

Architecture Considerations for Tracing Incoherent Rays

Timo Aila

Tero Karras

NVIDIA Research

Abstract

This paper proposes a massively parallel hardware architecture for efficient tracing of incoherent rays, e.g. for global illumination. The general approach is centered around hierarchical treelet subdivision of the acceleration structure and repeated queuing/postponing of rays to reduce cache pressure. We describe a heuristic algorithm for determining the treelet subdivision, and show that our architecture can reduce the total memory bandwidth requirements by up to 90% in difficult scenes. Furthermore the architecture allows submitting rays in an arbitrary order with practically no performance penalty. We also conclude that scheduling algorithms can have an important effect on results, and that using fixed-size queues is not an appealing design choice. Increased auxiliary traffic, including traversal stacks, is identified as the foremost remaining challenge of this architecture.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

We study how GPUs and other massively parallel computing architectures could evolve to more efficiently trace incoherent rays in massive scenes. In particular our focus is on how architectural as well as algorithmic decisions affect the memory bandwidth requirements in cases where memory traffic is the primary factor limiting performance. In these cases, the concurrently executing rays tend to access different portions of the scene data, and their collective working set is too large to fit into caches. To alleviate this effect, the execution needs to be modified in some non-trivial ways.

It is somewhat poorly understood when exactly the memory bandwidth is the primary performance bottleneck of ray tracing. It was widely believed that that was already the case on GPUs but Aila and Laine [AL09] recently concluded that in simple scenes that was not the case, even without caches. Yet, those scenes were very simple. We believe that applications will generally follow the trend seen in rendered movies, for example Avatar already has billions of geometric primitives in most complex shots. If we furthermore target incoherent rays arising from global illumination computations, it seems unlikely that currently available caches would be large enough to absorb a significant portion of the traffic.

Also, the processing cores of current massively parallel

computing systems have not been tailored for ray tracing computations, and significant performance improvements could be obtained by using e.g. fixed-function hardware for traversal and intersection [SWS02, SWW*04, WSS05]. However, for such custom units to be truly useful, we must be able to feed them enough data, and eventually we will need to sustain immense data rates when the rays are highly incoherent, perhaps arising from global illumination computation. In this paper we seek architectural solutions for situations where memory traffic *is* the primary bottleneck, without engaging into further discussion about when and where these conditions might arise.

We study this problem on a hypothetical parallel computing architecture, which is sized according to NVIDIA Fermi [NVI10]. We observe that with current approaches the memory bandwidth requirements can be well over hundred times higher than the theoretical lower bound, and that traversal stack traffic is a significant issue. We build on the ray scheduling work of Pharr et al. [PKG97] and Navratil et al. [NFLM07], and extend their concept of queue-based scheduling to massively parallel architectures. We analyze the primary architectural requirements as well as the bottlenecks in terms of memory traffic.

Our results indicate that scheduling algorithms can have an important effect on results, and that using fixed-size

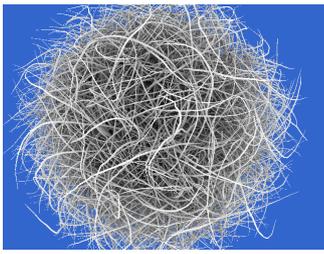
	VEGETATION	HAIRBALL	VEYRON
			
Number of triangles	1.1M	2.9M	1.3M
Number of BVH nodes	629K	1089K	751K
Total memory footprint	86MB	209MB	104MB
Traffic lower bound	158MB	104MB	47MB

Table 1: Key statistics of the test scenes. Half of the BVH nodes are leaves. VEGETATION and HAIRBALL are difficult scenes considering their triangle count, whereas VEYRON has very simple structure.

queues is not an appealing design choice. While our approach can reduce the scene data-related transfers by up to 95%, the overall savings are somewhat smaller due to traversal stacks and newly introduced queue traffic. An important feature of the proposed architecture is that the memory traffic is virtually independent of ray ordering.

2. Test setup

We developed a custom architecture simulator for the measurements. In order to have a realistic amount of parallelism, we sized our ray tracing hardware roughly according to NVIDIA Fermi: 16 processors, each 32-wide SIMD, 32×32 threads/processor for latency hiding, each processor coupled with a private L1 cache (48KB, 128B lines, 6-way), and a single L2 cache (768KB, 128B lines, 16-way) shared among the processors. Throughout this paper we will use "B" to denote byte. Additionally, we chose round robin scheduling and implemented the L1 and L2 as conventional write-back caches with LRU eviction policy.

We assume that the processors are very fast in acceleration structure traversal and primitive intersection tasks, possibly due to a highly specialized instruction set or dedicated fixed-function units, similar to the RPU architecture [WSS05]. Whether the processors are programmable or hardwired logic is not relevant for the rest of the paper.

Our primary focus will be on the amount of data transferred between the chip and external DRAM. We count the number of DRAM atoms read and written. The size of the atoms is 32B, which gives good performance with currently available memories. We ignore several intricate details of DRAM, such as page activations or bus flips (read/write), because existing chips already have formidable machinery for optimizing the request streams for these aspects.

We assume that the memory transfer bottleneck is either between L2 and DRAM or between L1 and L2, and not elsewhere in the memory hierarchy. As a consequence, we require that each L1 can access multiple cachelines per clock

(e.g. [Kra07] p. 110), as otherwise the L1 fetches could become a considerable bottleneck. We also acknowledge that L2 caches are not infinitely fast compared to DRAM — for example AMD HD 5870's L2 is only about $3 \times$ faster than DRAM — so we need to keep an eye on the ratio between the total L2 and DRAM bandwidth requirements.

