of incoherent rays the branch efficiency only decreases slightly to 85%. This indicates that threads in a warp mostly agree on their execution path. The cache hit rate does not suffer noticeably. The amount of data transferred between the different caches in the memory hierarchy shows that most data requests are serviced by caches. We notice a significantly lower load and SIMD efficiency after the first iteration, which later on increases with the number of primary rays per batch. The achieved occupancy stays relatively close to the theoretical maximum of 75% which means that work is well spread over the GPU's multiprocessors. For the hairball scene we made a similar observation. Runtime increases from 2.49 ms up to 75.29 ms after a significant number of rays did not hit the environment map but the geometry. Especially the load efficiency as well as the SIMD efficiency collapses due to a combination of the incoherent memory access pattern and the depth complexity of the BVH tree.

Furthermore we evaluated the gain of storing the BVH in texture memory as was proposed by Aila et al. [ALK12] who did not state expected speed-ups.

6.2. BVH and node layouts

Tables 2 and 3 show a ranking of all BVH and node layout combinations which were accessed via global memory or texture memory. The ranking is performed w.r.t. the average achieved speedup compared to the DFS layout in the respective memory area. The SWST, TDFS and TBFS layouts require a threshold probability. We have tested a number of different values to find the best performing one. The best threshold is required to perform well for all scenes in our data set so that its performance extends to unknown data sets. We use the sum of the scene runtimes to measure the performance of a threshold and choose the best performing ones. The determined thresholds are stated next to the respective BVH layout names in the tables. Following, we will compare the best performing combinations of threshold, BVH and node layout in each memory area to the other introduced BVH layouts.

Global Memory Overall, the node layout has the biggest impact on performance. The AoS layout performs best followed by SoA32_24 and SoA16_8, except for the kitchen scene where it is roughly the other way around. Performance differences of the AoS and SoA16_8 based layouts range from 10% – 35%. For AoS performance differences between tree layouts are only up to 2%. Only in the san-miguel scene the treelet DFS layout manages to achieve 6%. Our baseline already performs quite well. For the other node layouts tree layouts improve performance up to 9%, i.e., without access statistics the simple BFS layout performs best for all node layouts, followed by the more complex vEB layout, whereas DFS on average performs worst for all node layouts. Interestingly the AoS SWST layout, which is basically a DFS layout, on average performed slightly worse than DFS. The better performing tree layouts have a number of effects. On average, slightly less data is transferred by global load instructions, the average number of transactions per global load request is decreased and instruction replay overhead dropped minimally (see [SWWG13] for details). This is reflected in a higher IPC count because fewer instructions have to be issued due to memory replays. We can see the impact that the SoA32_24 node layout has on the caches in a lower L2 global load hit rate. A L2 cache miss is more expensive and displaces twice as many nodes than when using the AoS node layout. The situation is similar but exacerbated for the SoA16_8 layouts which all exhibit worse performance due to their even higher miss penalty which results in very low cache hit rates.

Texture Memory Again we can see that the node layout has the biggest performance impact with the AoS layout performing best followed by SoA32_24 and SoA16_8. Performance differences of the AoS and SoA16_8 based layouts range from 17% – 50%. The best performing combination is the TDFS BVH layout with a threshold of 0.6 using the AoS node layout. It is only marginally faster than the baseline layout combination. Interestingly, there is an increase in the amount of data transferred across all scenes and layout combinations though the transaction size of both the texture cache and the global memory L2 cache is 32B. The only explanation we can provide for a lower traffic size of global memory is superior broadcasting compared to texture memory.

Comparison If we compare the runtime of the best layout combinations in texture and global memory, we can ob-

© The Eurographics Association 2013.

