CPU-Style SIMD Ray Traversal on GPUs

Alexander Lier
Computer Graphics Group, University of Erlangen-Nuremberg

Marc Stamminger
Computer Graphics Group, University of Erlangen-Nuremberg

Kai Selgrad
Computer Graphics Group, University of Erlangen-Nuremberg

ABSTRACT

In this paper we describe and evaluate an implementation of CPU-style SIMD ray traversal on the GPU. We show how spreading moderately wide BVHs (up to a branching factor of eight) across multiple threads in a warp can improve performance while not requiring expensive pre-processing. The presented ray-traversal method exhibits improved traversal performance especially for increasingly incoherent rays.

CCS CONCEPTS

•**Computing methodologies** →**Ray tracing;** *Shared memory algorithms; Vector / streaming algorithms;*

KEYWORDS

Ray tracing, GPU, SIMD

ACM Reference format:
Alexander Lier, Marc Stamminger, and Kai Selgrad. 2018. CPU-Style SIMD Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics, Vancouver, Canada, August 10–12, 2018 (HPG '18),* 4 pages.
DOI: 10.1145/3231578.3231583

1 INTRODUCTION

In this paper we propose a simple extension of state-of-the-art GPU ray traversal [Aila et al. 2012] that requires only minor changes to the underlying data structures and shows significantly improved performance for incoherent rays.

Ray traversal performance using acceleration structures such as BVHs and kd-trees is an excessively researched field. Yet there is still a consistent stream of improvements by closely mapping ray traversal to the underlying hardware [Aila et al. 2012; Barringer and Akenine-Möller 2014; Benthin et al. 2015; Gunther et al. 2007; Guthe 2014; Wald et al. 2014] and also by employing efficient compression [Ylitie et al. 2017]. The method we propose in this paper is classical in that it tackles raw traversal speed and does not consider hierarchy generation or compression.

The main proposition of our method is to use the GPU's warps in a more traditional SIMD-scheme [Dammertz et al. 2008; Ernst and Greiner 2008] by spreading BVH traversal across multiple lanes (spanning virtual sub-warp SIMD-registers).

HPG '18, Vancouver, Canada

978-1-4503-5896-5/18/08…$15.00
DOI: 10.1145/3231578.3231583

2 RELATED WORK

The work of Aila et al. [2009; 2012] is the de-facto reference in GPU ray traversal. Many successive works rely on their findings and improve upon them by adding features or hardware optimizations. For example, Guthe [2014] incorporated a 4-wide BVH and improved incoherent performance by up to 20% (on the Kepler and the Fermi architecture), while Ylitie et al. [2017] presented a compressed 8-wide BVH that improves incoherent performance by up to a factor of 3.3 (on the Maxwell and Pascal architecture).

Wide BVHs. Ray traversal using wide BVHs is common with CPU SIMD ray traversal [Christensen et al. 2006; Dammertz et al. 2008; Ernst and Greiner 2008]. Here, multiple bounding volumes are tested simultaneously with single rays without the need for using ray packets [Benthin et al. 2007; Gunther et al. 2007].

A 4-wide BVH in combination with a ray-direction ordered traversal utilizing SIMD was proposed by Dammertz et al. [2008] as well as by Ernst and Greiner [2008]. Ray-direction ordered processing is a heuristic that does not necessarily process the scene's bounding volumes in a strict front-to-back order, to save on sorting time. In contrast, Wald et. al. [2008] endorse a strict front-to-back order for their SIMD traversal of 16-wide BVHs, which requires sorting of intersections during traversal.

While sorting is trivial for binary BVHs, sorting networks [Batcher 1968] can be applied for higher branching factors (on the CPU this is used, e.g., in Embree [Wald et al. 2014]). On the GPU sorting networks can be implemented very efficiently with Cuda's *shuffle* instructions [Demouth 2013].

Wide BVHs on GPUs. While all these approaches are tailored for optimal SIMD utilization with single rays on the CPU, none of them are specifically targeted at, nor optimized for GPUs. Aila and Laine [2009] mentioned a GPU adaptation, but dismissed further analysis in their paper due to generally lacking performance. However, in subsequent work [Aila and Karras 2010] they stated that such a direction might still be promising, but did not follow it. Consequently, BVHs are commonly processed on the GPU on individual lanes, regardless of their branching factor [Guthe 2014; Ylitie et al. 2017]. To the best of our knowledge, our presentation is the first one to follow through on this direction. However, Binder and Keller [2015] applied a related but distinct approach that handles individual world-space components with separate threads.