2.1. Rays

We are primarily interested in incoherent rayloads that could arise in global illumination computations. These rays typically start from surfaces and need to find the closest intersection; furthermore any larger collection of rays tends to contain rays going to almost all directions and covering large portions of the scene. The exact source of such rays is not important for our study, and we choose to work with diffuse interreflection rays. We first determine the primary hit point for each pixel of a 512×384 viewport, and then generate 16 diffuse interreflection rays per pixel, distributed according to the Halton sequence [Hal60] on a hemisphere and furthermore randomly rotated around the normal vector. This gives us 3 million diffuse rays, and we submit these in batches of 1 million rays to the simulator. Each batch corresponds to a rectangle on the screen (256×256 , 256×256 , 512×128); we did not explore other batch sizes or shapes.

To gauge the effect of ray ordering we sort the rays inside a batch in two ways prior to sending them to the simulator. MORTON sorting orders the rays according to a 6D (origin and direction) space-filling curve. This order is probably better than what one could realistically accomplish in practice. RANDOM sorting orders the rays randomly, arriving at pretty much the worst possible ordering. From a usability point of view, an ideal ray tracing platform would allow submitting the rays in a more or less arbitrary order without compromising the performance. Also, the benefit of static reordering can be expected to diminish with increasing scene complexity as the nodes visited by two neighboring rays will generally be further apart in the tree.

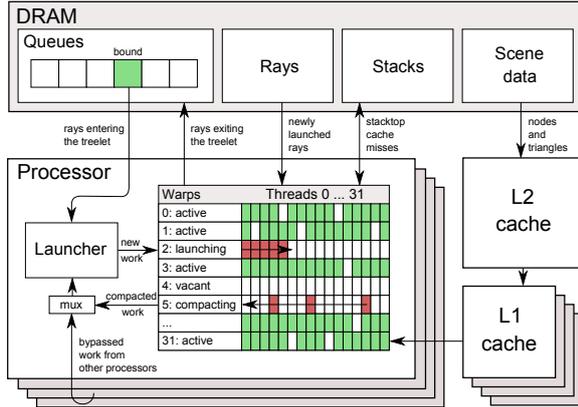


Figure 1: High-level view of our architecture along with the most important aspects of data flow.

2.2. Scenes

Ideally, we would conduct experiments with very large and difficult scenes. Due to simulator infrastructure weaknesses (execution speed, memory usage) we had to settle for much smaller scenes, in the range of 1–5M triangles, but we were able to select scenes with fairly difficult structure. In fact, the number of triangles in a scene is a poor predictor for the amount of memory traffic. With highly tessellated compact objects such as cars and statues, rays tend to terminate quickly after having descended to a leaf node. With vegetation and other organic shapes, on the other hand, it is common for rays to graze multiple surfaces before finally intersecting one. We therefore selected two organic scenes, VEGETATION and HAIRBALL, in addition to a car interior VEYRON. Key statistics of the scenes are shown in Table 1. Traffic lower bound is the total amount of scene-related traffic when each node and triangle is fetched exactly once per batch. It can exceed the footprint of the scene because we trace the rays in three batches.

We use bounding volume hierarchies (BVH) built using greedy surface area heuristic (SAH) without triangle splitting. The maximum size of a leaf node is set to 8 triangles.

3. Architecture

We will start by explaining some basic concepts and the functionality of our architecture. We then run an existing GPU ray tracer on our architecture and analyze its memory bandwidth requirements. It turns out that stack-related traffic is an important subproblem, and we introduce a simple approach for avoiding most of it. The ideas in this section can be employed independently from the more advanced aspects of our architecture that will be explained in Section 4.

Figure 1 illustrates the basic concepts of our architecture. Each processor hosts 32 warps and each warp consists of a static collection of 32 threads that execute simultaneously

Data type	Size	Contents
Ray	32B	Origin, direction, tmin, tmax
Ray state	16B	Ray index (24 bits), stack pointer (8 bits), current node index, closest hit (index, t)
Node	32B	AABB, child node indices
Triangle	32B	Three vertex positions, padding
Stack entry	4B	Node index

Table 2: The most common data types.

in SIMD fashion. We assume one-to-one mapping between rays and threads. Traversal and intersection require approximately 30 registers for each thread (32 bits per register); about 12 for storing the ray and ray state (Table 2), and the rest for intermediate results during computation. Each processor contains a launcher unit, that is responsible for fetching work from a queue the processor is currently bound to. The queue stores ray states, and the ray itself is fetched later directly from DRAM. Launcher copies the data from the queue to a warp’s registers. Once the warp gets full, it is ready for execution, and the launcher starts filling the next vacant warp if one exists.

At this point, it is sufficient to only consider a single queue (i.e. the input queue), to which all processors are bound from the beginning. The queue is initialized to contain all the rays of a batch, and processors consume rays from it whenever one or more of their warps become vacant. Since ray states in the input queue always represent new rays that start from the root, it would be possible to optimize their memory footprint by omitting the unnecessary fields. However, the total traffic generated by the input queue is negligible in any case, so we choose to ignore such optimizations for simplicity. A scenario with multiple queues is introduced in Section 4.

3.1. Work compaction

Whenever a warp notices that more than 50% of its rays have terminated, it triggers compaction if the launcher grants a permission for it. The terminated rays write their result to memory and are ejected from the processor. Non-terminated rays are sent back to the launcher, which then copies their data to the warp it is currently filling. The process is similar to fetching new rays from a queue, but in case of compaction we also copy the ray to avoid having to read it again. Since compaction is a relatively rare operation, it does not have to happen instantaneously.

