High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Reduced Precision for Hardware Ray Tracing in GPUs

S. Keely[†]

University of Texas at Austin

Abstract

We propose a high performance, GPU integrated, hardware ray tracing system. We present and make use of a new analysis of ray traversal in axis aligned bounding volume hierarchies. This analysis enables compact traversal hardware through the use of reduced precision arithmetic. We also propose a new cache based technique for scheduling ray traversal. With the addition of our compact fixed function traversal unit and cache mechanism, we show that current GPU architectures are well suited for hardware accelerated ray tracing, requiring only small modifications to provide high performance. By making use of existing GPU resources we are able to keep all rays and scheduling traffic on chip and out of caches. We used simulations to estimate the performance of our architecture. Our system achieves an average ray rate of 3.4 billion rays per second while path tracing our test scenes.

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 hardware acceleration of axis aligned bounding volume hierarchy (BVH) ray tracing and how GPUs can grow to accommodate it. Recently there has been renewed interest in hardware ray tracing, particularly in mobile devices. The combination of large displays and low complexity scenes makes ray tracing an attractive option in the mobile space, particularly for coherent effects such as shadows and specular reflections. In contrast we seek ray tracing support for incoherent effects in large scenes, such as global illumination in scenes with several million polygons.

Several ray tracing hardware architectures have been proposed centered around treelet scheduling [AK10] and utilizing MIMD ray traversal. Scheduling ray traversal based on queuing at treelets, small sub-trees of the BVH, has been shown to be an effective means of obtaining cache reuse [KSS*13, AK10, NFLM07]. Ray divergence at queuing operations and the variable length of the queues makes MIMD ray traversal an attractive option, avoiding SIMD divergence costs. However, highly divergent MIMD execution seems to be at odds with the wide SIMT architectures most GPUs employ. Combined with the high arithmetic demand of ray traversal, this has led to the use of co-processor or fully custom designs. We also adopt compact MIMD ray tracing and treelet scheduling, and combined with GPU integration we are able to maintain high ray occupancy while keeping rays and scheduling traffic entirely on chip.

Our system makes use of a new analysis of BVH traversal in which intersection of a BVH node is a cumulative operation involving the intersection of all ancestor nodes. This reformulation enables small area, low power, ray traversal units through the use of low precision arithmetic. These units are small enough to be placed within GPU cores, thereby allowing resources to be shared between the SIMT GPU and the MIMD ray traversal units. As we will show this sharing is a natural fit despite the difference in execution model.

In common with prior work we use a form of treelet scheduling to maximize cache reuse. Treelet scheduling presents two opposing constraints, small treelets are needed to ensure maximum cache reuse but increase ray queuing and treelet selection costs. We resolve this difficulty by introducing a cache feedback mechanism. We queue rays at small treelets but allow the ray to traverse beyond the active treelet so long as the required node is *already in the cache*. This ensures that high cache reuse is obtained when treelets are scheduled and, as our simulations show, maintains low queuing frequency.

[†] e-mail:snkeely@gmail.com

DOI: 10.2312/hpg.20141091

Our simulations show that, together with simple BVH compression, this system provides an average ray rate of 3.4 billion rays per second (113 million rays per frame at 30 Hz) when integrated to a single current generation GPU.

2 Background

Algorithms for ray tracing can be roughly divided into two camps, packet and non-packet techniques. Packet techniques perform well for ray sets which are known to be coherent [ORM08, BW09, BWW*12, AL09, GPSS07]. On both CPU and GPU platforms these techniques make good use of SIMD processing because coherent rays exhibit low lane divergence. However, packet techniques are at a disadvantage when tracing incoherent rays, such as global illumination rays. As we are primarily interested in incoherent effects, we turn our attention to non-packet techniques.

Recently software techniques for ray tracing on both CPUs and GPUs appear to have reached a common performance limitation [WWB*14, AL09]. After a few hundred million rays per second both CPUs and GPUs appear to reach saturation, less for incoherent rays. As shown in [AL09] this is due to saturating core performance, likely floating point performance, not external memory bandwidth. To reach higher performance we and other researchers turn to hardware solutions.

Early ray tracing hardware research [SWS02, WSS05, RGD09] employed similar algorithms and scheduling as software techniques. Yet they showed benefits could be obtained from intersection and traversal pipelining, small dedicated caches, separating phases of ray tracing into separate hardware units, balancing the ratio of each type of unit, and mitigating packet divergence in a variety of interesting ways. However, bandwidth consumption remained a critical issue.

External memory bandwidth received a systematic treatment in [AK10]. Here memory bandwidth due to scene data is shown to be primarily due to the order of execution of rays. The scheduling concepts from [NFLM07] are leveraged to dynamically sort rays into spatially coherent sets as they are traced. These sets are defined by traversal into memory contiguous subtrees of the BVH, termed *treelets*. This is shown to reduce scene traffic by as much as 90%, however, auxiliary traffic due to ray queuing is introduced.

Real time ray tracing requires substantial floating point throughput. In these effects many rays must be cast per pixel in order to achieve acceptably low noise. With display resolutions at or above 2M pixels and 50-100 rays/pixel needed for low noise, we can expect to need on the order of 150 million rays per frame. At 30 Hz and 40 steps/ray this is an impressive floating point demand, requiring the rough equivalent of 4.5 TFlops of software performance just for ray traversal. As this is nearly the size of the largest GPU currently available it seems reasonable to consider fully custom architectures to support ray tracing. The most recent ray tracing hardware proposals do just this.

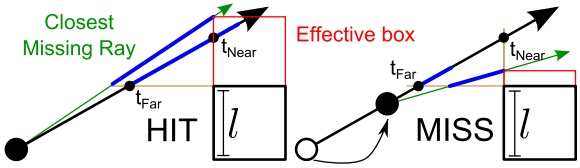

Figure 1: Traversal point update: $\varepsilon = 2ulp(t_{Far})$ is shown in blue. On the left the traversal point is distant from the far-plane of the box making ε large, causing a false hit. On the right we have applied a traversal point update. ε is now much smaller allowing us to detect a miss.

[KSS*13], and [SKKB09], propose programmable rendering architectures based on ray tracing. In contrast to current GPUs, these employ a MIMD design which aids in handling incoherent rays. [KSS*13] adopts treelet based scheduling and eliminates auxiliary off chip traffic by keeping all ray state in an L2 cache re-purposed as a programmer controlled scratchpad.

Ray tracing on mobile processors has received recent interest [LSL*13a, KKW*13, SKBD12]. Most of these operate as a co-processor to a GPU, using the GPU for programmable shading and ray generation. Each of these implements a MIMD ray tracing architecture and optimizes for low bandwidth. Most recently work has been done to lower the performance cost of co-processor design [SLL*13].

Our system adopts MIMD ray traversal, treelet scheduling, and GPU shading without requiring co-processor design. We are able to avoid co-processor design through aggressive sharing of internal resources and our compact ray traversal hardware.

