Dual Streaming for Hardware-Accelerated Ray Tracing

Konstantin Shkurko
University of Utah

Tim Grant
University of Utah

Daniel Kopta
University of Utah

Ian Mallett
University of Utah

Cem Yuksel
University of Utah

Erik Brunvand
University of Utah

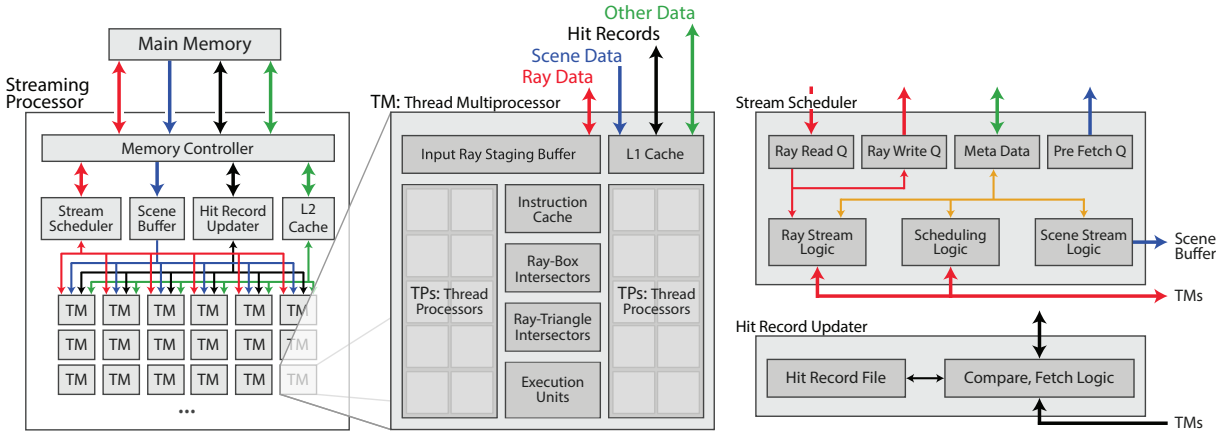

Figure 1: Overview of our dual streaming architecture. Lines connecting hardware modules indicate the flow of data colored by its type: (red) ray data, (blue) scene data, (black) hit records, and (green) other data. The streaming processor uses many Thread Multiprocessors (TMs) for computation, which share chip-wide stream units. A TM combines many lightweight hardware thread processors (TPs) that share instruction cache and computation units.

ABSTRACT

Hardware acceleration for ray tracing has been a topic of great interest in computer graphics. However, even with proposed custom hardware, the inherent irregularity in the memory access pattern of ray tracing has limited its performance, compared with rasterization on commercial GPUs. We provide a different approach to hardware-accelerated ray tracing, beginning with modifying the order of rendering operations, inspired by the streaming character of rasterization. Our *dual streaming* approach organizes the memory access of ray tracing into two predictable data streams. The predictability of these streams allows perfect prefetching and makes the memory access pattern an excellent match for the behavior of DRAM memory systems. By reformulating ray tracing as fully predictable streams of rays and of geometry we alleviate many long-standing problems of high-performance ray tracing and expose new opportunities for future research. Therefore, we also include extensive discussions of potential avenues for future research aimed at improving the performance of hardware-accelerated ray tracing using dual streaming.

e-mail:{kshkurko, tgrant, dkopta, imallett, cem, elb}@cs.utah.edu.

HPG '17, July 28-30, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5101-0/17/07…$15.00
https://doi.org/10.1145/3105762.3105771

CCS CONCEPTS

• **Computer systems organization** → **Multiple instruction, multiple data**; • **Computing methodologies** → **Graphics processors**; **Ray tracing**;

KEYWORDS

Raytracing hardware

ACM Reference format:
Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. 2017. Dual Streaming for Hardware-Accelerated Ray Tracing. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 11 pages.
https://doi.org/10.1145/3105762.3105771

1 INTRODUCTION

The two popular approaches to rendering, rasterization and ray tracing, differ in the way they process a 3D scene. While rasterization can stream the scene data (e.g. triangles), ray tracing accesses only the necessary portion of the scene using hierarchical acceleration structures. Since the theoretical complexity of ray tracing is sub-linear in scene size, it has been anticipated for decades that ray tracing would perform faster than rasterization, which has linear complexity dependence on scene size, when the regularly growing average scene size reaches a certain point. However, from the point of view of the memory system, data accesses driven by acceleration structures (used by ray tracing) look more random than those of a triangle stream (used by rasterization). Because data movement can be the primary performance limitation and energy consumer

in modern computing systems, the random access to the scene data has so far prevented ray tracing from reaching its theoretical potential. It appears unlikely that ray tracing will ever outperform rasterization without making its scene access pattern a better match for the memory system hardware.