Compaction is very important with highly divergent ray loads. Without it the percentage of non-terminated rays dropped to as low as 25% in our tests. With this kind of simple compaction we can sustain around 60–75% even in difficult scenarios, which suggests over $2\times$ potential increase in overall performance. However, since compaction increases the number of concurrently executing rays, their collective working set grows as well. While this inevitably puts more pressure on the caches, it is hard to imagine a situation where

Test setup	Total traffic (GB)	Scene traffic L2↔DR. (GB)	Stack traffic L2↔DR. (GB)	L1↔L2 vs. L2↔DR. required	Threads alive (%)
VEGETATION M	13.6	5.6	7.7	2.3×	75
VEGETATION R	24.9	12.5	12.1	1.6×	73
HAIRBALL M	11.2	5.0	6.0	2.0×	75
HAIRBALL R	18.5	9.2	9.1	1.6×	65
VEYRON M	1.9	0.6	1.1	6.9×	68
VEYRON R	9.2	3.9	5.1	2.5×	73

Table 3: Measurements for the baseline method [AL09] with compaction. The remaining ~ 0.2 GB of traffic is due to ray fetches and result write. M=MORTON, R=RANDOM.

the resulting increase in memory traffic would nullify the benefits of increased parallelism.

Compaction is turned on in all tests so that the numbers are comparable.

3.2. Baseline ray tracing method

We take the persistent while-while GPU tracer [AL09] as our baseline method. In their method processors are pointed to a pool of rays and autonomously fetch new rays whenever a warp terminates. During execution a ray always fetches and intersects the two child nodes together, proceeds to the closer intersected child, and pushes the other intersected child’s index to a per-ray traversal stack.

Table 3 shows the baseline method’s memory traffic in our scenes with MORTON and RANDOM sorting of rays. The total L2↔DRAM bandwidth requirements are very large compared to the lower bound shown in Table 1. With RANDOM we see 150–200× more traffic than theoretically necessary. MORTON sorting reduced the traffic almost 5× in VEYRON, but in other scenes the effect was surprisingly small given that this is basically the worst-to-best case jump. In VEYRON we can furthermore see that while DRAM traffic was greatly reduced, the required L2 bandwidth compared to DRAM bandwidth was rather large (6.9×), implying that L1 caches were not effective even in this simple case.

3.3. Stack top cache

Our baseline method uses the the stack memory layout of Aila and Laine [AL09], who kept them in CUDA’s [NV108] thread-local memory. This memory space is interleaved so that the first stack entry of 32 adjacent threads (i.e. warp) are consecutive in memory, same for the second entry, and so on. This is an attractive memory layout as long as the stack accesses of a warp stay approximately in sync, but with incoherent rays this is no longer the case and reordering of rays due to compaction further amplifies the problem. An additional test that ignored all traversal stack traffic revealed that stacks are an important problem and responsible for approximately half of the total traffic in all six cases of Table 3.

Test setup	Total traffic N = 4 (GB)	Stack traffic N = 4 (GB)	Total traffic N = 8 (GB)	Stack traffic N = 8 (GB)
VEGETATION MORTON	6.0	0.186	5.9	0.016
VEGETATION RANDOM	12.9	0.186	12.8	0.016
HAIRBALL MORTON	5.3	0.104	5.2	0.009
HAIRBALL RANDOM	9.6	0.104	9.5	0.009
VEYRON MORTON	0.9	0.119	0.8	0.006
VEYRON RANDOM	4.2	0.119	4.1	0.006

Table 4: Total and stack traffic measurements for baseline augmented with an N-entry stack top cache.

An alternative layout is to dedicate a linear chunk of stack memory for each ray. While this approach is immune to compaction and incoherence between rays, all rays are constantly accessing distinct cache lines. Since this exceeds the capacity of our caches, thrashing follows.

Horn et al. [HSHH07] describe a *shortstack* that maintains only a few topmost (latest) stack entries, and restarts (kd-tree) traversal from root node with a shortened ray when the top is exhausted. This policy can lead to a very significant amount of redundant work when the missing stack entries are reconstructed. We use a related approach that keeps the full per-ray traversal stack in memory and accesses it through a stack top cache, whose logic is optimized for direct DRAM communication. Our implementation uses a ring buffer with capacity of N entries (the size is dynamic $[0, N]$), each of which consists of 4B data and a dirty flag. The ring buffer is initially empty. Push and pop are implemented as follows:

- **push:** Add a new entry to the end of the ring buffer, mark it as dirty, and increment the stack pointer. In case of overflow, evict the first entry. If the evicted entry was marked as dirty, write all entries belonging to the same atom to DRAM. Mark them as non-dirty and keep them in cache.
- **pop:** Remove an entry from the end of the ring buffer and decrement the stack pointer. In case of underflow, fetch the required entry as well as all earlier stack entries that belong to the same DRAM atom, and add them to the ring buffer as non-dirty. If the number of entries exceeds N , discard the earliest ones.

We maintain the ring buffer for each thread in the register file because a tiny stack top of 4 entries typically suffices, and it would be wasteful to allocate a 128B cacheline for it when the register file can store it in 16B. The stack top absorbs almost all traversal stack traffic and consequently reduces the total traffic by approximately 50% in our tests, as shown in Table 4. In the rest of the paper we will refer to the baseline with stack top as *improved baseline*.

To avoid unnecessary flushes during compaction (Section 3.1), we copy the cached stack entries similarly to the rest of the state. However, if we push a ray to a queue, we

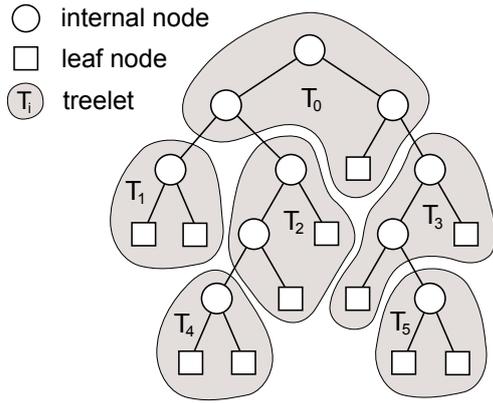


Figure 2: Example treelet subdivision of an acceleration structure.

must flush dirty stack entries, as we have chosen not to store cached stack entries in the queue. This design gave slightly lower memory traffic on our system but if the DRAM atom was any larger, it would be better to store the cached entries to the queue.