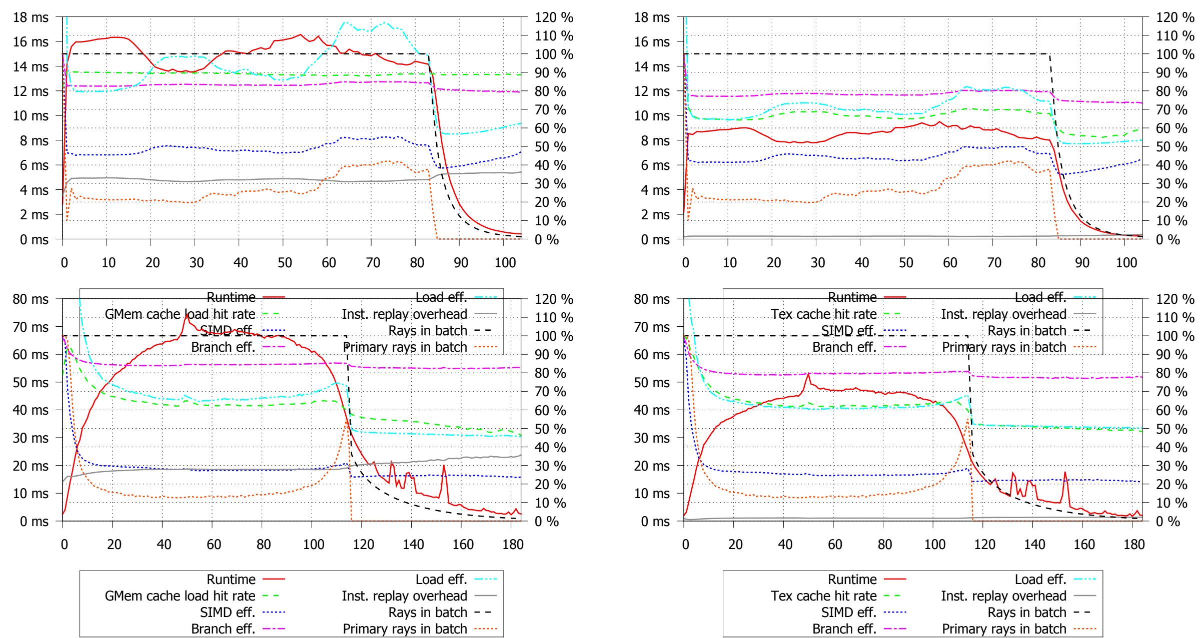

Figure 3: *Trace kernel profiling graphs of the baseline for the kitchen (top row) and hairball scene (bottom row) using a Geforce GTX 680. Resolution is 1024x768 with 32spp. Nodes have AoS format and are stored in DFS order. The figure shows the runtime behavior (left y axis) and GPU metrics (right y axis) over all rendering loop iterations (x axis). BVH nodes are either stored in global memory (left column) or texture memory (right column).*