Streamed memory accesses improve performance and energy use for at least three reasons: 1) data streaming relieves the processor of address calculation tasks and pointer chasing for traversal of tree-like data structures, 2) streaming helps hide memory latency via prefetching, and 3) the circuit-level architecture of the Dynamic Random Access Memory (DRAM) chips that make up main memory is designed for streaming. Each access to DRAM will stream contiguous data across the memory interface (typically a cache line at minimum), and internally DRAM chips provide fast and low-energy access to even larger blocks of contiguous data.

In this paper, we introduce the *dual streaming* approach for ray traversal, which reorders the traditional ray tracing algorithm to make it more suitable for hardware acceleration, considering DRAM behavior. Since dual streaming is not designed for general-purpose processors, we also describe the first custom hardware architecture implementation, outlined in Figure 1.

Our dual streaming approach organizes the memory access pattern of ray tracing into two predictable and prefetch-friendly data streams: one for scene data and one for ray data. Therefore, we pose ray tracing, for the first time, in a fully streamed formulation, reminiscent of rasterization. The *scene stream* consists of the scene geometry data (including the acceleration structure) that is split into multiple *segments*. The *ray stream* consists of all rays in flight collected as a queue per scene segment they intersect. Our scheduler *prefetches* a scene segment and its corresponding ray queue from main memory into on-chip buffers before they are needed for traversal (i.e. perfect prefetching). Hence, the compute units no longer access the main memory directly. Rays at the same depth are traced as a wavefront, so each additional bounce requires an additional pass. A predictable scene traversal order ensures that each scene segment is streamed *at most once* per wavefront. Thus, we regularize the memory traffic for scene data and reduce it to its absolute minimum.

Dual streaming provides a new ray traversal order that resolves some of the decades-old problems of high-performance ray tracing:

- *Random access to main memory during traversal is avoided.* All necessary scene and ray data are streamed on chip beforehand.
- *Scene data traffic from main memory is minimized.* Since each scene segment is streamed at most once per wavefront, memory bandwidth is not wasted by fetching the same data several times. This is particularly important for large scenes and incoherent rays (such as secondary rays).
- *Memory latency is hidden by perfect prefetching.* A traditional solution hides memory latency by adding more threads, which is effective only when the memory system can process their requests fast enough. Instead, dual streaming hides latency by perfectly predicting the next workload.
- *The memory access pattern for each stream fits how DRAM operates.* This results in extremely high row buffer hit rates, minimizing DRAM operations needed to access the requested data. Thus, DRAM chips process requests faster and at lower energy, further resulting in better bandwidth utilization.

Notice that all of these improvements relate to the memory system, since traditional ray tracing, especially for large scenes, can be bound by memory rather than compute. Data movement is also the main culprit for energy use. Therefore, all of these outcomes are critical to addressing the traditional problems with high-performance ray tracing in terms of both rendering speed and energy use.

On the other hand, dual streaming exposes new challenges and restrictions. First, to maximize scene reuse, all rays in flight are stored in and streamed from the main memory, instead of just a small number being stored (and processed) on chip. Although streaming rays from DRAM is highly efficient, fetching rays introduces additional load on main memory, which we found to be offset by the reduction in scene traffic for most scenes. Secondly, our implementation of the predictable traversal order requires some rays to be duplicated. Even though the duplication eliminates the need to store a traversal stack per ray, it still puts extra pressure on the memory system, in terms of both storage and bandwidth, and requires atomic hit record updates. Finally, unlike traditional ray tracing, implementing efficient early ray termination and optimizing the ray traversal order with dual streaming is nontrivial.

Contributions: We introduce dual streaming, which is, to our knowledge, the first ray tracing method that is completely predictive in terms of accessing both scene and ray data and allows perfect prefetching ahead of computation using two separate streams. We describe the first custom hardware architecture accelerating dual streaming and evaluate it using cycle-accurate hardware simulation to present the advantages and current limitations of our implementation. We explore only a fraction of the potential design space that could exploit the predictable traversal structure of the dual streaming approach. Yet, our results show that the initial hardware architecture we describe is already competitive with other state-of-the-art hardware acceleration for ray tracing, and has the potential to improve that performance substantially, especially for large scenes. Furthermore, we include an extensive discussion of potential avenues for future research.

2 BACKGROUND

In this section, we provide a summary of recent work on improving ray tracing memory behavior through both algorithmic improvements and custom hardware designs. There is, of course, a large body of work on enhancing other aspects of ray tracing, but we focus specifically on memory issues.