4. Treelets

The primary problem with the baseline method is that rays explore the scene independently of each other, and thus the concurrent working set can grow very large. This problem was approached by Pharr et al. [PKG97] in the context of out-of-core ray tracing. They collected rays that enter nodes of a uniform space subdivision, and later on processed the rays as batches in order to reduce swapping. More recently, Navratil et al. [NFLM07] optimized L2 \leftrightarrow DRAM traffic of a sequential ray tracer with a setup that collected rays into *queue points*, and then processed all rays in one queue point together. Each queue point was placed so that the entire subtree under it fits into cache. All data between the root and queue points was assumed to stay in cache, and thus the related memory traffic was not tracked. There was also no stack traffic since whenever a ray exited a queue point, it was shortened and traversal restarted from the root node. We extend this work to hierarchical queue points and massive parallelism.

Figure 2 shows a simple tree that has been partitioned into six *treelets*. Nodes are statically assigned to treelets when the BVH, or any other acceleration structure, is built (Section 4.1). We store this assignment into node indices that encode both a treelet index and a node index inside the treelet. A ray starts the traversal from the root and proceeds as usual until it encounters a treelet boundary. At this point the processing is suspended and the ray's state is pushed to a queue that corresponds to the target treelet. At some point a scheduler (Section 4.2) decides that it is time to start processing rays from the target treelet. Hopefully by this time many more

```

1: // Start with an empty treelet.
2: cut  $\leftarrow$  treeletRoot
3: bytesRemaining  $\leftarrow$  maxTreeletFootprint
4: bestCost[treeletRoot]  $\leftarrow$   $\infty$ 
5:
6: // Grow the treelet until it is full.
7: loop
8: // Select node from the cut.
9: (bestNode, bestScore)  $\leftarrow$  ( $\emptyset$ ,  $-\infty$ )
10: for all  $n \in$  cut do
11:   if footprint[ $n$ ]  $\leq$  bytesRemaining then
12:     gain  $\leftarrow$  area[ $n$ ] +  $\epsilon$ 
13:     price  $\leftarrow$  min(subtreeFootprint[ $n$ ], bytesRemaining)
14:     score  $\leftarrow$  gain / price
15:     if score > bestScore then
16:       (bestNode, bestScore)  $\leftarrow$  ( $n$ , score)
17:     end if
18:   end if
19: end for
20: if bestNode =  $\emptyset$  then break
21:
22: // Add to the treelet and compute cost.
23: cut  $\leftarrow$  (cut  $\setminus$  bestNode)  $\cup$  children[bestNode]
24: bytesRemaining  $\leftarrow$  bytesRemaining - footprint[bestNode]
25: cost  $\leftarrow$  (area[treeletRoot] +  $\epsilon$ ) +  $\sum_{n \in \text{cut}}$  bestCost[ $n$ ]
26: bestCost[treeletRoot]  $\leftarrow$  min(bestCost[treeletRoot], cost)
27: end loop

```

Figure 3: Pseudocode for treelet assignment using dynamic programming. This code is executed for each node in reverse depth-first order.

rays have been collected so that a of number rays will be requesting the same data (nodes and triangles) and thus the L1 cache can absorb most of the requests. The tracing of the rays then continues as usual, and when another treelet boundary is crossed the rays are queued again.

While a treelet subdivision can be arbitrary, it should generally serve two goals. We want to improve cache hit rates and therefore should aim for treelets whose memory footprint approximately matches the size of L1. We should also place the treelet boundaries so that the expected number of treelet transitions per ray is minimized because each transition causes memory traffic.

The rest of the paper assumes one-to-one mapping between treelets and queues, and that a ray can reside in at most one queue at any given time. While one could speculatively push a ray to multiple queues, we have found that this is not beneficial in practice.

4.1. Treelet assignment

The goal of our treelet assignment algorithm is to minimize the expected number of treelets visited by a random ray. Disregarding occlusion, the probability of a long random ray intersecting a treelet is proportional to its surface area. We therefore chose to optimize the total surface area of

the treelet roots, which was also experimentally verified to have a strong correlation with treelet-related memory traffic. Our method produces treelets that have a single root node, although the rest of this paper does not rely on that.

We use dynamic programming, and the algorithm’s execution consists of two stages. In the first stage the pseudocode in Figure 3 is executed for each BVH node in reverse depth-first order (starting from leaves) to find the lowest-cost treelet that starts from the specified node. The expected cost of a treelet equals its root node’s surface area plus the sum of costs of all nodes in the cut, i.e. the set of nodes that are just below the treelet.

The algorithm is greedy because instead of trying all possible treelet shapes it selects nodes from the cut according to a simple heuristic (lines 12–14). The heuristic is primarily based on surface area: if a node has large area, a treelet starting from it would probably be intersected often. Since any node in the cut can be added to the current treelet without increasing the treelet’s area, we choose the largest. However, if a node in the cut represents a subtree with tiny memory footprint, we would like to include it into the current treelet instead of creating a separate, tiny treelet for it.

In the second stage of the algorithm nodes are assigned to treelets. We start from the root node, with essentially the same code, and enlarge the treelet until its cost equals the value that was stored during the first pass. Once the cost is met, the loop terminates and the function is called recursively for each node in the cut.