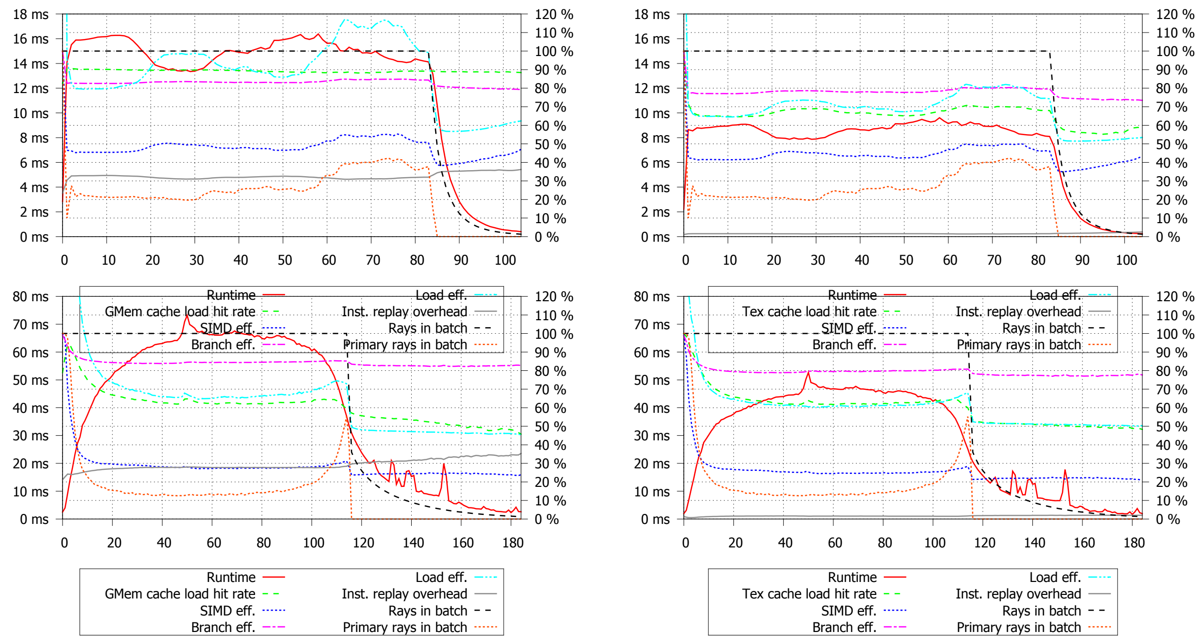

Figure 4: *Trace kernel profiling graphs using the best performing TDFS layout with a threshold of 0.6 and AoS node layout using a Geforce GTX 680. Used scenes are again kitchen (top row) and hairball (bottom row). Resolution is 1024x768 with 32spp. The figure shows the runtime behavior (left y axis) and GPU metrics (right y axis) over all rendering loop iterations (x axis). BVH nodes are either stored in global memory (left column) or texture memory (right column).*

© The Eurographics Association 2013.

serve that using texture memory is approximately 30 – 40% faster. Even some of the slowest layout combinations in texture memory are faster than the best layout combinations in global memory. Our baseline layout with nodes in texture memory also outperforms all layout combinations in global memory by 25% – 38%. The reason why texture memory performs better is not entirely clear. As we have seen in Section 3, the average access latency of texture memory is consistently lower than for global memory and for access patterns which cause very high latency in global and shared memory, the texture memory's latency increases only by a comparably small amount. We presume that this property of the texture memory cache, in conjunction with quite possibly other unknown hardware details, let it deal very well with incoherent memory accesses such that it is on average faster than the L2 global memory cache. Contrary to [ALK12] our path tracer benefited from using texture memory for loading nodes when run on a Fermi GPU (see [SWWG13]).

7. Conclusion

We have presented a number of different BVH layout schemes and analyzed their performance on tracing "real-world" distributions of incoherent rays. Two schemes make use of information gathered in a pre-processing pass over the BVH. Our TDFS layout had the best average speedup in global and texture memory. In global memory we have achieved a runtime reduction by 1%–6%. We gained a 30%–40% runtime reduction compared to the baseline in global memory when the BVH is stored in texture memory similar to Aila et al. [ALK12]. But also accessing the baseline in texture memory, an improvement of only 0.5%–4.0% was observable for the TDFS layout. The common DFS layout performed worst for all node layouts in both memory areas. Excluding layouts that use statistics the equally simple to construct BFS layout on average performed best and similar to the TDFS layout.

There are several possibilities for future work. We have not included the performance impact on tracing coherent rays with our analyzed layouts. As optimizing the BVH layout and ray grouping and reordering techniques are orthogonal one might investigate if there is any synergy when using both techniques together. Increasing SIMD efficiency will result in more memory accesses being issued at a time which may be beneficial for our reordering techniques. The dynamic fetch kernel from Aila et al. [ALK12] which replaces terminated rays when SIMD efficiency drops below a certain percentage could be used to investigate this. The GK110 [NVIb] features a 48 KB read-only data cache, which is the same as the texture memory cache. It would be interesting to see the effects of a much larger texture cache. As kd-tree nodes are much smaller than BVH nodes more nodes fit into cache lines. Though this should increase cache hits, more nodes get evicted on cache misses. It would be interesting to see which effect outweighs the other and what performance gains can be achieved with different tree layouts.

Acknowledgments The work of S. Widmer and D. Wodniok is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt. Hairball scene courtesy of Samuli Laine. Crytek-sponza scene courtesy of Frank Meinl. San-miguel scene courtesy of Guillermo M. Leal Llaguno.

References

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proc. HPG* (2010). 2, 4

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. HPG* (2009). 2, 3, 5

[ALK12] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum.* Tech. Rep. NVR-2012-02, 2012. 5, 7

[AS00] ASHIKHMIN M., SHIRLEY P.: An anisotropic phong brdf model. *J. Graph. Tools* (2000). 3

[BDFC02] BENDER M. A., DEMAINE E. D., FARACH-COLTON M.: Efficient tree layout in a multilevel memory hierarchy. In *Proc. ESA* (2002). 2

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: Packet-based whitted and distribution ray tracing. In *Proc. Graphics Interface* (2007). 2

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive ray packet reordering. In *Proc. IEEE IRT* (2008). 2

