# Performance Considerations When Using a Dedicated Ray Traversal Engine

Tomáš Davidovič
Saarland University and DFKI
Campus E1 1
DE, Saarbruecken
davidovic@cs.uni-saarland.de

Lukáš Maršálek
Saarland University and IVCI
Campus E1 1
DE, Saarbruecken
marsalek@cs.uni-saarland.de

Philipp Slusallek
Saarland University and DFKI
Campus E1 1
DE, Saarbruecken
slusallek@cs.uni-saarland.de

**ABSTRACT**

In the recent years we have witnessed massive boost to hardware graphics accelerators (graphics cards), not only in the raw performance, but also in their programmability, introducing the concept of GPGPU. However, despite this, the current architectures still favor feed-forward algorithms over recursive ones. While shading is, in this sense, a feed-forward algorithm, ray tracing, and specifically ray traversal, is a recursive rather than feed-forward algorithm. Adding a dedicated hardware ray traversal engine should therefore prove to be an interesting option. Also, with dedicated hardware, we can perform many optimizations on arithmetic units due to their fixed interaction. This can reduce the area well below a simple sum of areas of the individual units. In this paper we offer for consideration analysis of memory requirements for combination of a dedicated hardware ray traversal and intersection engine with highly parallel general purpose processor used for shading. We show results and requirements of such a combination on scenes of moderate complexity, with regard to speed, bandwidth and latency.

**Keywords:** ray tracing, hardware, GPU, bandwidth

# 1 INTRODUCTION

CPU ray tracing has received tremendous speed ups in the past decade, with works on acceleration structures, traversal algorithms, and packet (or frustum) traversal [7, 17, 16, 12, 6]. While these advancements brought ray tracing into the real-time area [9], the speed is still generally not competitive to rasterization.

One of the chief problems causing this gap is the dedicated rasterization hardware, which is something ray tracing lacks. The modern graphics processing units (GPU) consist from a relatively small number of fixed function raster engines (four in NVIDIA's Fermi architecture) and a large number of general purpose processing elements. Typically the general purpose part of GPU prepares triangles (vertex and geometry shaders) for raster engines, which dice triangles into a stream of fragments that are fed back into the general purpose part that performs fragment shading on them.

There are many papers [10, 6, 2] on GPU ray tracing, as well as some deliverable products [8]. All such approaches use solely the general purpose part, and most of them focus on overcoming limitations of their contemporary architectures. In [15] Seiler et al. present ray tracing on Intel's Larrabee, also utilizing only general purpose elements of the chip.

The opposite approach, a stand-alone ray tracing hardware, has also been explored [14, 18, 19, 13, 11]. While such solutions have potential to deliver the best performance, the development in rasterization hardware shows necessity of general purpose shading capabilities.

Surprisingly little research has been done on the middle ground, replacing the GPU raster engines with another kind of fragment generator. The only significant work on this has been done by Caustic Graphics [3], who proposed a separate ray tracing accelerator board, that can utilize present GPU for shading.

A recent paper by Aila and Kerras [1] focuses on bandwidth issues of complex scenes. The paper assumes a separate ray tracing engine, but presents results for a whole NVIDIA Fermi GPU used as such unit. However, it leaves open possibility of different implementations, including dedicated hardware.

In this paper we present a solution, which proposes to place a small dedicated hardware ray traversal engine (RTE) either directly on the GPU die, or as a coprocessor on the graphics card.

# 2 RAY TRAVERSAL ENGINE

Our ray traversal engine (RTE) implementation is based on the well-documented FPGA implementation of the DRPU by Woop [18]. We will first give a brief overview of the basic blocks used, then discuss frequency, area scaling, and the connected design decisions.

## 2.1 Design blocks

The RTE (Figure 1) is a heavily pipelined unit with fine grain multithreading. It uses B-KD trees [20] for its acceleration structure. Our B-KD tree implementation offers a two-level hierarchy, where each leaf of the top tree can contain either a triangle or a transformation matrix and pointer to a subtree.