The additional control parameter ϵ encourages subdivisions with larger average treelet footprint. We prefer larger (and thus fewer) treelets for two reasons. First, fewer treelets means fewer queues during simulation. Second, our subsequent analysis of the optimal treelet size would be dubious if the average size was nowhere near the specified maximum. If the scene footprint (nodes and triangles) is S and maximum treelet size is T , at least S/T treelets will be needed. As a crude approximation for the expected surface area of the smallest treelets we use $\epsilon = \text{Area}_{BVH_{root}} * T / (S * 10)$ to discourage unnecessarily small treelets. This penalty has a negligible effect on the total surface area while increasing the average treelet memory footprint, e.g., from 3KB to 19KB.

Table 5 provides statistics for our test scenes with treelets not larger than 48KB. Our method deposited the BVH leaf nodes adaptively to levels 1–8, whereas prior work had exactly 2 layers of treelets [NFLM07]. As a further validation of our greedy treelet assignment, we also implemented an $O(NM^2)$ search that tried all possible cuts in the subtree instead of the greedy heuristic. Here, N and M denote the number of nodes in tree and treelet, respectively.

Scene	Num 48KB treelets	Avg. size (KB)	Stddev size	Avg. treelets per ray	Treelet layers	Area vs. optimal
VEGETATION	4588	19.3	15.4	11.3	1–5	+20%
HAIRBALL	14949	14.3	16.8	6.1	2–8	+15%
VEYRON	3563	30.0	13.3	9.2	1–8	+5%

Table 5: Treelet assignment statistics with 48KB max size.

4.2. Scheduler

The remaining task is to choose which queues the processors should fetch rays from. We will analyze two different scheduling algorithms for this purpose.

LAZY SCHEDULER is as simple as possible: when the queue a processor is currently bound to gets empty, we bind the processor to the queue that currently has most rays. This policy has several consequences. The scheduler tends to assign the same queue to many processors, and most of the time only a few treelets are being processed concurrently by the chip. Considering DRAM traffic, this behavior tends to favor treelets that are roughly the size of L2 cache. Also, scene traffic is well-optimized because queues are able to collect maximal number of rays before they are bound.

BALANCED SCHEDULER defines a target size for all queues (e.g. 16K rays). When a queue exceeds the target size, it starts requesting progressively more processors to drain from it. This count grows linearly from zero at the target size to the total number of processors at $2 \times$ target size. We also track how many processors are currently bound to a queue, and therefore know how many additional processors the queue thinks it needs. The primary sorting key of queues is the additional workforce needed, while the number of rays in the queue serves as a secondary key. The binding of a processor is changed if its current queue has too much workforce while some other queue has too little. The input queue that initially contains all rays requires special treatment: we clamp the number of processors it requests to four to prevent the input queue from persistently demanding all processors to itself. This policy leads to more frequent binding changes than LAZY scheduler, but different processors are often bound to different queues and therefore L1-sized treelets are favored. It also creates additional opportunities for queue bypassing (Section 4.4).

The graphs in Figure 4 show the total DRAM traffic as a function of maximum treelet size for both schedulers in VEGETATION. It can be seen that LAZY scheduler works best with L2-sized treelets and BALANCED scheduler with L1-sized treelets, although this simplified view disregards potential L1 \leftrightarrow L2 bottlenecks. In both cases the minimum traffic is reached with a maximum size that is slightly larger than the caches. Two possible explanations are that the average treelet size is always somewhat smaller than the maximum, and that rays may not actually access every node and triangle of a treelet, thus reducing the actual cache footprint.

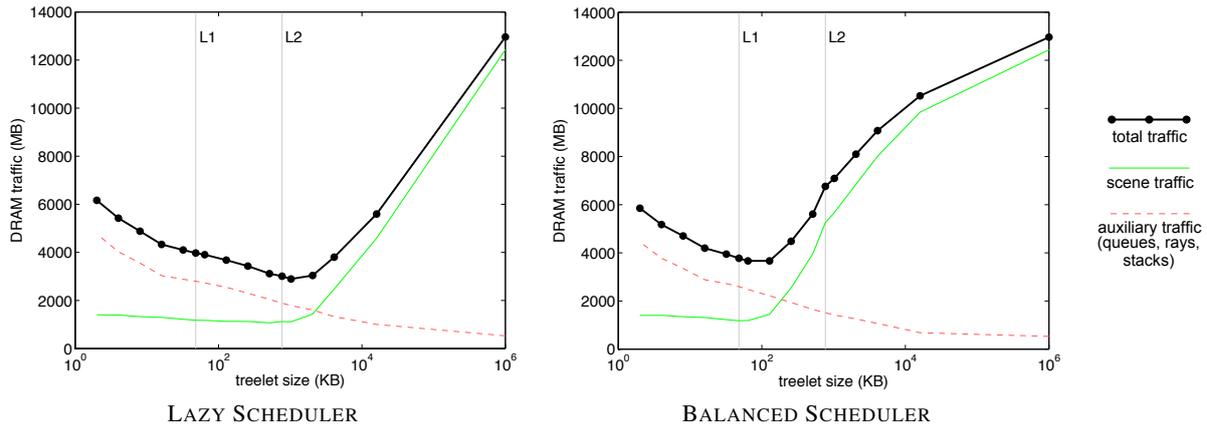


Figure 4: DRAM traffic as a function of treelet size in VEGETATION for both schedulers. Lazy scheduler gives the lowest total traffic with treelets that are sized according to L2, whereas the BALANCED scheduler works best with L1-sized treelets.

We will provide detailed results in Section 5, but it is already evident that scene traffic remains clearly higher than the lower bound for this scene (158MB). The crucial observation here is that we could expect to process each treelet once only if all rays visited the nodes in the same order. In practice, however, some rays traverse the scene left-to-right and others right-to-left. Taking this further, we can expect to traverse the scene roughly once for each octant of ray direction, which would suggest a more realistic expectation of $8\times$ the lower bound.