BVH Construction. Starting with an efficient binary BVH [Stich et al. 2009], a similarly efficient wide BVH can be constructed efficiently by pulling up individual nodes to create a BVH of a specific width [Wald et al. 2008]. Similarly, fast construction methods for binary BVHs [Karras 2012; Lauterbach et al. 2009; Selgrad et al. 2015] can be turned into efficient methods for wide BVHs in the same way.

3 SIMD RAY TRAVERSAL ON THE GPU

The foundation of our approach is teaming multiple lanes of a warp and letting them traverse the BVH together for one single ray. This concept mimics a regular SIMD-based BVH traversal known from methods utilizing SSE and AVX extension on the CPU [Dammertz et al. 2008; Ernst and Greiner 2008; Wald et al. 2014]. But in contrast to CPUs, switching between vector instruction (e.g. parallel intersection tests) and scalar instruction (e.g. stack management) is not easily (or even efficiently) possible on GPUs. In our case, some operations (e.g. loading, storing, and stack management) have to be handled individually and at times redundantly on each lane. Therefore, we supply each lane with its own copy of the ray data, nearest hit information, and stack pointer. However, the stack itself resides in shared memory and thus is not redundant.

BVH Construction. We start the construction of our acceleration structure with a regular binary BVH. There are no limits to the construction of this BVH, except that we strive for the number of leaf primitives to match the width of our target BVH, i.e. 2, 4, or 8 elements, similarly to CPU implementations [Ernst and Greiner 2008]. In our case, since we added our method to the code published by Aila et al. [2012], we compute high-quality SBVH [Stich et al. 2009]. For the construction of our 4-wide BVHs, we apply the method described by Guthe [2014], which aims at reducing the overall amount of inner nodes by eliminating specific leaf nodes first. For our 8-wide BVHs, we first pull up child nodes with the largest surface area without further adjustments [Wald et al. 2008]. Both methods eliminate nodes by integrating them into their parents until the respective parent node is saturated (i.e. fully populated). This process is repeated on every remaining child node of such a saturated parent until each BVH level has been processed.

Node Layout. For all our BVHs, we slightly adjust the node layout (from Aila et al. [2012]) in order to improve coalesced memory accesses. The original implementation loads two bounding boxes and two child indices at once. Each thread individually fetches four *Vec4* elements and both child indices are stored in the last *Vec4* element:

```
int i = node_index * 4;
nodes[i+0] = vec4(box1.min.x, box1.max.x, box1.min.y, box1.max.y);
nodes[i+1] = vec4(box2.min.x, box2.max.x, box2.min.y, box2.max.y);
nodes[i+2] = vec4(box1.min.z, box1.max.z, box2.min.z, box2.max.z);
nodes[i+3] = vec4(child1.idx, child2.idx, 0         , 0         );
```

In our version, nodes are individually loaded in parts by multiple adjacent threads. Therefore, it is beneficial to separate individual bounding volumes and child indices. In case of a binary BVH, the following layout allows coalesced memory access to particular *Vec4* elements for adjacent threads by applying an offset of 1:

```
int i = node_index * 4;
nodes[i+0] = vec4(box1.min.x, box1.max.x, box1.min.y, box1.max.y);
nodes[i+1] = vec4(box2.min.x, box2.max.x, box2.min.y, box2.max.y);
nodes[i+2] = vec4(box1.min.z, box1.max.z, child1.idx, 0         );
nodes[i+3] = vec4(box2.min.z, box2.max.z, child2.idx, 0         );
```

A corresponding layout is used for our 4- and 8-wide BVHs. The first half of a node always consists of the x and y components of its bounding volumes, followed by the second half containing their corresponding z components and child indices.

Thread Organization. We apply a block width of 32 according to the warp size of the Pascal architecture. The block height is limited to 2 in order to maximize the amount of shared memory available exclusively for each group of threads and our kernel launch configuration aims to populate each streaming multiprocessor with 64 active warps.