The basic computation unit is a thread, which processes four rays at once. Each pipeline stage can accommodate a different thread, and there is no overhead in switching threads. The whole RTE can process up to 64 threads (256 rays) at once. While there is no strict requirement for coherence of rays within the same thread, it can bring performance boost. Also, coherence between all processed rays improves cache hit rates.

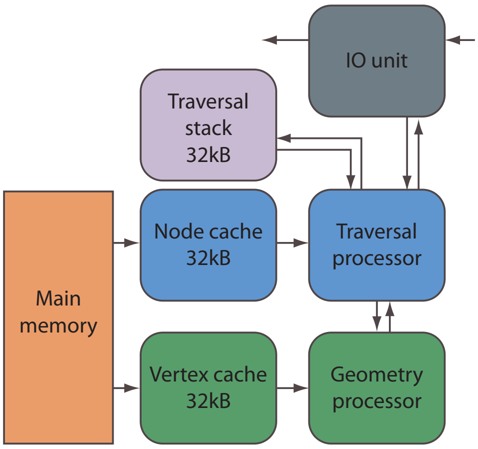


Figure 1: **RTE block diagram**. The RTE core consists of traversal and geometry units. Each unit is connected to main memory via 32 kB cache. Traversal unit has an additional 32 kB memory for traversal stack and connection to the IO interface to return results and receive new queries.

The IO unit strongly depends on the particular implementation details of integrating RTE with the rest of the system. We will therefore not elaborate on its implementation, but we can imagine it as a kind of memory controller that handles ray and result queues in the main memory, akin to what is assumed in [1]. The basic functionality is to fetch rays for idle threads and store results from finished ones.

The actual RTE core consists of two units: traversal and geometry. Each is connected to the main memory via a 4-way associative 32 kB read-only cache. A single larger cache has been considered, but there would be either high contention for its single port, or it would have to be two port cache, resulting in higher complexity. Also, as each unit requires different data, there is no redundancy in fetches from main memory that would lower the efficiency.

Another two parts, not shown in the Figure 1 are per-thread stack memory and ray state buffer. The per-thread stack is accessed with up to 1 read and 1 write each cycle, where the read occurs when a thread is submitted to traversal unit, and the write occurs when the result of traversal stage is that both children should be intersected. Each item in stack consists of a node address (4 B) and near and far values for each ray (32 B), for the total of 36 B per item. For convenience we consider maximum scene depth of 32. Making the internal stacks shorter and spilling overflow into main memory, as described in [1], is also an option.

Ray state buffer is per-ray storage with current closest hit distance, ID of the closest triangle and barycentric coordinates.

For each thread, the traversal unit fetches the required node. If the node is a leaf, it is sent to the geometry unit. For inner nodes it intersects the child nodes and, based on the result, it either pops a new node from stack, traverses the single hit child, or traverses the closer child and pushes the farther child onto the stack. If the stack is empty it informs the IO unit that the ray has finished.

The geometry unit computes both the intersection with triangles and the transformation of rays for traversing the bottom tree when a two-level hierarchy is used. This is possible because both operations require very similar functional units with a minimal reconfiguration. When this unit receives a leaf node, it first fetches three vertices (for intersection) or matrix rows (for transformation). The thread waits until all three memory fetches are finished before further execution. Each ray of the thread occupies 2 consecutive pipeline stages, therefore it takes 8 cycles to process the whole thread.

The unit is, for some of the measurements, augmented with a 1 MB 4-way associative L2 read-only cache. This cache has latency of 100 cycles, and assumes 512 bit wide access to memory (fetches 64 B at once). When L2 cache is not present, the L1 caches are connected directly to the memory and assume 128 bit wide access. The main memory is assumed to have latency 600 cycles.

## 2.2 RTE synthesis

We first synthesized all the major arithmetic sections of both units as well as the cache logic using proprietary IBM 90 nm process [4]. All synthesized parts were able to run at frequencies over 2 GHz. This was done without any special fusion of dependent arithmetic units. Also, considering the already quite high latency of both traversal (14 cycles) and geometry (36 cycles) units, more stages could be added to further stabilize the frequency, without a significant impact on the overall performance. While both optimizations can lead to higher frequency than reported, we opted against their implementation as the preliminary results are more than satisfactory.