3 Reduced Precision Computing

Our architecture exploits reduced precision to significantly reduce power and area. It is centered around a new analysis of BVH traversal which enables using reduced precision arithmetic in hierarchy traversal. Reduced precision enables power efficient, small area traversal units by greatly shrinking the needed ALUs, particularly the multipliers. These multipliers are a major sink of power and area, which scale quadratically with argument precision. For instance just the multipliers used for traversal in the above example load (4.5B RPS, 40 steps/ray, 12 mul/step) require ∼27W [KDK*11]. However, with reduced precision we need only ∼1W.

Reduced precision also reduces memory access energy by conserving bandwidth. We store BVH nodes with reduced precision by quantizing them, which enables compression. This reduces bandwidth both directly and by amplifying the effect of caches.

3.1 Reduced Precision BVH Traversal

In our formulation of BVH traversal each ray-box intersection leverages the result of intersection with ancestor boxes.

Key to our analysis is the recognition that typically relatively little refinement in box edge length is achieved at each level of a BVH hierarchy.

We use Smits' test for ray-box intersection, which computes the interval of the line within the box [Smi98]. If this interval, $[t_{Near}, t_{Far}]$, is inverted then the line does not intersect the box, otherwise it does. A study of the numerical robustness of the test shows that to declare a miss the interval must not only be inverted but t_{Near} must exceed t_{Far} by $\varepsilon = 2ulp(t_{Far})$ [Ize13]. This gives us a conservative miss test which we can safely use with arbitrarily reduced precision operations.

Alone this conservative test is of little help to us. For a distant box ε may dominate the effective size of the box, resulting in a false hit. Moving the ray origin closer to the box's near-plane reduces ε, returning the box to its natural size where we might detect a miss (See Fig. 1). To ensure that false hits due to ε are limited we must have $\varepsilon \ll l$, where l is the box's edge length. For a ray which misses $t_{Near} > t_{Far} \gg \varepsilon$, so we need a point on the ray such that $t_{Near} \lesssim l$. Our task is to design a fast BVH traversal algorithm, using only low precision arithmetic, which provides this sufficiently close point, which we call the traversal point.

Simply moving the traversal point to the near-plane of the box would suffice. Unfortunately this requires several multiplies which are precisely the operations we want to reduce the precision of. So we must find the minimum precision with which we can update the traversal point and maintain the required bound. We must also be able to apply the update repeatedly, at each step in the hierarchy, and maintain the bound.

Computing with t_{Near} with reduced precision gives $t'_{Near} = \gamma(\gamma(B-P)\cdot invD\gamma)$, where γ is the representation error due to reduced precision, P is the traversal point, and B is the near-plane coordinate of the box. We do not want the point to drift off the line so we should update the point with high precision. Fortunately t'_{Near} is a low precision number so without loss of accuracy $P' = P + t'_{Near}D$ can be implemented with a high-by-low precision multiply. This updates the traversal point by a factor $f = (1-\gamma^3)$ of the true near-plane, leaving a new non-zero t_{Near}.

If the box has unconstrained size then, since the new t_{Near} is non-zero, we can construct a box which violates the bound. To maintain the bound we must limit the ratio of the parent to child box dimensions. Let the limit be S, the maximum change in scale between parent and child. Applying the maximum scale change at each level allows us to determine the cumulative effect of truncation as we traverse the tree. When the ray is created we require that its origin be in or on the world box. Then at the first level the distance from the ray origin to child box is C, the maximum parent-child distance given the maximum scale change S. Updating the traversal point leaves a gap, accounted for in the next level by adding it to that level's maximum parent-child distance, and so on down the hierarchy. This gives:

$$t_{Near,1} = fC \qquad t_{Near,2} = f(fC + C/S) \qquad \ldots$$

Since the box length shrinks by S every step it is convenient to normalize to the box length at each level. This gives:

$$t_{Near,1} = fC \qquad t_{Near,2} = f(fSC + C) \qquad \ldots$$

$$t_{Near,L} = fC\sum_{i=0}^{L}(Sf)^i$$

When $\lim_{L\to\infty} t_{Near,L} \leq 1$ then $t_{Near} \lesssim l$ for all boxes and any number of levels, which is achieved for $f(C+S) = 1$. So we see that the precision we must update the traversal point with depends on how much smaller than its parent a child box can be. For example if we allow a child to be at most 32x smaller on an edge than its parent, then we require $\gamma \geq 0.996$ which is provided with just 8 mantissa bits.

While we have reduced the necessary precision substantially the required size may have only been reduced by 6x due to the high-by-low precision multiply used in point update. As we will show in our results it is actually acceptable to use only a **single bit** of precision when updating the traversal point! This is possible because most internal nodes only reduce child edge lengths around 2x and usually in just one direction. Also, if the ray takes an unnecessary step due to a distant traversal point, it will abort the path after a few steps, when the point approaches the incorrectly hit box. This is guaranteed because the child box being approached is within the incorrectly taken ancestor. Once the traversal point is close enough to determine that the ancestor should not have been hit, it will also be close enough to detect a miss for all child boxes.

Using single bit updates removes all multiplies from traversal point update replacing them with exponent adjustments, small integer adds. Because this is very small compared to even 8 bit updates we use single bit updates in all our simulations.

So far in the analysis we have been concerned with the size of multiplies, but at this point addition deserves some attention. Floating point adds and subtracts are large due to their need for alignment circuitry. In our case the subtracts all generate inputs to low precision multiplies. This allows us to reduce the precision of the subtracts such that the maximum shift is reduced, thereby greatly reducing the size of the alignment circuit. The adds in $P' = P + t'_N D$ could accumulate error over many steps so this solution does not apply here. Using 3 single precision adders in our traversal unit requires approximately the area of a single precision multiplier [LSL*13a]. With 4 traversal units in a single GPU core this comes to less than 6% of preexisting ALU area. It is possible to significantly reduce the precision of the final adds as

well, but the analysis is more involved and beyond the scope of this paper.

Now that we have a small traversal point update procedure we can usefully deploy a conservative, reduced precision, ray-box intersection test. When we move the traversal point we must also update the active depth range $[t_{min}, t_{max}]$ of the ray. This requires subtracting t'_{Near} from each value. Because t'_{Near} has only one set bit this add is also small.

An important benefit of using low precision multiplies is that the divides can also be made low precision. In our architecture divides are replaced with 5 bit lookup tables, reducing their area and latency expenditure and removing the need to share divide resources between traversal units.

All together we are able to adjust traversal points and intersect boxes with only exponent adjustments (8 bit adds), adds, and low precision multiplies (we use 5 bits in our simulations). This reduces the area and power spent on ALU resources for traversal by as much as 23x, making possible the inclusion of very high throughput fixed function ray traversal units without a large increase in ALU area or power consumption. What remains is to ensure that the other on chip resources needed for traversal, such as ray storage and scheduling needs, do not amount to an infeasible area.