Ramani et al. [RGD09] propose an architecture that essentially treats every node as a treelet. As can be extrapolated from our graphs their design point is bound to exhibit very significant auxiliary traffic and therefore mandates keeping the rays and related state in on-chip caches, which is what Ramani et al. do. One challenge is that we have about 16K rays in execution at any given time and based on preliminary experiments, we would need to keep at least $10\times$ as many rays in on-chip caches to benefit significantly from postponing. Considering current technology, this is a fairly substantial requirement. The scene traffic might also be higher with their approach since we can expect to fetch the scene data approximately 8 times per a batch of rays, and currently our batch size can be in millions, whereas their approach would require smaller batches.

4.3. Queue management

We are implicitly assuming that queue operations are very fast, since otherwise our treelet-based approach might be limited by processing overhead instead of memory traffic. This probably requires dedicated hardware for queue operations, which should be quite realistic considering that plenty of other uses exist for queues [SFB*09]. Our queues are implemented as ring buffers with FIFO policy in external memory, and the traffic goes directly to DRAM because whenever we push a ray to a queue, we do not expect to need it

for a while. Caching would therefore be ineffective. Writes are compacted on-chip before sending the atoms to external memory; queues could therefore work efficiently with any DRAM atom size.

We tried two memory allocation schemes for the queues. The easier option is to dedicate a fixed amount of storage for each queue at initialization time. This means that queues can get full, and therefore pre-emption must be supported to prevent deadlocks. Our pre-emption moves the rays whose output is blocked back to the input queue, which is guaranteed to have enough space. With fixed-size queues, it is not clear to us whether any realistic scheduler could guarantee deadlock-free execution without pre-emption; at least our current schedulers cannot. BALANCED SCHEDULER can often avoid pre-emptions but it still needs them occasionally. However, the real problem with this approach is that a large amount of memory has to be allocated (e.g. thousands of entries per treelet) and most of this memory will never be used.

We implemented dynamic resizing of queues to use memory more efficiently and guarantee no deadlocks even without pre-emption. When a queue gets full, we extend it with another page (fixed-sized chunk of memory, e.g. 256 entries) from a pool allocator. Queues maintain a list of pages in their possession. When a page is no longer needed, it is returned to the pool. If we limit the system to have a certain maximum number of rays in flight at a time, the memory requirements are bounded by this threshold plus one partial page per treelet as well as one per processor. As long as the pool allocator has been appointed a sufficient amount of memory, queues cannot get full and hence deadlocking is impossible. This is the method actually used in our measurements. We did not study the possible microarchitecture of this mechanism in detail, but we do not foresee any fundamental problems with it. Lalonde [Lal09] describes other prominent uses for pool allocators in graphics pipelines.

Scene	Scheduler	Treelet max size (KB)	Treelet changes per ray	Total traffic (GB)	Scene traffic (cached)			Aux. traffic (uncached)			Threads alive (%)	Queue ops bypassed (%)
					L2↔DR. (GB)	L2↔DR. vs. lower bound	L1↔L2 vs. L2↔DR.	Queues (GB)	Rays (GB)	Stacks (GB)		
VEGETATION	LAZY	768	6.9	3.01	1.12	7.1×	7.7×	0.59	0.64	0.57	62	18
VEGETATION	BALANCED	768	6.9	6.76	5.26	33.3×	1.7×	0.44	0.49	0.48	52	41
VEGETATION	LAZY	48	11.3	3.97	1.18	7.5×	2.3×	0.89	0.93	0.89	64	23
VEGETATION	BALANCED	48	11.3	3.77	1.18	7.5×	1.7×	0.81	0.86	0.83	63	30
HAIRBALL	LAZY	768	2.7	1.78	0.94	9.0×	9.0×	0.24	0.29	0.24	62	25
HAIRBALL	BALANCED	768	2.7	5.78	5.05	48.5×	1.7×	0.19	0.24	0.21	53	43
HAIRBALL	LAZY	48	6.1	2.68	1.10	10.6×	2.4×	0.50	0.55	0.47	64	23
HAIRBALL	BALANCED	48	6.1	2.53	1.08	10.4×	1.7×	0.44	0.49	0.43	63	32
VEYRON	LAZY	768	5.7	1.73	0.16	3.4×	23.3×	0.48	0.52	0.47	64	21
VEYRON	BALANCED	768	5.7	2.94	1.75	37.2×	2.5×	0.34	0.39	0.37	51	47
VEYRON	LAZY	48	9.2	2.49	0.19	4.0×	3.3×	0.75	0.79	0.67	67	21
VEYRON	BALANCED	48	9.2	2.03	0.20	4.3×	3.6×	0.57	0.61	0.56	60	41

Table 6: Measurements for our treelet-based approach with two schedulers using 48KB and 768KB treelets. Stack top size was fixed to four and bypassing according to two previous queue bindings was allowed. Additionally, result writes used ~ 0.1 GB.

4.4. Queue bypassing

We can avoid the round-trip through a queue if the queue is already bound to another processor. This optimization is obviously possible only in parallel architectures. Since there is no need to preserve the ordering of rays, we can forward the ray along with its ray state and stack top to the another processor’s launcher. This is basically the same operation as compaction (Section 3.1) but to a different processor. In our architecture, as well as in most massively parallel designs, any two processors can be reached from the shared L2 cache, and thus the *queue bypass* operation can be carried out at L2 bandwidth without causing DRAM traffic.

Since the stack top cache talks directly to DRAM, there is no risk that some of the traversal stack data would be only in the previous processor’s L1 cache. Therefore coherent L1 caches are not required.