We have not explicitly synthesizes all the required memories, instead we borrowed statistics from similar memories from Cell/B.E.$^{TM}$ processor [5]. This processor has 32 kB L1 cache, running at 3.2 GHz, as well as 1-read 1-write port local store memory running at the same frequency. As these two types of memory fulfill all requirements we have on our memory blocks, we conclude that memory frequency will not be an issue.

The area of RTE is not of the prime concern for our results, so we present only rough upper bound on RTE area. We used a range of area estimation techniques to confirm that RTE comfortably fits into the area of less than 15 mm$^2$. Further reduction of size is possible by aforementioned fusion of arithmetic units.

# 3 SIMULATOR ARCHITECTURE

We have performed two distinct simulations. The first simulation was on the actual low-level VHDL code, that is used as base for our synthesis results. While this confirms that the design is correct and the synthesized frequencies valid, the simulation itself is very slow.

To solve this we also designed a cycle-accurate SystemC model of RTE. SystemC is a C++ library that allows using many high-level language constructs not available in VHDL, while at the same time allowing to simulate exactly the same timings, to the cycle levels. We integrated this RTE model as a part of our ray tracing framework, essentially replacing its standard traversal and intersection routine.

Normally our framework uses the following workflow. First, a primary ray is generated from camera. It is then intersected with the scene and the closest geometry is found. If there is any hit, integrator is invoked that, based on its type, queries material for its BRDF, scene for the lights, performs shading and possibly shoots other rays. Once the primary ray is finished and the final color is computed, another primary ray is generated, until the whole image has been rendered.

While this sequential process works very well for CPU rendering with low amount of parallelism (4-16 threads at once), it is ill-suited for massively parallel hardware acceleration. We have therefore modified the algorithm as follows. First, all primary rays are generated and put into input queue of the RTE. The RTE emulation engine is then started, and after each cycle its output queue is checked. If there is a ray in the output queue, it is passed to the integrator that processes the ray. This can, and generally does, generate other rays, that are in turn fed back into the input queue. Where there are no more rays in the input queue and no active rays in the RTE, the frame has finished. Should generating all primary rays occupy too much memory at once, it is also possible to split the image into several blocks or batches, based on the memory configuration.

## 3.1 Shader models

Unfortunately, this system does not lend itself easily to the standard recursive shaders, nor to any kind of shader where the integrator needs result of a traced ray. There are two possible solutions to the problem. First, adopted by [3], is to employ tail recursion. Here the integrator generates all required rays in a fire-and-forget manner, attaching to them all the necessary information. Shadow rays would, for example, work in such a way, that all lights are counted as contributing. Shadow rays are then generated with a flag marking them as shadow, maximum length and weight for the particular light. When they do not hit anything, no integrator code is invoked and the light contribution stays added. If they hit, the integrator simply accumulates the negative weight to the pixel, effectively subtracting the light contribution. In this way, the original integrator does not have to wait for shadow ray results, nor do we need special *no hit* integrator to add light contribution when shadow ray does not hit anything.

This approach has a drawback of spawning large and poorly controllable number of rays. The other approach is to use continuations. Here the integrator is effectively split at each call for *traceRay* and its state is stored. Once the ray tracing has been finished, the state is fetched from a global storage and the integrator continues. This can be easily implemented by a finite state machine, but requires stack in the case ray bifurcation is allowed. One of the advantages is that we do not need to sum over all rays contributing to a single pixel, as all the computations concerning one pixel are very self-contained. Disadvantages include larger per-ray storage and higher sensitivity to latency.

## 3.2 Acceleration structure partitioning

To increase cache coherence, Aila and Kerras [1] introduce concept of partitioning the acceleration structure into multiple treelets (subtrees), each of which fits into the cache. As we consider this approach very relevant for our results, we adapted it for B-KD-trees and included it in our measurements.