3.2 Fixed Function Traversal Unit

Our fixed function traversal unit is fully pipelined, implements single bit ray origin updates, and uses 5-bit precision for ray-box tests. A single traversal unit performs two ray-box intersections in parallel, allowing one ray-node traversal per clock. Each box intersection subunit implements the pseudocode in Figure 2. The final add for traversal point update (line 7) is repeated at full precision once the near box has been identified. Like [KSS*13] we use a single bit record for the traversal stack. The precise organization of the unit does not impact our analysis. As long as the unit's latency is less than L1 hit latency and throughput of one ray-node intersection per clock is achieved then any organization is acceptable.

Each unit contains a small register file and shares a node cache with the other units in its core. Both are sized to hide the latency of a cache hit, rather than a cache miss. Cache hit latencies vary widely between current GPU architectures indicating that wide variation, and therefore some selection, of cache hit latency is possible. We assume a hit latency between 50 and 100 cycles, requiring 50-100 active rays in each traversal unit at any time. These rays are traversed until they hit, terminate, or cache-miss, then are re-queued in the GPU register file.

The register file and node cache reduce pressure on the GPUs SIMD register file and L1 while remaining small due to the relatively low latency they need to hide. We provide a separate mechanism to hide cache misses which we will discuss in Section 5.2.

Reduced Precision Ray-AABBox Test

	BoxTest	**Precision**
1:	$//BoxNear \leftarrow$ Box Near Coordinates	
2:	$//BoxFar \leftarrow$ Box Far Coordinates	
3:	$invDir \leftarrow Inv(Dir)$	5-bit LUT
4:	$\mathbf{T} \leftarrow (\mathbf{BoxNear} - \mathbf{Point}) * \mathbf{invDir}$	1-bit Sub, 1bit Mul
5:	$\mathbf{Tclose} \leftarrow \mathbf{Max}(\mathbf{T.x}, \mathbf{T.y}, \mathbf{T.z})$	1 set bit
6:	$\mathbf{Offset} \leftarrow \mathbf{Dir} * \mathbf{Tclose}$	8-bit int add (exp. adjust)
7:	$\mathbf{NewPoint} \leftarrow \mathbf{NewPoint} + \mathbf{Offset}$	5-bit add
8:	$\mathbf{tMin} \leftarrow \mathbf{tMin} - \mathbf{Tclose}$	Sub with 1 set bit
9:	$\mathbf{tMax} \leftarrow \mathbf{tMax} - \mathbf{Tclose}$	Sub with 1 set bit
10:	$TNear \leftarrow (BoxNear - NewPoint) * invDir$	5-bit Sub, 5-bit Mul
11:	$TFar \leftarrow (BoxFar - NewPoint) * invDir$	5-bit Sub, 5-bit Mul
12:	$Near \leftarrow Max(TNear.x, TNear.y, TNear.z, tMin)$	
13:	$Far \leftarrow Min(TFar.x, TFar.y, TFar.z, tMax)$	
14:	**if** $Near > Far + 2ULP(Far)$ **then**	Far has 5-bit mantissa
15:	**return** MISS	
16:	**return {NewPoint, tMin, tMax, Near+Tclose}**	Add with 1 set bit

Figure 2: The reduced precision Ray-AABBox test with new operations in bold. The precision necessary for each operation is listed to the right. Lines 4-7 update the traversal point with a 1-bit estimate of the distance to the box. Lines 8-9 update the ray's active region. Lines 10-13 compute the standard ray-box test but with 5-bit precision. On a hit the new traversal point, active T-range, and distance to the box are returned. The distance to the box is given from the input traversal point, not the new traversal point. This allows determining which of the two boxes in a BVH node is the near box. The adds in line 7 should be full precision adds to prevent the traversal point from accumulating error. However they can be performed at low precision for the purpose of determining hit or miss, if they are repeated at full precision once the near box has been found. Doing so reduces the number of full precision adders by half.

3.3 Node Compression

To aid in reducing bandwidth we use a compressed BVH node format very similar to [Mah05]. We store short indexes into a local coordinate frame rather than global coordinates yielding considerable storage and potential bandwidth savings. The major difference with [Mah05] is that we do not use the parent box edge length directly to quantize child boxes. Rather we use the least power of two not less than the parent edge length (the $AlignUp$ of the edge length). This change quantizes a box which may be larger than the parent box but permits implementation using only shift (or exponent adds) and add. A secondary benefit of quantization is that it provides an enforcement of the maximum reduction in child edge length desired when using reduced precision traversal.

Boxes are quantized to b-bits in each axis. They are defined as $C_0 = P_0 + SI_0$ and $C_1 = P_0 + S(I_1 + 1)$ for each axis, where $C_{0/1}$ is the child box's low/high point, P is the parent box coordinate, S is the parent box scale given by $S = AlignUp(P_1 - P_0)/2^b$ which is itself a power of two, I is the b-bit index of the child point and b is the number of bits used for indexing.

Using the $AlignUp$ of the box width for scaling does not

always permit all 2^b indexes to be valid child indexes since some may be outside the box. However it removes the need for divides or multiplies when traversing the tree, which are the major sources of error in the algorithm as well as the only expensive operations. We further require non-zero length for all box edges when using this encoding, because parent edge lengths can not be directly recovered during upward traversal through such a box.

Nodes using this format are not randomly accessible and decompression order must form a continuous path through the tree. To enable quickly resuming queued rays each treelet's uncompressed bounding box is stored in a separate table and is used to re-initialize treelets when they are scheduled.

We use 5 bits for coordinate indexes and 16 bit relative indexing for child and parent pointers. This gives a node size of 12 bytes. We note that this is close to the node size used in [KKW*13] but to the best of our knowledge the details of their format are not published. When compared with a typical BVH node size of 64 bytes this gives a 5.3x storage savings. This reduction amplifies the effect of the small caches used in GPUs and results in significant bandwidth savings.

4 GPU Integration

A guiding principle of our architecture is to seek silicon reuse whenever possible. In a co-processor design with a shared memory system we suspect that external bandwidth will prevent both rasterization and ray tracing from operating concurrently at full capacity. This turns our attention to the otherwise idle resources present in a GPU and leads to our integrated approach.

Prior work has shown that it is important to maintain scheduling data and as many rays as possible on chip [KSS*13, AK10, LSL*13a]. This presents a challenge since GPUs typically have low amounts of addressable on chip storage. Fortunately high latency GPUs provide substantial storage with high bandwidth in the form of the register file. Further because of the high number of threads on GPUs and the need to reconfigure register allocations for different programs, the register file is not tightly coupled to the pipeline [AMD13b, AMD13a]. Instead the GPU operates the register file as a scratchpad with restricted addressing. The recent addition of indexable register file instructions demonstrates this flexibility [AMD13b]. In our system we reuse the register file for ray storage. Storage of rays accounts for the largest on chip storage need in our system as well as in recent prior work [KSS*13].