Threads are organized in groups (corresponding to CPU SIMD registers) of 2, 4, or 8 elements, which is applied manually on top of the block configuration. The group size depends on the width of the processed BVH, while each thread is aware of its group and its offset within its group. The thread group is trivially computed by dividing the lane index by the current group size (the local thread offset within the group is similarly obtained):

```
uint lane_group  = lane / 4; // for groups of 4 threads
uint lane_offset = lane % 4; // for groups of 4 threads
```

In addition, each group has its own group mask, where the corresponding number of group members regarding their position within the warp is set to 1. Again, the mask is trivially computed with the lane index (e.g for groups of 4):

```
uint group_mask = 0x0000000f << (lane & 0xfffffffc);
```

Ray Management. The presented traversal scheme is built upon *Persistent Threads* [Aila and Laine 2009]. Our ray pool implementation is similar to the original method's, which uses a global counter to track processed rays, but in contrast to the original, we do not reload rays dynamically if the warp utilization drops below a threshold. We observed a general reduction in performance when we utilize dynamic reloading, and therefore decided against it. In addition, omitting dynamic fetch removes the need to identify the number of terminated threads and enables fetching a constant number of rays in each round.

```
if (lane == 0)
   ray_idx = atomicAdd(g_warpCounter, num_groups);
ray_idx = __shfl(ray_idx, 0) + my_group;
```

As depicted, lane 0 of each warp is utilized for incrementing the global ray counter by the number of thread groups. Following the atomic increment, the initial ray index is distributed to all threads, all of which then apply an offset corresponding with their group. Afterwards, all threads of the same group load the same ray data during traversal initialization. Loading rays only once and distributing them among threads was generally slower than duplicated loading.

Traversal. The traversal can be classified as *if-if*-based [Aila and Laine 2009]. In each iteration, a node index is fetched from the stack. Based on that index, either a box-intersection or a triangle-hit test is performed. We do not incorporate a *while-while* setup, since it resulted in slower traversal speed than the straight-forward approach.

The traversal stack resides entirely in shared memory and gets initialized by lane 0 with the root node prior traversal. During traversal, each thread of a group pops the same index from the stack. Yet adjacent threads load adjacent node data from global memory by applying their relative position within the group as an offset, where each node-data block represents a single bounding box and an index to a child node. This access pattern heavily depends on the node layout described ealier:

```
int curr_idx = stack[lane_group][stack_pointer--];
vec4 xy = nodes[curr_idx     + lane_offset];
vec4 zi = nodes[curr_idx + 2 + lane_offset];
```

Triangle intersection is handled similarly by each thread applying an offset and loading one single triangle from global memory. Consequently, each thread tests only one bounding box or one single triangle for intersections in each iteration.

Hit Detection. The traversal of a BVH with grouped threads requires the distribution of specific information among group members. An essential part is detecting if all group members missed their nodes. If that is the case, processing the current node can be aborted and the traversal can continue with the next iteration.

We apply the *ballot* voting function to distribute hit information within a warp and *popc* to count the number of hits.

```
bool hit = intersect_box(xy, zi, dist);
unsigned int warp_vote = __ballot(hit);
unsigned int group_hits = __popc(warp_vote & group_mask);
if (group_hits == 0) continue;
stack_pointer += group_hits;
```

By masking the global warp-wide vote with the thread's group mask, we can count the valid hit occurrences in each thread and properly increment the thread's stack pointer.

Intersection Sorting. Implementing a strict front-to-back traversal requires sorting of intersected nodes by their hit-distances. We adapted a bitonic sorting algorithm tailored for Cuda [Demouth 2013] to sort indices and distances:

```
void swap(float& dist, int& index, uint mask, uint dir){
  float shfl_dist  = __shfl_xor(dist,  mask);
  int   shfl_index = __shfl_xor(index, mask);
  bool  swp = dist != shfl_dist && dist > shfl_dist == dir;
  index = swp ? index : shfl_index;
  dist  = swp ? dist  : shfl_dist;
}
```

This allows us to sort child-indices of, e.g., a 4-wide group by applying only three swap operations:

```
int child_index = zi.z // index of nodes's child
float dist = FLT_MAX;
bool hit = intersect_box(xy, zi, dist);
dist = hit ? dist : FLT_MAX;

swap(dist, child_index, bfe(lane, 1) ^ bfe(lane, 0));
swap(dist, child_index,               bfe(lane, 1));
swap(dist, child_index,               bfe(lane, 0));
// bfe: bit field extract
```

In case of an invalid hit, the distance value is set to a sentinel value to ensure a valid ordering and to distinguish valid hits from invalid ones after the sorting step. After this, each thread contains an index to be used for one of the next iterations, in ascending order. Indices of missed nodes are kept by threads with offsets higher than the number of valid hits. Since the stack pointer was already increased by the number of hits, the stack entries for the next iteration can be stored directly in the proper position by applying the threads' local offsets:

```
if (dist < FLT_MAX)
  stack[lane_group][stack_pointer - lane_offset] = child_index;
```

The last important aspect is identifying the closest triangle intersection. We apply an adapted reduction [Demouth 2013] for distributing triangle hit information within a group.

```
#pragma unroll
for (int mask = 2; mask > 0; mask >>=1) {
  float shfl_dist = __shfl_xor(dist, mask);
  int   shfl_addr = __shfl_xor(tri_addr, mask);
  tri_addr = shfl_dist < dist ? shfl_addr : tri_addr;
  dist     = fminf(dist, shfl_dist);
}
```

The final ray traversal results are stored by the first member of each thread group at the end of the traversal.