The basic principle is that the whole structure is split into many small subtrees, called treelets. Each of the treelets has its own ray queue and when a ray crosses a treelet boundary, its traversal is stopped and it is put into corresponding treelet's input queue. While the original paper introduced several methods for scheduling treelets to ray traversal engines, we use only the simplest one called *lazy scheduler*. It simply takes the treelet with largest queue and processes it until the queue is empty. Then it switches to another treelet, with the currently largest queue. The more complex methods are meant to balance situation where there are multiple ray traversal engines, which is something we do not currently consider for our scenario.

# 4 RESULTS

We tested on three scenes of moderate complexity, the **Conference** (283k triangles), **Fairy forest** (174k triangles), and **Kitchen** (253k triangles). We test both

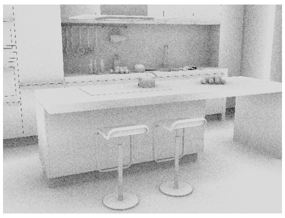
| | Conference | | Fairy forest | | Kitchen | |
|---|---|---|---|---|---|---|
| | | | | |  | |
| Triangles: | 282759 | | 174117 | | 253433 | |
| Res: | $512 \times 512$ | | $512 \times 512$ | | $600 \times 450$ | |
| **Continuations shaders** | | | | | | |
| L2 | off | on | off | on | off | on |
| L1 hit rate [%] | 68/53 | 69/55 | 55/44 | 56/45 | 83/77 | 89/85 |
| L1 bandwidth [GB s$^{-1}$] | 6.7/4.7 | 15.7/11.0 | 5.0/3.4 | 11.7/7.9 | 11.2/7.6 | 16.7/11.3 |
| L2 hit rate [%] | - | 95/84 | - | 88/73 | - | 90/80 |
| L2 bandwidth [GB s$^{-1}$] | - | 4.8/4.9 | - | 5.2/4.3 | - | 1.8/1.7 |
| Ray bandwidth [GB s$^{-1}$] | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 |
| Mem bandwidth [GB s$^{-1}$] | 4.4 | 4.2 | 4.3 | 7.5 | 3.7 | 2.1 |
| Latency [cycles] | 6.0 M | 2.7 M | 12.0 M | 5.4 M | 3.3 M | 2.3 M |
| Throughput [MRays/s] | 47.8 | 112.4 | 20.8 | 48.1 | 66.6 | **100** |
| **Tail recursive shaders** | | | | | | |
| L2 | off | on | off | on | off | on |
| L1 hit rate [%] | 76/64 | 78/67 | 63/53 | 64/55 | 87/82 | 85/80 |
| L1 bandwidth [GB s$^{-1}$] | 8.3/5.8 | 16.3/11.2 | 6.0/4.1 | 12.5/8.4 | 12.6/8.5 | 16.8/11.4 |
| L2 hit rate [%] | - | 93/79 | - | 86/71 | - | 93/86 |
| L2 bandwidth [GB s$^{-1}$] | - | 3.6/3.7 | - | 4.5/3.8 | - | 2.6/2.3 |
| Ray bandwidth [GB s$^{-1}$] | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 | 0.14 |
| Mem bandwidth [GB s$^{-1}$] | 4.4 | 4.1 | 4.1 | 6.2 | 3.3 | 2.1 |
| Latency [cycles] | 4.9 M | 2.6 M | 10.0 M | 5.0 M | 3.0 M | 2.3 M |
| Throughput [MRays/s] | 61.1 | **117.4** | 25.0 | **52.3** | 75.4 | 99.2 |

Table 1: **Results without treelets.** We measure all three scenes with both tail recursive and continuation shaders and both with and without 1 MB L2 cache. The reported results are L1 cache hit rate, required L1 bandwidth in GB s$^{-1}$, the same for L2 cache (if applicable), ray traffic bandwidth in GB s$^{-1}$, total required memory bandwidth, both from cache and rays, latency in cycles (Latency) and throughput in million rays per second. The L1 and L2 cache results are given in format vertices/nodes.

shader approaches, using 16 rays per pixel ambient occlusion. We focus on the cache hit rates, throughput (rays per second), ray latency (important for continuation shaders), and the bandwidth requirements of our unit, measured both with and without an L2 cache.