We also need to store queue scheduling data on chip. GPUs again present us with reusable resources in the form of shared local memory. Shared memory is a relatively low latency, high bandwidth, random access memory, with remote logic capabilities [AMD13b, AMD13a]. By extending the logic capability we can reuse this memory for scheduling

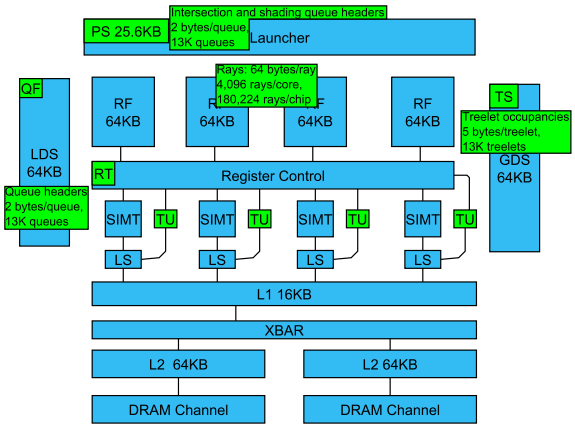

Figure 3: GPU block diagram with our modifications highlighted in green. Only 1 core and 2 memory channels shown.

and take advantage of the high bandwidth interfaces already in place.

Our system uses the programmable GPU cores for shading, intersection, and ray generation, in common with prior work. However, because we are already integrating into a GPU we do not need to incur SOC overheads to accomplish this. Since rays are already stored in the GPU's register file we need only ensure that the ray is addressable by software and aligned properly for SIMT use. This likely requires changes to the register index decoding logic but we expect that this change will seem minor when compared to the complexity of a co-processor design.

Altogether integration with the GPU has provided us with a large ray capacity, scheduling management space, light weight access to shading resources, access to data cache, and a bandwidth optimized memory controller, all with a minimum of additional area. Combined with our compact traversal unit and node decompressor this forms the primary elements of our architecture.

4.1 Example Integration Platform

While we believe that our architecture would be suitable for integration to a range GPUs, it will aid the analysis to select a concrete example system. We choose to use a hypothetical GPU similar to AMD's Hawaii (R9-290X) as the example platform for this paper. Hawaii's architecture provides several features which we use to simplify integration. Register file access width is convenient for our ray format and shared local memory is implemented with dedicated scratchpads. This alleviates concerns about sharing memory bandwidth between scene data and scheduling. Additionally a global scratchpad is available which we use to obtain a global view of traversal scheduling. A more detailed description of Hawaii's architecture can be found in [Mic13, AMD13b, AMD13a]. As a guide to our system we summarize the most important features and how we use them below (also see Fig. 3).

- Hawaii is organized into 44 cores, each with 4 16-wide SIMT units. It executes instructions with a logical width of 64 SIMT threads and has an ALU latency of 4 clocks for most single precision floating point instructions. We use the programmable cores for ray-polygon intersection and shading.
- The register file (RF) is composed of a large number of independent, high bandwidth, indexable memories. Taken as a whole, the register file provides 11MB of storage, allows 176 different simultaneous accesses (4 segments per core) and provides a peak bandwidth of 33TB/s. We utilize this for storage of queued rays and to pass data between pipeline stages.
- Each core contains a 64KB local scratchpad (LDS) shared by the 4 SIMT units in the core. This scratchpad contains 32 banks, has remote atomic capability on each bank, and can load or store 4 bytes on each bank every clock. In our system the LDS provides queue head pointers for the core's local region of the register file and filters treelet selection messages (QF). We also use it to store polygon data during intersection and shading.
- All the cores on Hawaii share a single global scratchpad, the GDS, which is also 64KB and has 32 banks. The GDS contains a more sophisticated remote logic unit than the LDS, which we extend to provide global traversal treelet selection (TS).
- Each core has a 16KB L1 data cache. Each cache can serve 4 16-byte loads per clock. With our compressed BVH format each L1 can provide upto 4 nodes per clock to traversal units within the core. Also a 1MB L2 data cache is shared by all cores. In our system both cache controllers are modified to add a *hit-only* load type.
- Over all the cores is a kernel dispatch complex (the launcher). We extend this complex to include the ability to queue rays for traversal, intersection and shading (PS).
- In our system we introduce a register transpose unit (RT) to each core's register file. Our traversal units store a single ray into a single SIMD register, placing a different component into each lane. This allows the entire ray to be retrieved in one register fetch, enabling high performance MIMD access to the GPU's SIMD register file. The transpose unit allows efficient conversion of rays to SIMD layout, needed before software can reasonably operate on them.
- Finally we add 4 traversal units (TU) to each core. Each unit is accompanied by a small register file holding just enough rays to hide cache hit latency. The 4 units in a core share a small node cache.

With this example platform we have a peak throughput of 176 billion traversal steps per second. Keeping in mind that on GPUs all vector instructions consume FPU issue slots, the traversal units provide the equivalent of 4.4 TFLOPs additional performance, yet occupy only 4-8% of current ALU die area.

5 Scheduling

Our system employs two global schedulers. The first is a high level scheduler which balances the ray tracing pipeline by managing when rays and programmable kernels launch. The second scheduler focuses on cache utilization during traverse and, like prior work, manages queues of rays which are traversing the BVH. Both schedulers work entirely from on chip storage.

5.1 Pipeline Scheduler

The primary purpose of the pipeline scheduler is to maintain high ray occupancy by managing the pipeline between traversal, intersection, and shading. Recent GPUs have thread launchers within reach of managing software pipelines on chip [Mic13, Mij12] and we extend this capability to include management of queues and traversal units.

This scheduler is queue based, operating similarly to that described in [LKA13]. However, to avoid off chip traffic, all queues are maintained on chip in the register file. Each queue is formed as a dynamically sized linked list of rays with one or two rays in each register (see Table 1). Because the register file is a limited resource and some operations require allocating from it, such as ray generation during shading, it is possible to reach deadlock conditions. We avoid this by compacting rays when rays queue for shading (see Table 1) and maintaining a small reservation sufficient for shading to make progress. Unused register space in excess of this reservation, and any other reservations which have been setup, are eagerly filled with new first generation rays. This ensures progress and high ray occupancy without needing off chip storage or complex logic.

For maximum efficiency during traversal a single ray is stored in a single SIMD register. However, SIMT software can not reasonably operate on data presented in this format. The register transpose units are used by the scheduler to marshal rays between these layouts. Having rays in MIMD format, one ray to a register, removes the difficulty of divergent queue destinations, such as occurs after intersection. Register transpose units ease this queuing operation as well.

Because the launcher is a chip global resource it may, on occasion, move data between cores. While this should in general be minimized, it is possible with at least L2 bandwidth. Other communication paths may also be available with differing characteristics, such as through the global scratchpad.

To ensure access to intersection threads promptly we reserve 8 waves on each core for intersection. This is is sufficient to mask the latency of the ALU pipeline for the operations we need [AMD13a]. This is not sufficient to hide main memory latency however, so the scheduler must prefetch polygon data into the local scratchpad. Based on measurements from the scenes we tested we expect to use less than a quarter of the total local scratchpad capacity for polygons.