We can optionally allow bypassing decisions according to e.g. two previous queues in addition to the current one. The justification is that whenever a processor is bound to a new queue, it will continue to have rays from the previous queue/treelet in flight for some time. Also, towards the end many treelets will have fewer rays than fit into a processor and therefore rays from more than two treelets will be in flight simultaneously. Having multiple treelets in flight does not immediately lead to working set explosion because when a smaller number rays propagate inside a treelet, only a part of its data is accessed.

5. Results

Table 6 shows results for RANDOM sorted rays in the three scenes using both schedulers with L1 and L2-sized treelets. We do not provide separate results for MORTON sorting because they were virtually identical with RANDOM; it therefore does not matter in which order the rays are submitted

to the simulator. The lowest total traffic was obtained in all scenes with LAZY scheduler and L2-sized treelets. For this design point to be realistic L2 should be 7.7 – 23.3× faster than DRAM, which would require a radical departure from current architectures. The second lowest total traffic was obtained with BALANCED scheduler and L1-sized treelets, and this combination places only modest assumptions on L2 speed (1.7 – 3.6× DRAM). In the following we will focus on the latter design point.

Figure 5 shows the total and scene traffic for baseline, improved baseline (baseline with stack top cache), and treelets with RANDOM sorted rays. The improved baseline essentially eliminates the stack traffic, and the total traffic is roughly halved. The reduction is virtually identical for MORTON sorted rays (Tables 3 and 4).

Treelets offer an additional 50–75% reduction in total memory bandwidth on top of the improved baseline, yielding a total reduction of 80-85%. Figure 5 also reveals that when treelets are used, the scene traffic is greatly diminished but the auxiliary traffic (rays, stacks, queues) increases a lot. This is the most significant remaining problem with the treelet-based approach, and the reason why it can increase memory traffic in simple scenes. With MORTON sorted rays the reduction on top of the improved baseline is still around 50% in VEGETATION and HAIRBALL but in VEYRON treelets actually lose (2.0GB vs 0.9GB).

These results assume that bypassing is allowed to two previous queues. Table 7 shows that if bypassing is allowed only to the current queue, the total traffic would increase by about 10%, and disabling bypassing would cost another 10%. Increasing the number of previous queues further did not help.

To gauge scalability, we performed additional tests with 64 processors and L1 caches instead of 16, all other parameters held constant. With BALANCED scheduler and 48KB treelets the total memory traffic changed very little (<20%)

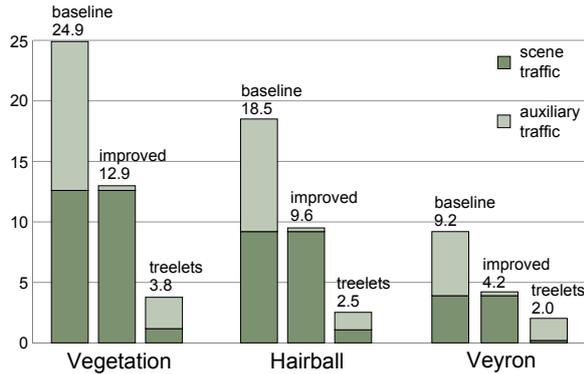


Figure 5: Memory traffic with RANDOM sorted rays for baseline, improved baseline, and treelets (from left to right).

and actually decreased in VEYRON. While the scene traffic did increase somewhat, there were more bypassing opportunities on a larger chip, which reduced the auxiliary traffic. This suggests that our approach scales well, and that a relatively small L2 can be sufficient even in massive chips.

So far we have assumed that each processor can drain data from L1 at least 512 bits at a time (2 adjacent sibling nodes). This may not be the case on existing architectures; for example on NVIDIA Fermi the widest load instruction is 128 bits. This means that four consecutive loads are needed, and there is a possibility that the data gets evicted from L1 between these loads. We introduced this constraint to the 16-processor chip, and it almost doubled the total traffic in the baseline, whereas our treelet-based approach was barely affected because cache thrashing happens so rarely. In the most difficult cases, RANDOM sorted rays in VEGETATION and HAIRBALL, the reduction in total memory traffic compared to the baseline was now slightly over 90%, whereas the simplest case (VEYRON with MORTON) was unaffected.

6. Possible future improvements

6.1. Batch processing vs. continuous flow of rays

We have concentrated on processing the rays in batches. A potentially appealing alternative would be to let the tracing process run continuously so that new rays could be pulled in whenever the scheduler so decides. Unfortunately our current schedulers do not guarantee fairness in the sense that rays that end up in a rarely visited treelet could stay there indefinitely long. With batch processing this cannot happen since all queues/treelets are processed upon the batch's end. It should be possible to improve the schedulers so that fairness could be guaranteed. We also suspect, without tangible evidence, that the schedulers can be improved in other ways as well. Another complication from continuous flow of rays is memory allocation for traversal stacks, ray data, and ray results. In batch processing all of this is trivial, but when

Test	Total traffic (GB)	Scene L2↔D. (GB)	Aux. traffic (GB)			By-pass (%)
			Queues (GB)	Rays (GB)	Stacks (GB)	
VEGETATION 2 PQ	3.77	1.18	0.81	0.86	0.83	30
VEGETATION 0 PQ	4.07	1.21	0.92	0.97	0.88	20
VEGET. no bypass	4.61	1.23	1.14	1.19	0.97	0
HAIRBALL 2 PQ	2.53	1.08	0.44	0.49	0.43	32
HAIRBALL 0 PQ	2.68	1.09	0.50	0.55	0.46	28
HAIRB. no bypass	3.00	1.10	0.63	0.68	0.51	0
VEYRON 2 PQ	2.03	0.20	0.57	0.61	0.56	41
VEYRON 0 PQ	2.41	0.20	0.71	0.76	0.64	24
VEYRON no bypass	2.94	0.20	0.93	0.98	0.74	0

Table 7: Measurements without bypassing, and without and with previous queues using BALANCED scheduler and 48KB treelets. PQ=previous queue bindings.

individual rays come and go, a more sophisticated memory allocation scheme would be called for.