The reason why we provide not only cache hit rates, but also their bandwidth is that one should not automatically assume that higher hit rates are more desirable. Should we devise the perfect ray tracing algorithm, that only ever needs each node and triangle exactly once, our caches would have hit rate 0%, but the cache bandwidth would be significantly lower. Moreover, looking at the bandwidth is still not sufficient, because the most obvious way to lower bandwidth while keeping the same hit rates is to reduce the overall throughput of the engine, which is not at all desirable. We therefore have to look at the ray throughput, cache hit rates and bandwidth together and draw conclusions only from the combination of the three, as we will see below.

At the beginning in Section 4.1 we provide results for RTE without the use of treelets, follow up with Section 4.2 commenting results using treelets and close with Section 4.3 on using BVH instead of B-KD-tree.

## 4.1 Standard implementation

Table 1, providing results for the standard RTE implementation, offers several insights. First and most important is, that the L2 cache helps in all the scenes, giving us speed up factor 1.3-2.3×. The reason for this is somewhat obvious, the L2 cache provides large portion of the scene with lower latency (100 cycles) than the main memory (600 cycles). The L2 cache also exhibits relatively high hit rate, above 85 % for nodes and over 70 % for vertices. It is important to note, that while the L2 hit rate is higher for continuations shaders, this is actually caused by the lower L1 cache hit rate, which causes more requests to L2 cache, increasing the bandwidth as noted at the beginning.

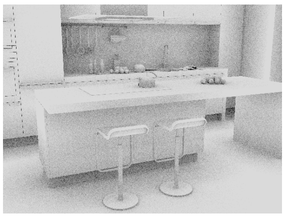
| | Conference | | Fairy forest | | Kitchen | |
|---|---|---|---|---|---|---|
| | | | | |  | |
| Triangles: | 282759 | | 174117 | | 253433 | |
| Res: | 512 × 512 | | 512 × 512 | | 600 × 450 | |
| **Continuations shaders** | | | | | | |
| L2 | off | on | off | on | off | on |
| L1 hit rate [%] | 83/70 | 86/73 | 79/65 | 81/66 | 89/82 | 91/85 |
| L1 bandwidth [$\mathrm{GB\,s^{-1}}$] | 9.3/6.5 | 14.0/9.8 | 8.0/5.4 | 12.1/8.1 | 11.7/7.9 | 15.3/10.3 |
| L2 hit rate [%] | - | 79/62 | - | 72/57 | - | 76/66 |
| L2 bandwidth [$\mathrm{GB\,s^{-1}}$] | - | 2.0/2.7 | - | 2.3/2.8 | - | 1.4/1.6 |
| Ray bandwidth [$\mathrm{GB\,s^{-1}}$] | 1.3 | 1.3 | 2.2 | 2.2 | 1.4 | 1.4 |
| Mem bandwidth [$\mathrm{GB\,s^{-1}}$] | 5.5 | 7.1 | 5.8 | 9.5 | 4.2 | 4.8 |
| Latency [cycles] | 2.8 M | 2.2 M | 5.8 M | 4.6 M | 3.6 M | 3.1 M |
| Queue switches [-] | 104718 | 103067 | 196828 | 199421 | 130285 | 129507 |
| Avg. queue size [-] | 87 | 88 | 86 | 85 | 79 | 79 |
| Throughput [MRays/s] | 67.2 | 101.2 | 33.0 | 49.3 | 69.1 | 90.4 |
| **Tail recursive shaders** | | | | | | |
| L2 | off | on | off | on | off | on |
| L1 hit rate [%] | 96.2/92.3 | 97/93 | 95/89 | 96/89 | 97/95 | 98/95 |
| L1 bandwidth [$\mathrm{GB\,s^{-1}}$] | 14.0/9.8 | 15.4/10.7 | 13.9/9.3 | 15.5/10.3 | 15.0/10.2 | 16.1/10.9 |
| L2 hit rate [%] | - | 73/61 | - | 70/64 | - | 70/62 |
| L2 bandwidth [$\mathrm{GB\,s^{-1}}$] | - | 0.4/0.7 | - | 0.6/1.1 | - | 0.4/0.5 |
| Ray bandwidth [$\mathrm{GB\,s^{-1}}$] | 1.2 | 1.2 | 2.0 | 2.0 | 1.3 | 1.3 |
| Mem bandwidth [$\mathrm{GB\,s^{-1}}$] | 2.5 | 2.8 | 3.7 | 4.4 | 2.3 | 2.5 |
| Latency [cycles] | 15.8 M | 15.4 M | 30.0 M | 28.7 M | 24.3 M | 23.7 M |
| Queue switches [-] | 13220 | 13176 | 19118 | 19252 | 15092 | 15255 |
| Avg. queue size [-] | 659 | 661 | 829 | 823 | 653 | 647 |
| Throughput [MRays/s] | 103.6 | **112.8** | 57.8 | **64.4** | 90.2 | **96.7** |