Traversal	Dir_x	Dir_y	Dir_z	UV	ResultPtr	UserValue	Flags (incl. linked list ptr)	T_{Min}	T_{Max}	stack	X	Y	Z	$Orig_x$	$Orig_y$	$Orig_z$
Shading	Dir_x	Dir_y	Dir_z	UV	ResultPtr	UserValue	Flags (incl. linked list ptr)	Length	Space for another shading ray							

Table 1: Ray format used during traversal and shading. Each column represents a 4 byte lane of a single 16 lane GPU register. The Flags entry contains bits denoting color or shadow ray (1-bit), output operation (3-bits), depth in the current treelet (5-bits), additional destination bits (5-bits), and a register index used as a linked list pointer (18-bits, sufficient to address every register on the chip).

We also expect to need roughly 10% of the ray capacity processing intersections concurrently at any time. In total we reserve 20% of the ray capacity for intersections which provides buffering space.

This arrangement yields the benefits of queue based shader scheduling, yet generates no overhead traffic since all structures remain entirely on chip. Use of the register transpose units simplifies divergent queuing and provides a lightweight path between SIMD and MIMD hardware units.

5.2 Traversal Scheduler

To minimize traversal traffic we use a variant of treelet scheduling with small treelets. Queue traffic can be larger than scene traffic when using small treelets [AK10] which stresses the need to keep all rays and scheduling traffic on chip. As mentioned previously, ray queues are formed as linked lists in the register file and during traversal are managed by the traversal units within each core. We use the global and local scratchpads for the remaining queue data: global treelet occupancies and queue head pointers (register indexes) respectively. This keeps all scheduling traffic on chip and away from off chip data paths, such as cache.

Unlike the launcher the traversal units are not global so each core must be capable of holding a ray queue for each treelet. Each core's traversal queues are held in the portion of the register file within the traversal unit's host core. This admits a simple queuing operation, when a ray queues it can simply be assigned to a different queue within the same core. While this is simple and fast it may require unnecessarily high L2 output bandwidth due to multiple cores fetching the same lines from L2 to L1. One mitigating factor is that a crossbar is used in the L1-L2 interface. In general crossbars can be used for efficient broadcast. This can be used to load scheduled treelets into the L1s of the cores needing to traverse them, handling the obvious major source of duplicated line loading. We revisit this issue in Section 9.

Our traversal units make use of a new load type, *hit-only*. On a hit, loads with this type are processed exactly as an ordinary load. On a miss no data is retrieved, the cache signals that the data is not available, and the load is aborted. This type allows the traveral units to use the cache for deferrable speculative loads without risking cache pollution. We want small treelets to improve our certainty that a cache line will see reuse. We also want rays to travel as far as possible before re-queuing. The *hit-only* load type gives us this flexibility, knowing that rays cannot evict the treelet we scheduled nor generate unexpected off chip traffic. This load type is used for all node fetches issued from the traversal units.

Because the traversal units can not generate off chip traffic, the scheduler must load treelets into cache. In our system this is done by the logic unit attached to the global scratchpad, where treelet occupancies are collected and binned. Once a treelet is selected from the max bin its nodes and root box are loaded into the cache. On completion of the loads the treelet selection is broadcast to all cores which begin traversal of the associated ray queues. We extend the logic unit of the local scratchpads to receive and filter treelet selection messages. By examining the head pointer of the selected queue the unit may quickly and asynchronously determine if the core needs to process the queue. Because the traversal units begin traversal of the treelet nearly simultaneously off chip bandwidth is not impacted by the number of cores which consume the treelet.

As mentioned treelets are binned at the global scratchpad. The global scratchpad contains an array of the total number of rays queued to each treelet, or occupancy, and a list of treelets partially sorted by occupancy. When a ray is queued the traversal unit sends an occupancy increment message, via the existing remote atomic pathway, to the global scratchpad. The scratchpad performs a saturating increment on the treelet's occupancy. If the occupancy crosses a binning threshold then the treelet's position in the sorted list is updated, an exchange of two elements. To aid in locating the treelet in the sorted list another array holds the position of each treelet in the sorted list. The partial sorting allows most occupancy updates to terminate after the atomic add. This reduces list update frequency, reducing contention during occupancy updates. This partial sorting effectively bins treelets by occupancy without requiring statically allocated bins. In our experiments treelets typically have 100 or fewer rays queued to them, though some have many more. We use 1 byte for queue occupancy. In total the global scratchpad must hold 5 bytes for each treelet, 1 byte for occupancy and two 2 byte indexes, along with pointers to the start of each bin within the list.

With this arrangement treelet selection requires simply reading the leading element of the list. Overall the list is maintained as a ring allowing simple maintenance during selection. When selected the occupancy of the leading treelet is reduced to zero and the pointer to the start of the max bin is incremented.

Because we use at most the entire global scratchpad for

treelet data, we can support only a limited number of treelets. With a 64KB scratchpad we can support 13,107 treelets. The treelet count sets the number of queue headers stored in the local scratchpads, requiring under 40% of the scratchpad for 13K queue headers (12-bits are needed to fully address the core's registers). This indicates that there is space for both queue headers and polygon data in the local scratchpads.

With small treelets and centralized scheduling its possible for queue selection rate to become a performance issue. From simulation we observe that the number of steps a ray takes once activated is strongly clustered between 8-15 steps. Given the maximum available ray occupancy and treelet count we will have 10 rays/treelet in the worst case. This requires selecting just two queues per clock.

5.3 Scheduling Limits

Our architecture keeps all scheduling data on chip. This allows us to use small treelets which saves bandwidth but also imposes resource limits which constrain the size of the BVH. If the queues we select are too large then cache could become oversubscribed, causing some of the treelets to not be fully resident. Rays in these queues could encounter missing lines and re-queue to the same treelet, effectively disabling the queue and lowering the traversal rate. In the extreme if we use too few treelets then a single treelet may be larger than the cache preventing traversal from being possible at all.

For the example platform these considerations limit the largest practical BVH to 5M nodes. Beyond this performance is expected to fall off. The limitation is on the BVH, not the polygon count, but the two are closely related. We expect scenes above 10-20M polygons to result in trees which are too large for our hardware to handle well. Raising this limit requires growing the size of the caches to support larger treelets, or growing the scratchpads and register file to support more treelets.

The maximum number of queues is limited by hardware but queue partitioning is a function of BVH build software. This provides an important performance control for software. Using the maximum number of queues possible could result in loss of performance due to queue restart overheads. Using too few loses coherence during traversal and intersection, costing bandwidth. Further as noted in [AK10] the shape of the treelet impacts its ray collection rate and therefore the anisotropy of the ray distribution among the queues.