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *CGF* (2008). 2

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proc. IEEE IRT* (2008). 2

[GI99] GIL J., ITAI A.: How to pack trees. *Journal of Algorithms* (1999). 2

[GL10] GARANZHA K., LOOP C. T.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *CGF* (2010). 2

[GPSS07] GUNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on gpu with bvh-based packet traversal. In *Proc. IEEE IRT* (2007). 2

[Hav99] HAVRAN V.: Analysis of cache sensitive representation for binary space partitioning trees. *Informatica (Slovenia)* (1999). 2

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: Racbvhs: Random-accessible compressed bounding volume hierarchies. *IEEE TVCG* (2010). 2

[MBK*10] MOON B., BYUN Y., KIM T.-J., CLAUDIO P., KIM H.-S., BAN Y.-J., NAM S. W., YOON S.-E.: Cache-oblivious ray reordering. *ACM Trans. Graph.* (2010). 2

[MMAM07] MANSSON E., MUNKBERG J., AKENINE-MOLLER T.: Deep coherent ray tracing. In *Proc. IEEE IRT* (2007). 2

[NFLM07] NAVRATIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proc. IEEE IRT* (2007). 2

[NVIa] NVIDIA: CUDA Compute Unified Device Architecture. www.nvidia.com/object/cuda_home_new.html. 2, 3

[NVIb] NVIDIA: Kepler GK110 whitepaper. nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. 7

© The Eurographics Association 2013.

GMem													
BVH lay.	node lay.	crytek-sponza			kitchen			hairball-glass			san-miguel		
		R	H	SZ	R	H	SZ	R	H	SZ	R	H	SZ
TDFS 0.6	AoS	581.7	86.7	94.3	1386.2	89.1	218.6	7504.9	60.5	948.8	2071.4	71.4	287.4
BFS	AoS	583.3	86.5	94.3	1364.6	89.0	218.6	7462.2	60.4	948.8	2131.1	70.7	287.6
TBFS 0.3	AoS	582.6	86.6	94.3	1371.8	89.1	218.6	7469.5	60.4	948.8	2142.8	70.8	287.7
vEB	AoS	582.5	86.6	94.3	1374.6	89.0	218.6	7469.4	60.6	948.6	2165.4	70.7	287.7
COL	AoS	582.5	86.7	94.3	1385.5	89.1	218.6	7539.5	60.6	949.1	2166.9	70.9	287.7
DFS	AoS	583.5	86.6	94.3	1394.9	89.0	218.6	7576.0	60.7	949.1	2205.3	70.6	287.9
SWST 0.5	AoS	582.2	86.8	94.3	1391.9	89.1	218.6	7638.8	60.8	949.3	2267.9	71.0	288.0
BFS	SoA32_24	581.0	85.1	94.1	1310.1	88.4	217.8	9099.2	55.3	950.3	2683.3	64.4	287.4
vEB	SoA32_24	583.9	85.3	94.2	1355.1	88.7	217.9	9059.4	55.7	950.1	2824.0	64.6	287.6
COL	SoA32_24	585.1	85.4	94.2	1335.2	88.8	217.9	9269.1	55.4	950.7	2859.4	64.9	287.7
DFS	SoA32_24	595.3	84.6	94.2	1357.9	88.3	217.9	9320.0	54.8	950.7	2932.9	63.3	288.2
BFS	SoA16_8	637.4	77.2	93.4	1346.1	81.7	213.4	10747.6	38.1	942.2	3270.6	48.3	285.3
vEB	SoA16_8	641.3	77.9	93.3	1364.4	83.4	215.3	10555.1	40.0	942.8	3376.6	48.6	286.1
COL	SoA16_8	651.3	77.8	93.2	1371.4	84.0	215.3	10780.1	39.6	943.4	3437.5	49.1	286.3
DFS	SoA16_8	680.1	75.8	93.6	1436.5	82.0	217.1	10969.3	38.3	943.0	3667.2	45.4	287.1

Table 2: *Ranking of layout combinations w.r.t. average speedup in global memory. Runtime (R) in milliseconds, cache hit rate (H) in percent and transferred data size (SZ) in gigabytes are shown.*