Table 2: **Results with treelets.** We again measure all three scenes with both tail recursive and continuation shaders and both with and without 1MB L2 cache. The reported results are L1 cache hit rate, required L1 bandwidth in $\mathrm{GB\,s^{-1}}$, the same for L2 cache (if applicable), ray traffic bandwidth in $\mathrm{GB\,s^{-1}}$, total required memory bandwidth, both from cache and rays, latency in cycles (Latency) and throughput in million rays per second. Two treelet specific statistics are the number of queue switches and the average size of queue that has been scheduled for processing.

The ray traffic bandwidth, created by reading and writing rays from input and into output queues, is in all the measurements an order of magnitude lower than the total bandwidth to the main memory and thus relatively insignificant. We can also see that with the total memory traffic between 2 and 6 $\mathrm{GB\,s^{-1}}$, we are well beneath the peak performance of current GPU memory systems.

Another very important thing is the ray latency (noted in the table simply as *Latency*). This represents the average number of cycles between receiving a ray into the input queue and writing the result into the output queue. The latency goes from 2 million to 12 million cycles (1-6 ms at 2 GHz) for both tail recursive and continuation shaders. This effectively prohibits any kind of active or passive waiting on the shading side. By active waiting we mean an actual spin loop that checks ray status. By passive waiting we mean not scheduling the thread, akin to when threads are waiting for global memory access on NVIDIA GPUs. We would therefore keep a work queue of rays to be processed and whenever tracing a ray is required, we would store the whole shader state, submit the ray query and fetch a different ray from the work queue. Considering that with ray bifurcation the shader state actually contains a ray stack, the whole process becomes significantly more involved than the tail recursive shaders.

Also, looking at the ray throughput, we can see that the tail recursive shaders lead to almost universally better results than continuation shaders, and never actually perform significantly worse. The improvement always corresponds to increase in L1 cache hit rate and bandwidth, suggesting that the tail recursion gives us noticeably more coherent rays.

So far we have concluded that the overall best choice would be using tail recursion with L2 cache, giving us 50-100 million rays per second. However, assuming that the L2 cache is occupied roughly equally by both nodes and vertices, it can hold up to 25 % of the whole scene, for each of our scenes. The cache therefore effectively lowers demand on ray coherence, but considering larger scenes, this effect would become less pronounced. We would therefore prefer to increase the coherence itself rather than mitigate the impact of incoherence.

## 4.2 Treelet implementation

Towards this goal we implemented the treelet approach introduced by Aila and Kerras [1], as described in Section 3.2. The results are summed up in Table 2.

We present the same statistics as in Table 1, but on top of that provide two statistics that are specific to the treelet mechanism. The first is the number of input queue switches. It represents the how many times was the RTE switched from working on one treelet to another. Obviously, the lower the number the better, as each switch effectively means cache invalidation (whole different treelet is loaded). Corresponding to that is the average queue size, measured when the queue's treelet was switched to active. It represents the number of rays that are processed between the cache invalidations. Here, the larger the number the better.