6 Polygon Data Format

We employ a packed, indexed primitive format for polygon data, similar to [SE10]. Each leaf contains a separate vertex and index buffer. Indexes use a small number of bits, typically to 3-8 bits, just enough to index the largest leaf and is constant for the BVH to retain simple software. We observe vertex to polygon ratios of 1.0 to 1.6 with this format, saving 2-3x on polygon bandwidth, which dominates in our system.

Scene	Baseline	Compress	-T.P.	8-bit	1-bit	Overhead
Hairball	52.5	55.6	-	58.1	58.3	11.1%
Vegetation	58.2	62.8	-	64.6	65.6	12.7%
Powerplant	51.4	61.7	-	61.5	65.2	26.9%
Crown	41.2	43.9	1560	45.3	46.0	11.8%

Table 2: Average number of traversal steps per ray needed in our test scenes with varying configurations. All tests use the same set of rays. Baseline is the reference software ray tracer. Compress shows the increase due to BVH quantization, which enables compression. -T.P. shows the dramatic cost of attempting to use reduced precision intersection without the traversal point update step. 8-bit and 1-bit show using 8 and 1 bits respectively in the traversal point update step and 5 bit box tests. In the final column we can see that using 1-bit traversal point updates usually results in a reasonable additional algorithmic cost over full precision. Intersection counts show similar trends and magnitudes.

7 Simulation

To estimate the performance of our architecture we implemented two simulators. Our functional simulator is a software ray tracer which applies reduced precision BVH traversal and BVH compression. This allows us to measure the algorithmic cost of our changes separately and together. The functional simulator also generates a log containing every ray traced, the nodes it visits, and which rays it creates during shading. This log feeds our traffic simulator, which we use to estimate off-chip bandwidth and queue performance metrics.

7.1 Functional Simulator

Our functional simulator is based on the path tracer packaged with Embree [WWB*14]. We extended Embree to support the BVH constraints required for node compression and fit it with a reduced precision BVH traversal routine. Node compression is simulated by enforcing minimum child edge length constraints and quantization on an existing BVH. We use the BVH build algorithms contained in Embree to generate the initial BVH without any modifications. Reduced precision traversal is implemented by adding traversal point updates and truncating mantissa bits before and after each arithmetic operation in traversal.

Table 2 shows the average number of traversal steps and leaf intersections per ray required for the test scenes in several configurations. Reduced precision traversal and BVH quantization both increase the effective surface area of a BVH node. This leads to increased traversal and intersection costs. In all but one test scene this increase is a reasonable 10-15%. The Powerplant scene stands out with a large increase in cost which we do not yet understand. In all cases BVH quantization, which enables compression, impacts costs more than using reduced precision. Because the cost of BVH quantization is set when the BVH is constructed, it seems likely that BVH build algorithms could produce higher quality trees for this system if they were

aware of its quantization limits. Also in most cases using 1 bit, rather than the theoretical limit of 8 bits, for origin updates adds only a very small amount of work, as shown in the table.

Failing to update the traversal point prevents attaining certainty that a ray does not enter a box. This results in a huge increase in traversal and intersection operations as shown in Table 2.

The BVH must be partitioned into treelets for use in the traffic simulation. We partition the tree using a simple greedy bottom-up scheme. We select the largest, deepest treelet not larger than the estimated treelet size, then recur. If after picking all the treelets we exceed the maximum treelet count we increase the estimated treelet size and select new treelets. This procedure does not take into account the geometric properties of the treelets, which may have a significant impact on performance. As with BVH construction we suspect that gains could be made through improved treelet selection algorithms.

7.2 Traffic Simulator

The traffic simulator is focused on the critical components of our system's performance: DRAM traffic, number of traversal steps per ray activation, and queue selection rate. We simulate a single level of cache, the register file, queue management, traversal traffic, intersection traffic, and a limited form of shading.

Our simulator is not sophisticated enough to run GPU shader instructions so shading is limited to creation of secondary rays. This causes us to omit traffic generated during shading, such as texturing. Also it will be necessary to reserve additional registers for use in shading kernels, further reducing ray capacity. Both of these concerns will lower the performance estimate of the system somewhat. However we do not expect shading to generate large quantities of traffic for uses other than texturing. Advanced lighting techniques are expected to cast rays in our system, rather than consume complex data structures via software as is often the case in rasterization [CNS*11].

To obtain a sufficiently tractable simulator we do not attempt cycle accurate simulation. Instead we make conservative simplifications aimed at preserving performance features relevant to our measurement goals. For instance rather than attempt to simulate the very complex GPU memory controller we use a zero latency DRAM but perform all DRAM fetches ∼500 clocks prior to use of the data. This pessimizes cache hit rates, and therefore may overestimate off chip traffic, since potentially useful data is evicted from cache earlier than it would actually have been. Simultaneously the newly loaded data will not be used for longer than the real memory system latency.

In our experiments we explored several different cache types, including direct mapped and LRU set associative. A

	Constant RPF (133M rays/frame)			Constant FPS (30Hz)	
Scene	BW(GB/s)	Traffic (GB)	TU Util.	MRPS	Traffic (GB)
Hairball	208	7.3	94.0%	3711	7.3
Vegetation	275	16.6	76.5%	2248	9.4
Powerplant	119	4.5	91.6%	3580	4.1
Crown	229	8.9	73.5%	3964	7.7

Table 3: Test scene performance with constant rays-per-frame, followed by constant FPS. Each frame is 1920x1080 an uses a maximum bounce-depth of 4. In each case we vary the sampling rate to ensure that either 133M rays are traced or 30Hz is achieved. In the final two columns we show ray rate and off-chip traffic when rendering at 30Hz. Average traversal unit utilization (TU Util.) is also shown. The Powerplant and Hairball scenes are limited by traversal performance, rather than bandwidth.

LRU set associative cache with a small number of ways greatly outperforms a direct mapped cache. Because rays are not allowed to bring data into the caches during traversal a conflict miss causes rays to queue. A single conflict miss containing a parent node can impact all the rays of the queue, dramatically shortening the average number of steps per ray activation. The improved ability to avoid conflict misses afforded by a set associative cache is therefore quite important. In our simulations we use a 1MB, 4-way set associative cache with LRU replacement, and a 64-byte cache line.

Although our simulator does not attempt to provide detailed timings, an approximate timeline can be constructed from simulator output. This is possible because as we have argued the system's performance is limited by two key metrics, the number of simultaneous traversal queues and the number of traversal steps the average ray takes once activated, in addition to the off chip bandwidth demand. The simulator provides this information which we use to determine the time the processor will need to resolve the traversal steps in a dispatch of queues. This timeline then allows us to determine the instantaneous bandwidth demand to a resolution of a few microseconds, several times longer than the DRAM latency. If the bandwidth demand exceeds the available platform bandwidth then we further lengthen the simulation interval to account for bandwidth availability. We use these timelines to determine peak bandwidth demand and frame times in the next section.