Recent work has explored a variety of ways to leverage ray tracing acceleration structures to increase scene data memory access efficiency. Some methods manipulate rays by clever generation [Purcell et al. 2002; Wald et al. 2014], sorting [Bigler et al. 2006; Boulos et al. 2007; Eisenacher et al. 2013; Moon et al. 2010; Pharr et al. 1997], or grouping [Barringer and Akenine-Möller 2014; Lee et al. 2015]. Other approaches find rays that access the same portions of the BVH by splitting the BVH into treelets [Aila and Karras 2010; Navrátil et al. 2007]. Perhaps the most similar to our work are the approaches that attempt to reorder rays or scene data into stream-like memory accesses [Bikker 2012; Gribble and Ramani 2008; Wald et al. 2007]. The distinction of our approach is to make the streams exactly predictable to allow for perfect prefetching.

One hardware approach to optimizing the memory behavior of ray tracing has been to provide a special fixed-function traversal unit as part of the processor [Kim et al. 2010, 2012; Lee et al. 2012; Nah et al. 2014; Schmittler et al. 2002, 2004; Woop et al. 2006, 2005]. While effective in terms of speeding up the issuing of memory requests, they both restrict the acceleration structures to one exact type, and also do not necessarily reorder or repackage the memory requests to take advantage of memory system organization. Other approaches such as StreamRay [Gribble and Ramani 2008; Ramani and Gribble 2009] provided a pre-filter operation to filter rays into SIMD-friendly groups for processing. These filtered sets of rays look like streams once they are assembled, but are not predictable in advance to allow for effective prefetching. Most ray tracing specific hardware architectures assume a fairly standard cache/DRAM hardware architecture and rely on the software to package ray and scene data requests efficiently [Aila and Laine 2009; Aila et al. 2012; Govindaraju et al. 2008; Kelm et al. 2009; Kopta et al. 2010; Spjut et al. 2009, 2008], with some recent work specifically targeting compression of the scene data to reduce memory bandwidth [Keely 2014; Liktor and Vaidyanathan 2016]. STRaTA [Kopta et al. 2013, 2015] is an architecture that assumes a slightly enhanced physical memory architecture by adding on-chip ray queues to the memory hierarchy. Because STRaTA stores rays just in on-chip buffers, it can process only a limited number of rays simultaneously, restricting shader complexity and treelet effectiveness. Nonetheless, because of its focus on optimizing the DRAM access for at least the scene data, we choose STRaTA as our primary comparison.

2.1 DRAM Main Memory Considerations

Main memory typically consists of large amounts of dynamic random access memory (DRAM). Although relatively high-capacity and affordable, DRAM is also notoriously complex in terms of access behavior and characteristics, and relatively slow in terms of access latency [Balasubramonian et al. 2000; Brunvand et al. 2014; Chatterjee et al. 2012; Jacob et al. 2008; Kopta et al. 2015; Wulf and McKee 1995]. In addition to being volatile, and thus needing periodic refresh operations, the internal structure of DRAM chips supports accessing data sized in cache line chunks, typically 64 bytes. Internally, every access fetches an entire *row* of data (up to 8KB) from one of the low-level memory circuit arrays into the *row buffer*. Reading a row is destructive to the DRAM contents, and the 8KB in the static row buffer (across the set of chips activated for each access) is then written back to the DRAM array in each chip to restore the data.

An important behavior of the row buffer is that because it is implemented as fast static memory, if the next 64B data access is also within that 8KB row buffer, known as an *open row* access, then that access is dramatically faster and more energy-efficient than if the access requires opening a new row. In a sense, the row buffer acts like an additional cache that lives across DRAM chips.

The memory controller is another critical piece of the DRAM memory system because it interfaces between the (typically) 64B memory requests from the processor and the complex DRAM memory. Aside from managing DRAM, the memory controller also accumulates data requests and reorders them based on which rows they map to. This improves row buffer hit-rates but introduces variability in access latency. A ray tracer carefully restructured to improve memory access patterns during traversal can help the memory controller increase the row buffer hit-rate, thus reducing both DRAM latency and energy [Kopta et al. 2015]. Looking ahead, accessing a contiguous stream of data comparable in size to a row buffer can represent a best-case use of DRAM in terms of achievable latency and power consumption.

3 DUAL STREAMING

The main goal of dual streaming is to eliminate the irregular accesses to the main memory and minimize scene data transfer by reformulating ray tracing as two separate streams: a *scene stream* and a *ray stream*.