Looking at the L1 hit rate, bandwidth and the overall performance, we can conclude that the treelets do provide overall improvement over the standard implementation. Because the rays are moved to and from the RTE on each treelet boundary crossing we can see an order of magnitude increase in ray traffic bandwidth. This is offset by the fact that due to the increased coherence between rays, the total bandwidth to the memory (including the ray traffic bandwidth) is actually lower than in the implementation without treelets.

We can see drop of about 10 % in the L2 hit rate, combined with a significant drop in the L2 bandwidth. This is also manifested by much closer ray throughput between the versions with and without L2 cache.

The tail recursive shaders clearly and consistently provide better results than continuation shaders, mainly due to the fact that they have much more rays in flight that can be sorted into treelet queues. As result, there are about 10× less queue switches with queues being on average 10× larger than when using continuations.

The only drawback is significant increase in ray latency (to 15-30 million cycles, i.e. 7.5-15 ms), which, however, is not so important when tail recursion is used.

## 4.3 Using BVH

While using B-KD-tree treelets improved the situation significantly for both **Conference** and **Kitchen** scenes, the **Fairy forest** scene showed unsatisfactory results. The treelets did indeed balance the performance between versions with and without L2 cache, but the absolute performance was still only slightly above half of what we could achieve in the other two scenes.

We suspected that the B-KD-tree might be poor fit for the scene and modified our RTE to handle BVH instead. We modified only the simulation engine itself, without any considerations for the changes in area or frequency.

We implemented two different approaches to the BVH. The first approach we call *Node BVH*, where each node contains its own bounding box and only indices to the children. The traversal then checks whether ray hits a node, and if so always proceeds to both children, determining the first one based on node split plane and ray direction. The second approach we call *Child BVH*, where each node contains bounding boxes of both its children and we only descend to the child the ray intersects.

While there is no principal difference between the approaches, two things have to be considered. First, the *Child BVH* needs to perform 12 ray-plane intersections, while the *Node BVH* needs to perform only 6. This essentially means a factor of 2× in terms of area requirements for ray traversal unit. The other thing to consider is treelet implementation. Should we choose to use *Node BVH*, ray can descend to a child that resides in another treelet only to discover it does not intersect the child, thus generating two unnecessary treelet transitions.

| L2 | off | on |
|---|---|---|
| B-KD-tree [MRays/s] | 57.7 | 64.4 |
| Node BVH [MRays/s] | 50.0 | 54.9 |
| Child BVH [MRays/s] | 65.0 | 73.5 |

Table 3: **Acceleration structures.** We show performance in million rays per second on the Fairy forest scene, using tail recursive shaders, treelets and three different acceleration structures.

The Table 3 shows that the *Node BVH* drawback of unnecessary treelet transitions outweighs any performance gain by using BVH acceleration structure. The *Child BVH* approach offers approximately 12 % speed up, both with and without L2 cache, but further tests showed that a similar speed up is achieved in the other two scenes as well.

Given the fact that going from very light weight B-KD-tree nodes to BVH nodes required for *Child BVH* approach would introduce significant changes in the whole design, we did not pursue this any further.

We conclude that the lower performance in **Fairy forest** is not due to the acceleration structure itself, but rather due to the complex traversal paths of rays we generated. This is also supported by the highest number of queue switches as well as the longest average ray latency among the three scenes.

## 5 CONCLUSION AND FUTURE WORK

We introduced a hardware implementation of a ray traversal engine (RTE), that could act as a fragment generation unit in GPU, in lieu of current rasterization engines. The RTE has been confirmed to run at frequencies above 2 GHz and can fit into area less than 15 mm$^2$, and achieves performance of over 100 million rays per second while keeping bandwidth to the main memory below 5 GB s$^{-1}$.

The unit was tested in two variants, with and without 1 MB L2 cache. While the L2 cache is beneficial for the overall performance by mitigating impact of ray incoherence, we implemented a treelet approach to actually reduce the incoherence.

We tested not only on our basic acceleration structure, B-KD-tree, but also on BVH with two versions of traversal, showing that the B-KD-tree offers only slightly lower performance than the significantly more involved of the traversals and is actually superior to the less involved one.