8 Results

We compare our results with STRaTA [KSS*13], a recent desktop-scale hardware ray tracing system. STRaTA is an architecture similar in spirit to ours, using on chip memory to manage treelet based traversal. In light of this similarity reduced precision and compression introduces surprising performance advantages. By reducing the precision of BVH nodes we are able to compress and save 5x on node bandwidth. This also allows our cache to hold 5x more nodes, which can be expected to reduce bandwidth around 2x. Using the GPU register file allows us to hold more rays on chip at once, allowing more reuse and reducing bandwidth

Crown (4.9M triangles) Hairball (2.9M triangles) Vegetation (1.1M triangles) Powerplant (12.8M triangles)

Figure 4: Our test scenes as rendered in our system at 30 FPS. Table 3 shows the ray rate achieved and bandwidth needed.

by ∼1.5-2x. Our cache modification permits using much smaller treelets. According to our simulations this provides an additional 1.2-1.5x bandwidth reduction. Using indexed primitives and reduced precision indexing allows 2-3x bandwidth savings during intersection at the cost of additional polygon setup arithmetic. Increased ray occupancy and decreased scheduling granularity also increases the number of rays simultaneously intersecting a leaf. This amortizes polygon setup costs and reduces bandwidth another 1.5-2x during intersection. From these rough estimates we should expect performance improvements on the order of 6x for intersection dominated scenes and 22x for traversal dominant.

For direct comparison we reproduce two test scenes from [KSS*13], the hairball and vegetation scenes. We use the original configuration of one point light, an image size of 1024x1024, one sample per pixel, and a maximum ray-bounce depth of 5. Limiting our frame rates to 25Hz and 20Hz (the maximum achieved rates in [KSS*13]), we consume an average of 11.9GB/s and 12.8GB/s respectively. This amounts to a reduction in bandwidth of 21x for the hairball and 20x for the vegetation scenes.

The comparative test scenes represent relatively low ray rates of at most 210M rays/second. As our system has a peak ray rate closer to 4B rays/second we present additional results each with a workload of 133M *rays/frame*, the best we could achieve during peak operation at 30Hz though we do not expect to reach this maximum. To make the scenes more interesting we use infinite length ambient occlusion (hemisphere) lighting for the vegetation and hairball scenes. The crown scene uses a hemispherical light map for lighting. Powerplant uses hemisphere lighting as well as an area light, simulating a solar disk. In each case the number of samples per pixel is adjusted until the the total number of rays per frame is as close to 133M as possible.

Table 3 shows average off chip bandwidth demand for our test scenes at 133M rays per frame. Each frame is rendered at 1920x1080 and uses a maximum bounce-depth of 4. The Powerplant and Hairball scenes are primarily limited by traversal unit performance while the remaining are primarily limited by available bandwidth.

To confirm that we are obtaining benefits from small treelets we simulated with larger treelets as well. For this we used a treelet size of 768 nodes, the same number of nodes as in [AK10]. Using large treelets increased traffic in the Crown scene to 11.0GB, an increase of 1.2x.

Comparison with [LSL*13b] is difficult because of the difference in power and resources available to desktop vs mobile processors. To make a grossly inappropriate comparison we compare efficiency in MRays/GB of off chip traffic while rendering diffuse interreflection rays. We reproduced the hardest (lowest efficiency) scene from [LSL*13b], Sebenik, in our system and rendered it at 1920x1080. Comparing measures we are 75x more efficient. However, this merely highlights the difficulty in comparison because after BVH compression the entire BVH easily fits into our cache while a large fraction of the polygons fit into LDS plus remaining cache space. This allows many rays to be generated, trace and retire without generating off chip traffic, and so is not a measure of our system's typical efficiency.

Computing efficiency for their hardest scene (Sebenik - 80K triangles) and ours (Vegetation - 1.1M triangles) gives 23.5 MRays/GB and 8.0MRays/GB respectively. This puts our system behind by a more modest 3x. Considering the order of magnitude difference in scene size and complexity (interior architecture vs. dense foliage) we take this to indicate that the two architectures likely have comparable efficiency if implemented at similar scale.

9 Potential Future Work

9.1 Improved Simulation and Ray Migration

In our system the L1-L2 interface may present a bottleneck due to duplicate line loads. Our ongoing research includes simulation of the L1-L2 interface and explicitly handles ray migration between cores. Ray migration can have a strong impact on traffic crossing the L1-L2 interface. While this research is still ongoing we note that in the two extremes rays may be migrated through the L2 at every queuing, or never migrate, and that these extremes are what prior architectures have proposed [AK10, KSS*13, LSL*13a].

9.2 Scheduling Algorithms

Our system aggressively extracts coherence from the on chip ray set. Incoherent rays are halted on chip while coherent rays are processed and retire. In all our experiments the system loses coherency while bandwidth demand increases until the system settles into a relatively steady state with lowered bandwidth efficiency. This effect is noted in [AK10] as a reason to consider batching rays rather than continuously streaming them. However, our system operates inefficiently

when ray occupancy is low, making frequent batching undesirable. Initial experiments show that the coherence of the average queue increases when a fraction of selections pick medium or low occupancy queues. We observe that selecting these sub-optimal queues increases variation in ray distribution, allowing rays to move out of incoherent positions and making space for new, coherent rays. We suspect that improved queue selection algorithms may have a significant effect on overall traversal efficiency and/or queue selection hardware requirements.

Improved scheduling algorithms may also be able to improve our system's primary bottleneck, polygon intersection bandwidth. Using our simple polygon format we currently observe 1-1.5 vertices per polygon. However, the same scenes have vertex amortization rates near 0.5 vertices per polygon when using a single index and vertex buffer. A scheduler capable of intersecting several neighboring leaves simultaneously might achieve higher vertex amortization, potentially realizing reduced intersection bandwidth. By using index and vertex buffers which span several leaves and intersecting those leaves simultaneously, off chip bandwidth for vertex loads might be shared without increasing the arithmetic cost of individual ray-leaf intersections.

Scaling to larger scenes may also benefit from improved scheduling. In our system the number of treelets is limited by on chip storage for each treelet's associated queue header. Using the system on large scenes causes the treelet size to grow, which reduces traversal efficiency. In prior systems scheduling algorithms have been used to handle scenes larger than hardware resources would allow [NFLC12, PKGH97, WDS05]. Such algorithms may be applicable to hardware accelerated ray tracers as well, enabling graceful scaling to large scenes.

9.3 Intersection Hardware

A notable difference between ours and prior work is that we perform intersections in software. While for the scenes tested here software appears to maintain pace, intersection often requires most of the programmable resources to do so, leaving little room for shading. It has become clear that at the ray rates projected fixed function intersection hardware is a practical necessity. Our continuing research in this area adopts hardware intersection to good effect.