4 EVALUATION

We compare our method with Aila et al.'s [2012] original kernels for the Kepler architecture and Guthe's [Guthe 2014] adaption for 4-wide BVHs. In order to get comparable results, we extended the publicly available source code[1] with our method and additionally used Guthe's implementation for reference. Our implementation is publicly available online[2]. All our measurements, shown in Image 1, were taken on an nVidia GTX 1070 graphics card and all kernel code was compiled utilizing Cuda 9.1. A possible disparity between our measurements and the results found in the corresponding publications, may originate from the different hardware and software configuration we use and the fact, that we do not apply sorting on diffuse rays in any of our measurements and compared methods.

Ray Types and Models. We considered primary and increasingly incoherent diffuse rays in our evaluation. Primary rays were generated and measured for multiple view point throughout the benchmark. We measured rays up to the eighth bounce and show results for the first bounce (diffuse/1) and eighth bounce (diffuse/8). All considered models can be seen in the top area of Figure 1.

Coherent Rays. As it can be seen in Figure 1 the presented method is generally slower for primary rays. In worst case scenarios, 2-wide and 4-wide BVH traversal is around 30% slower than the binary reference. Yet, for primary rays in average, our 2-wide kernel performs similarly efficient as the 2-wide reference, wile the 4-wide kernel is about 20% slower. Our 8-wide method is up to 50% slower in specific setups and about 40% slower for primary rays on average. In comparison to the 4-wide reference [Guthe 2014], our approach is generally slower for primary rays.

Incoherent Rays. Considering diffuse rays, our approach becomes increasingly more performant. On average, our 2-wide and 4-wide versions outperform both references by 10% to 20% and up to 100% in individual cases. Our 8-wide version is generally inferior to both references considering diffuse rays.

Our method performs best with highly incoherent rays, as illustrated by the times for the eighth bounce. The 4-wide variant surpasses the 4-wide reference [Guthe 2014] by more than 65% on average and is more than 2.7 times faster in individual cases. Our 2- and 8-wide variants exhibit a solid increase of about 35% and 45% on average.

Ultimately, the presented method offers improved performance for diffuse rays and considerable efficiency gains for strongly incoherent rays.

[1] https://code.google.com/archive/p/understanding-the-efficiency-of-ray-traversal-on-gpus/
[2] https://github.com/lispbub/simd-ray-traversal

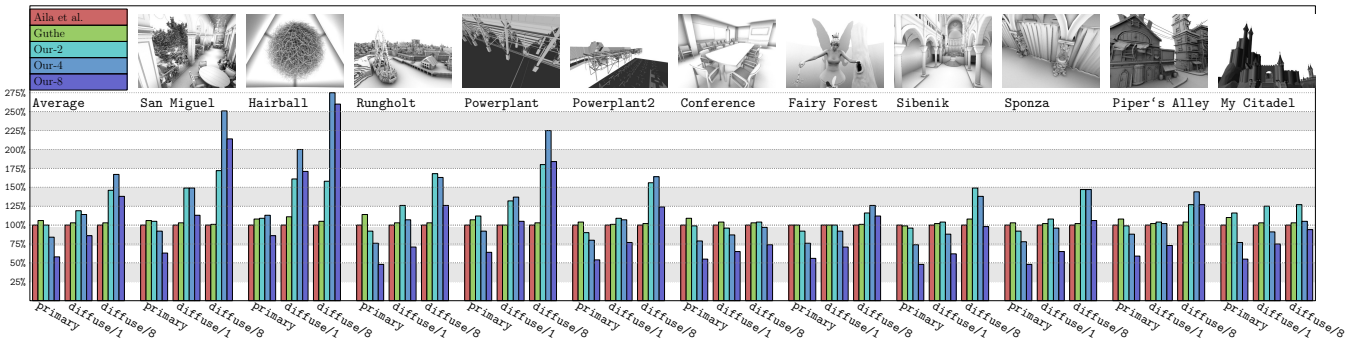

Figure 1: Graphs showing the relative performance of our 2-, 4-, 8-wide variants, and Guthe's approach [Guthe 2014] in relation to the 2-wide reference [Aila et al. 2012] for primary and diffuse (first bounce and eighth bounce) rays. Individual blocks correspond to the models shown above each group of primary and diffuse rays. In general, our methods perform very well for incoherent rays for rather big and increasingly complex scenes on the expense of decreased efficiency for primary rays. A full tabulation can be found in the supplemental material.