Two competing shader styles are compared, tail recursive shaders and continuations shaders. The first is found to be almost universally better, as even with 16 rays per pixel the memory consumption and bandwidth are very reasonable, while at the same time it provides a large pool of rays that nicely complements the treelet approach.

In the future, analysis of combination and cooperation of multiple such units would prove to be very useful. While testing on significantly more complex scenes proved to be challenge for our simulator implementation, the preliminary results show that feeding the treelet queues with only one unit is not optimal.

In conclusion, we propose that using tail recursive shading does match the feed forward scheme used in the current GPUs rather well, and that hardware ray traversal engine using tail recursive shaders with treelets would be an interesting, useful, and not very demanding addition to the current GPUs.

## APPENDIX A

The ray life cycle when tail recursion with treelets is used is the most complicated scheme we are using. We will therefore describe it in more detail. Please refer to Figure 2. The numbers mark data paths used in each step and correspond to the step numbers below:

1. Shader stores ray into Root Queue, a queue associated with root treelet.

2. The input output (IO) unit picks the largest queue for processing.

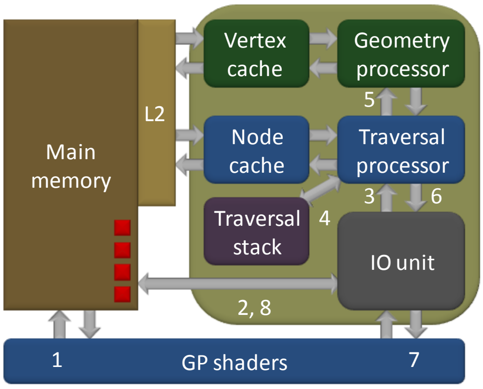


Figure 2: **RTE block diagram**. We use this diagram taken from the presentation to explain life cycle of ray in the tail recursion with treelets configuration.

3. Rays from this queue are sent to traversal processor to traverse ray in the treelet. This is not a block transfer, they are sent only when unit has free processing capacity.

4. During the traversal we use per-ray stack.

5. When ray encounters a leaf, ray triangle test is performed in Geometry processor.

6. When ray encounters a treelet boundary, ray goes back to IO unit.

7. If ray is completely finished, it is sent to shaders and initiates new shader code.

8. Otherwise it is stored in queue corresponding to the treelet it wants to traverse next.

9. When all rays from the processed queue are finished, algorithm goes to step 2. When all queues are empty, algorithm terminates.

## REFERENCES

[1] Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics 2010*, 2010.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[3] Caustic Graphics, Inc. CausticRT platform. http://www.caustic.com/, 2009.

[4] Tomas Davidovic, Lukas Marsalek, Nicolas Maeding, Markus Kaltenbach, Peter-Hans Roth, and Philipp Slusallek. Ray Tracing Element for cell/b.e. In *Poster, High Performance Graphics (HPG) 2009, New Orleans*, August 2009.

[5] Flachs et al. A Streaming Processing Unit for a CELL Processor. In *IEEE International Solid-State Circuits Conference*, pages 134–135, 2005.

[6] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.

[7] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[8] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[9] Daniel Pohl. Quake wars gets ray traced. *Visual Andrenaline*, 0, 2009.

[10] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, 26(3), September 2007. (Proceedings of Eurographics), to appear.

[11] Karthik Ramani, Christiaan P. Gribble, and Al Davis. Streamray: a stream filtering architecture for coherent ray tracing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 325–336, New York, NY, USA, 2009. ACM.

[12] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM.

[13] Jörg Schmittler. *SaarCOR - A Hardware-Architecture for Realtime Ray Tracing*. PhD thesis, Saarland University, 2006.

[14] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[15] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[16] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[17] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.

[18] Sven Woop. *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes*. PhD thesis, Saarland University, 2006.

[19] Sven Woop, Erik Brunvand, and Philipp Slusallek. Estimating Performance of a Ray-Tracing ASIC Design. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 7–14, September 2006.

[20] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, pages 67–77, 2006.