TMem													
BVH lay.	node lay.	crytek-sponza			kitchen			hairball-glass			san-miguel		
		R	H	SZ	R	H	SZ	R	H	SZ	R	H	SZ
TDFS 0.6	AoS	372.4	76.5	136.3	812.6	65.6	268.1	5369.4	59.8	1053.8	1300.4	61.0	337.3
BFS	AoS	373.1	76.4	136.3	807.7	65.6	268.1	5356.7	59.9	1053.8	1315.0	61.2	337.3
TBFS 0.2	AoS	373.1	76.5	136.3	810.2	65.7	268.1	5359.3	59.9	1053.8	1315.0	61.2	337.3
vEB	AoS	373.0	76.5	136.3	806.7	65.5	268.1	5357.3	59.8	1053.8	1326.4	61.1	337.3
COL	AoS	373.4	76.5	136.3	804.2	65.6	268.1	5386.1	59.8	1053.8	1334.8	61.0	337.3
SWST 0.4	AoS	373.8	76.5	136.3	805.1	65.5	268.1	5394.3	59.9	1053.8	1353.1	60.9	337.2
DFS	AoS	374.2	76.4	136.3	806.2	65.4	268.1	5394.9	59.8	1053.8	1356.4	60.9	337.3
BFS	SoA32_24	412.9	73.0	136.4	845.6	61.5	268.3	6868.8	56.6	1056.5	1877.0	56.4	337.3
vEB	SoA32_24	417.0	73.0	136.4	837.4	61.1	268.3	6839.6	56.4	1056.6	1955.8	56.2	337.4
COL	SoA32_24	417.1	73.0	136.4	852.9	61.2	268.3	6956.5	56.3	1056.5	1978.8	56.2	337.4
DFS	SoA32_24	423.4	72.7	136.4	852.7	60.7	268.3	6971.4	56.3	1056.5	2023.4	55.9	337.3
BFS	SoA16_8	497.0	60.2	135.7	988.0	43.9	264.8	9570.8	36.7	1048.7	2837.8	34.7	335.0
vEB	SoA16_8	495.3	61.1	135.6	981.6	43.9	266.5	9261.5	37.8	1048.9	2932.9	35.2	335.3
COL	SoA16_8	506.1	61.2	135.6	973.1	44.2	266.6	9515.5	37.7	1048.6	2999.3	35.3	335.4
DFS	SoA16_8	535.7	60.5	135.7	1042.8	42.9	267.6	9663.3	38.4	1049.5	3229.9	35.1	335.9

Table 3: *Ranking of layout combinations w.r.t. average speedup in texture memory. Runtime (R) in milliseconds, cache hit rate (H) in percent and transferred data size (SZ) in gigabytes are shown.*

[PB85] PLUNKETT D., BAILEY M.: The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Comput. Graph. Appl.* (1985). 2

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *Proc. SIGGRAPH* (2002). 2

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-tree traversal for high performance GPU ray tracing. *CGF* (2007). 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proc. SIGGRAPH* (1997). 2

[Pro99] PROKOP H.: *Cache-Oblivious Algorithms*. Master's thesis, MIT, 1999. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proc. HPG* (2009). 3

[SWWG13] SCHULZ A., WIDMER S., WODNIOK D., GOESELE M.: *Extended Data Collection: Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays*. Tech. Rep. 13rp003-GRIS, 2013. 4, 5, 7

[vA11] VAN ANTWERPEN D.: Improving simd efficiency for parallel monte carlo light transport on the GPU. In *Proc. HPG* (2011). 3

[vEB75] VAN EMDE BOAS P.: Preserving order in a forest in less than logarithmic time. In *Proc. SFCS* (1975). 2, 4

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs. In *Proc. IEEE IRT* (2008). 2

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.* (2007). 2

[WMLT07] WALTER B., MARSCHNER S. R., LI H., TORRANCE K. E.: Microfacet models for refraction through rough surfaces. In *Proc. EGSR* (2007). 3

[WPSAM10] WONG H., PAPADOPOULOU M.-M., SADOOGHI-ALVANDI M., MOSHOVOS A.: Demystifying GPU microarchitecture through microbenchmarking. In *Proc. ISPASS* (2010). 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. *CGF* (2001). 2

[YL07] YOON S.-E., LINDSTROM P.: Random-accessible compressed triangle meshes. *IEEE TVCG* (2007). 2

[YM06] YOON S.-E., MANOCHA D.: Cache-efficient layouts of bounding volume hierarchies. *CGF* (2006). 2, 4

© The Eurographics Association 2013.