Further Aspects and Limitations. As described in the previous section, our approach uses persistent threads. Still, we do not dynamically reload rays nor do we use speculative traversal as both optimizations reduced the general performance of our approach.

Our kernel code demands only 32 registers and allows up to 100% occupancy. However, profiling indicates, that the presented method is still heavily latency bound.

We also tested 16-wide and 32-wide BVHs and their traversal, but similarly to the 8-wide method, performance decreases with wider BVHs and larger thread groups. Consequently, we did not consider wider trees in our final evaluation.

Our evaluation was primarily focused on the Pascal architecture. Nevertheless, isolated tests on the Maxwell architecture indicated that the method is not suited for older GPUs. This might explain, why Aila and Laine [Aila and Laine 2009] reported lacking performance and this method has not been evaluated more thoroughly until now.

5 CONCLUSION

We have presented and evaluated an alternative concept for ray traversal on GPUs that mimics a method conventionally applied on CPUs. Despite its origin, the general idea maps nicely to GPUs and was surprisingly easy to implement and to integrate into existing frameworks. In addition, its simplicity eases implementing BVHs of various widths, while this flexibility does not result in diminished performance. Quite the opposite, the approach exhibits heavily improved performance for incoherent ray traversal.

Even though significant improvements for incoherent rays have been demonstrated, it should be noted that ray traversal performance for highly coherent rays is lower than with the standard approach. Also, efficiency gains become apparent primarily on the current Pascal architecture. Investigating the behaviour of the presented method on newer architectures, e.g. Volta, is a very tempting topic for further analysis. Alternatively, applying additional compression might also reveal beneficial synergetic effects.

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proc. High Performance Graphics (HPG '10)*. 113–122.

Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High Performance Graphics 2009 (HPG '09)*. 145–149.

Timo Aila, Samuli Laine, and Tero Karras. 2012. *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum.* Technical Report NVR-2012-02. NVIDIA Corporation.

Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic Ray Stream Traversal. *ACM Trans. Graph.* 33, 4 (July 2014), 151:1–151:9.

Kenneth E. Batcher. 1968. Sorting Networks and their Applications. In *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32. 307–314.

Carsten Benthin, Solomon Boulos, Dylan Lacewell, and Ingo Wald. 2007. Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces. *SCI Institute, University of Utah, Technical Report UUSCI-2007-011* (2007).

Carsten Benthin, Sven Woop, Matthias Nießner, Kai Selgard, and Ingo Wald. 2015. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proc. High-Performance Graphics (HPG '15)*. 5–12.

Nikolaus Binder and Alexander Keller. 2015. Stackless Ray Tracing of Patches from Feature-adaptive Subdivision on GPUs. In *ACM SIGGRAPH 2015 Talks (SIGGRAPH '15)*. 22:1–22:1.

Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*. 1–6.

Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. *Computer Graphics Forum* 27, 4 (2008), 1225–1233.

Julien Demouth. 2013. Keplers's SHUFFLE (SHFL): Tips and Tricks *(GTC '13)*.

Manfred Ernst and Günther Greiner. 2008. Multi Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*. 35–40.

Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. 2007. Real-time Ray Tracing on GPU with BVH-based Packet Traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing (RT '07)*. 113–118.

Michael Guthe. 2014. Latency Considerations of Depth-first GPU Ray Tracing. In *Eurographics 2014 (Short Papers)*, Eric Galin and Michael Wand (Eds.).

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proc. ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics (EGGH-HPG'12)*. 33–37.

Christian Lauterbach, Michael Garland, Shubho Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

Kai Selgrad, Jonas Müller, and Marc Stamminger. 2015. Faster Ray-Traced Shadows for Hybrid Rendering of Fully Dynamic Scenes by Pre-BVH Culling. In *Proc. Smart Tools and Apps for Graphics – Eurographics Italian Chapter Conference (STAG '15)*.

Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proc. High-Performance Graphics 2009 (HPG '09)*.

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting rid of packets – Efficient SIMD single-ray traversal using multi-branching BVHs. In *2008 IEEE Symposium on Interactive Ray Tracing*. 49–57.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 33, 4 (July 2014), 143:1–143:8.

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *Proc. High Performance Graphics (HPG '17)*. 4:1–4:13.