6.2. Stackless traversal

Stack traffic constitutes 17–28% of overall memory communication when treelets are used. One way to remove it altogether would be to use fixed-order skiplist traversal [Smi98], which would also reduce the scene traffic since there is a well defined global order for processing the treelets. The potentially significant downside of fixed-order traversal is that rays are not traversed in front-to-back order and in scenes that have high depth complexity a considerable amount of redundant work may be performed. That said, in scenes with low depth complexity (e.g. VEYRON) fixed-order traversal might indeed be a good idea, and with shadow rays that do not need to find the closest hit, it may generally win.

6.3. Wide trees

Our preliminary tests suggest that wide trees [DHK08, WBB08] could allow visiting fewer treelets per ray than binary trees. Wide trees have other potentially favorable properties too. One could map the node intersections to N lanes in an N -wide tree, instead of dedicating one ray per thread. As a result, the number of concurrently executing rays would be reduced to $1/N$ of the original, and the scene-related memory traffic should also be lower. We consider this a promising avenue of future work.

6.4. Compression

Both the baseline method and our treelet-based approach would benefit from a more compact representation of scene data and rays. The first step could be to drop the planes from BVH nodes that are shared with the parent node [Kar07], or possibly use less expressive but more compact nodes, such as bounding interval hierarchy [WK06]. Alternatively some variants of more complex compression methods [LYM07, KMKY09] might prove useful.

It is not clear to us how rays could be compressed. One could imagine quantizing the direction, for example, but presumably that would have to be controllable by the application developer. One could also represent, e.g., hemispheres as a bunch of directions and a shared origin, although in our design that would in fact increase the bandwidth since one would need to fetch two DRAM atoms per ray (unique and shared components). Nevertheless, this is a potentially fruitful topic for future work.

6.5. Keeping rays on-chip

Looking much further to the future, it may be possible that L3 caches of substantial size (e.g. 100MB) could be either pressed on top of GPUs as stacked die or placed right next to GPUs and connected via high-speed interconnect. In such a scenario it might make sense to keep a few million rays and perhaps stack tops on chip, thus eliminating the related memory traffic.

6.6. Shading

We have completely ignored shading in this paper. We postulate that the queues offer a very efficient way of collecting shading requests for a certain material, shader, or object, and then executing them when many have been collected, similarly to the stream filtering of Ramani et al. [RGD09].

7. Conclusions

Our simulations show that stack top caching and the treelet-based approach can significantly reduce the memory bandwidth requirements arising from incoherent rays. We believe that for many applications the possibility of submitting rays in an arbitrary order will be a valuable feature.

In terms of getting the required features actually built, several milestones can be named. Before dedicated support makes financial sense, ray tracing has to become an important technology in mainstream computing. We also have to have important applications whose performance is limited by the memory bandwidth. Regardless, we believe that most of the required features, such as hardware-accelerated queues, have important uses outside this particular application.

Acknowledgements Thanks to Peter Shirley and Lauri Savioja for proofreading. Vegetation and Hairball scenes courtesy of Samuli Laine.

References

- [AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009* (2009), pp. 145–149.
- [DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Comp. Graph. Forum* 27, 4 (2008), 1225–1234.
- [Hal60] HALTON J.: On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik* 2, 1 (1960), 84–90.
- [HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *Proc. Symposium on Interactive 3D graphics and games* (2007), ACM, pp. 167–174.
- [Kar07] KARRENBERG R.: Memory aware realtime ray tracing: The bounding plane hierarchy. Bachelor thesis, Saarland University, 2007.
- [KMKY09] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: random-accessible compressed bounding volume hierarchies. In *ACM SIGGRAPH '09: Posters* (2009), pp. 1–1.
- [Kra07] KRASHINSKY R. M.: *Vector-Thread Architecture And Implementation*. PhD thesis, MIT, 2007.
- [Lal09] LALONDE P.: Innovating in a software graphics pipeline. ACM SIGGRAPH 2009 course: Beyond programmable shading, 2009.
- [LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *Proc. IEEE Symposium on Interactive Ray Tracing 2007* (2007), pp. 19–26.
- [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 Symposium on Interactive Ray Tracing 2007* (2007), pp. 95–104.
- [NVI08] NVIDIA: *NVIDIA CUDA Programming Guide Version 2.1*. 2008.
- [NVI10] NVIDIA: Nvidia's next generation CUDA compute architecture: Fermi. Whitepaper, 2010.
- [PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proc. ACM SIGGRAPH 97* (1997), pp. 101–108.
- [RGD09] RAMANI K., GRIBBLE C. P., DAVIS A.: Streamray: a stream filtering architecture for coherent ray tracing. In *ASPLOS '09: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ACM, pp. 325–336.
- [SFB*09] SUGERMAN J., FATAHALIAN K., BOULOS S., AKLEY K., HANRAHAN P.: Gramps: A programming model for graphics pipelines. *ACM Trans. Graph.* 28, 1 (2009), 1–11.
- [Smi98] SMITS B.: Efficiency issues for ray tracing. *J. Graph. Tools* 3, 2 (1998), 1–14.
- [SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: Saarcor: a hardware architecture for ray tracing. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), pp. 27–36.
- [SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2004), pp. 95–106.
- [WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets: Efficient SIMD single-ray traversal using multi-branching BVHs. In *Proc. IEEE/Eurographics Symposium on Interactive Ray Tracing 2008* (2008), pp. 49–57.
- [WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Proc. Eurographics Symposium on Rendering 2006* (2006), pp. 139–149.
- [WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.* 24, 4 (2005), 434–444.