9.4 Ray Approximation

Our system requires all rays which contribute to a local radiance integral to be cast simultaneously. This presents the opportunity to acquire dynamic ray bandwidth data. Ray bandwidth data is useful for approximating rays and we suspect ray approximation to lead to reduced bandwidth demand by reducing incoherent intersection tests. In particular when computing global illumination, many rays may be approximated without objectionable artifacts, often on the basis of estimated ray bandwidth [LALD12, EHDR11, MWR12]. Identifying approximation algorithms potentially suitable for real time application is the first step in defining hardware capable of supporting a useful range of approximations.

10 Conclusions

Our analysis of BVH traversal enables compact low precision traversal units, which we believe will enhance any BVH capable hardware ray tracing system. We show that BVH compression and polygon format play significant roles in reducing bandwidth consumption and can be simple enough for compact implementation in hardware. Our simulations show that major resources within current GPU architectures are compatible with fixed function MIMD ray traversal. This opens the way for integrating ray tracing capability within GPUs, eliminating area and performance costs of co-processor designs. Finally, we show that cache miss signals can be used alleviate the tension between small treelets and queuing costs, obtaining certainty of reuse while maintaining low queuing frequency.

With the near term arrival of hardware accelerated ray tracing in mobile platforms, we believe that wide spread adoption of real time ray tracing is now inevitable. Our work represents an area and power efficient approach to scaling ray tracing capability into higher power platforms, where very high, incoherent, ray rates are needed for ray tracing to be a practical option.

Acknowledgements

Thanks to Don Fussell, Jeff Diamond, Christian Miller, and Ingo Wald for many helpful discussions and invaluable aid refining the paper. Thanks to Samuli Laine for use of Hairball and Vegetation scenes. Crown scene courtesy of Martin Lubich (http://www.loramel.net) with modifications by Intel. Additional thanks to Daniel Kopta for providing scene configurations. We also thank the anonymous reviewers for many helpful comments.

References

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics 2010* (2010), pp. 113–122. 1, 2, 5, 7, 8, 10

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009* (2009), pp. 145–149. 2

[AMD13a] AMD: Opencl programming guide. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/documentation/amd-app-documentation/, November 2013. 5, 6

[AMD13b] AMD: Sea islands series instruction set architecture. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/documentation/related-documentation/, December 2013. 5

[BW09] BENTHIN C., WALD I.: Efficient ray traced soft shadows using multi-frusta tracing. In *Proceedings of High-Performance Graphics 2009 (HPG09)* (2009). 2

[BWW*12] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W. R.: Combining single and packet-ray tracing for arbitrary ray distributions on the intel mic architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (Sept. 2012), 1438–1448. 2

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum 30*, 7 (2011), 1921–1930. 9

[EHDR11] EGAN K., HECHT F., DURAND F., RAMAMOORTHI R.: Frequency analysis and sheared filtering for shadow light fields of complex occluders. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 9:1–9:13. 11

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (sep 2007), pp. 113–118. 2

[Ize13] IZE T.: Robust BVH ray traversal. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (July 2013), 12–27. 3

[KDK*11] KECKLER S. W., DALLY W. J., KHAILANY B., GARLAND M., GLASCO D.: Gpus and the future of parallel computing. *IEEE Micro 31*, 5 (Sept. 2011), 7–17. 2

[KKW*13] KELLER A., KARRAS T., WALD I., AILA T., LAINE S., BIKKER J., GRIBBLE C., LEE W.-J., MCCOMBE J.: Ray tracing is the future and ever will be... In *ACM SIGGRAPH 2013 Courses* (New York, NY, USA, 2013), SIGGRAPH '13, ACM, pp. 9:1–9:7. 2, 5

[KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 121–128. 1, 2, 4, 5, 9, 10

[LALD12] LEHTINEN J., AILA T., LAINE S., DURAND F.: Reconstructing the indirect light field for global illumination. *ACM Trans. Graph. 31*, 4 (2012). 11

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 137–143. 6

[LSL*13a] LEE W.-J., SHIN Y., LEE J., KIM J.-W., NAH J.-H., JUNG S., LEE S., PARK H.-S., HAN T.-D.: Sgrt: A mobile gpu architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 109–119. 2, 3, 5, 10

[LSL*13b] LEE W.-J., SHIN Y., LEE J., LEE S., RYU S., KIM J.: Real-time ray tracing on future mobile computing platform. In *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications* (New York, NY, USA, 2013), SA '13, ACM, pp. 56:1–56:5. 10

[Mah05] MAHOVSKY J. A.: *Ray Tracing with Reduced-precision Bounding Volume Hierarchies*. PhD thesis, Calgary, Alta., Canada, Canada, 2005. AAINR06958. 4

[Mic13] MICHAEL MANTOR: AMD's Radeon R9-290X, One Big dGPU. http://www.slideshare.net/DevCentralAMD/gs4152-michael-mantor, November 2013. 5, 6

[Mij12] MIJAT R.: Take gpu processing power beyond graphics with mali gpu computing. http://malideveloper.arm.com/downloads/WhitePaper_GPU_Computing_on_Mali.pdf, August 2012. 6

[MWR12] MEHTA S. U., WANG B., RAMAMOORTHI R.: Axis-aligned filtering for interactive sampled soft shadows. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 163:1–163:10. 11

[NFLC12] NAVRÃĄTIL P. A., FUSSELL D. S., LIN C., CHILDS H.: Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 61–70. 11

[NFLM07] NAVRATIL P., FUSSELL D., LIN C., MARK W.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on* (Sept 2007), pp. 95–104. 1, 2

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W.: Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug 2008), pp. 41–48. 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 101–108. 11

[RGD09] RAMANI K., GRIBBLE C. P., DAVIS A.: Streamray: A stream filtering architecture for coherent ray tracing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 325–336. 2

[SE10] SEGOVIA B., ERNST M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proceedings of Graphics Interface 2010* (Toronto, Ont., Canada, Canada, 2010), GI '10, Canadian Information Processing Society, pp. 153–160. 8

[SKBD12] SPJUT J., KOPTA D., BRUNVAND E., DAVIS A.: A mobile accelerator architecture for ray tracing. In *Workshop on SoCs, Heterogeneous Architectures and Workloads* (2012). 2

[SKKB09] SPJUT J., KENSLER A., KOPTA D., BRUNVAND E.: Trax: A multicore hardware architecture for real-time ray tracing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28*, 12 (Dec 2009), 1802–1815. 2

[SLL*13] SHIN Y., LEE W.-J., LEE J., LEE S.-H., RYU S., KIM J.: Energy efficient data transmission for ray tracing on mobile computing platform. In *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications* (New York, NY, USA, 2013), SA '13, ACM, pp. 64:1–64:5. 2

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (Feb. 1998), 1–14. 3

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: Saarcor: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), HWWS '02, Eurographics Association, pp. 27–36. 2

[WDS05] WALD I., DIETRICH A., SLUSALLEK P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM. 11

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: Rpu: A programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3 (July 2005), 434–444. 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree–A Ray Tracing Kernel Framework for Efficient CPU Ray Tracing. In *conditionally accepted at ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2014). (to appear, preprint available from the authors). 2, 8