The scene stream consists of multiple *scene segments* that collectively form the entire scene geometry data and can be processed separately. Segments are sized as a multiple of the DRAM row buffer capacity to enable efficient streaming. Although most acceleration structures could be used to generate scene segments, our implementation splits a BVH into treelets [Aila and Karras 2010], each containing both internal and leaf nodes, as well as the scene geometry (e.g. triangles).

The ray stream is the collection of all rays that are in flight, split into multiple queues, one per scene segment. The ray stream consists of basic ray information: origin, direction, and a ray index. Since the scene segments are traversed independently, there is no need to store a global traversal stack for each ray, significantly reducing the overhead for ray state.

Throughout traversal, rays are added to the queues of scene segments they need to visit, and are removed from the queues as the associated segments are processed. Because our scene segmentation is hierarchical, the ray queue for a given segment is filled when traversing its parent segment, and is drained only after the parent segment has been fully processed.

During ray traversal, both the scene segment data and its ray queue are prefetched onto the chip, thereby eliminating the need for random accesses to the main memory to fetch the required data.

The construction of scene segments dictates the fixed traversal order based on how rays can flow from one segment to the next. Tree-based segmentations (e.g. BVH treelets) impose a hierarchical relationship between segments in which rays flow from a parent to its children, but never from children to their parent. This eliminates the need to reload scene segments, since rays are not allowed to revisit segments.

The order in which segments are loaded may follow strict breadth- or depth-first ordering, or may leverage run-time statistics, like the distribution of rays among scene segments. Once all rays finish traversing through a particular segment, its children become available to be processed. If a segment's ray queue is empty when it is time to be processed, the segment (and all of its descendants) can be safely skipped, since no rays visit that part of the scene.

All segments with non-empty ray queues are processed in parallel. Moreover, rays from the same queue can be distributed between different threads. Traversal ends when the ray queues for all scene segments have been emptied. Due to the predefined traversal order, each scene segment is processed (and therefore loaded) at most once per wavefront (i.e. once per ray bounce).

Within a segment, each ray follows a typical traversal using a local stack. However, when a ray finds an exit point from the current segment into one of its children, the ray does not immediately follow the path. Instead, the ray is duplicated into the ray queue for the child segment and continues traversal *within* the current segment until all exit points are found. We apply early ray termination locally: if a ray hits a triangle within the scene segment, it avoids traversing any nodes and enqueuing into any child segments farther than the hit. Ray duplication avoids reloading scene segments per ray wavefront, because rays do not revisit the parent segment to reach a sibling segment.

We maintain a shared hit record for each set of duplicate rays, which must be updated each time an intersection is found. Since the number of rays is very large, the hit records are stored in DRAM rather than sent along with each ray. Updating the shared hit records—and ray duplication in general—presents special challenges which we discuss in later sections.

Because rays can be duplicated, early ray termination becomes non-trivial. When an intersection is found, there is no easy way to check whether this distance is the global minimum, or if a duplicate ray being traced simultaneously in another scene segment has found a closer hit. Testing which hit to keep requires atomic access and updates of the shared hit record, handled by a dedicated hardware unit discussed in Section 4.3.

Because the shared hit records are kept off-chip, early termination tests must access DRAM during traversal, potentially incoherently. Although these accesses should get coalesced, threads could stall waiting for the current hit distance to return from main memory. We consider two separate mechanisms: *pre-test* and *post-test*. The pre-test checks the hit record before traversing a ray through the current scene segment. The post-test checks the record after traversal and before enqueueing into another segment.

4 HARDWARE ARCHITECTURE

Our hardware implementation of dual streaming is shown in Figure 1. The design follows the Single Program Multiple Data (SPMD) paradigm, with independent control flow for each thread. The architecture partitions a large number of Thread Processors (TPs) into a number of Thread Multiprocessors (TMs) to allow TPs to share units that are expensive in terms of area, like fixed-function intersection units and ray staging buffers.

The complete streaming processor is built from many TMs which share access to several global units: the *stream scheduler*, the *scene buffer*, and the *hit record updater*. These units connect to the memory controller, which interfaces with off-chip DRAM. Figure 1 shows details for these global units and Table 1 describes their areas.

Unlike a more typical architecture, our dual streaming implementation features no large L2 data caches. Instead, the chip area is used for dedicated scene and ray buffers, which are essentially large static random access memory (SRAM) scratchpads. Compared to typical caches of similar capacity, such scratchpads are simpler to implement, faster to access, and consume less energy per access.

4.1 Stream Scheduler

One of the key units in our implementation is the stream scheduler, shown in Figure 1. The stream scheduler marshals the data required for ray traversal to prevent TPs from accessing main memory directly for both scene and ray stream data. The stream scheduler also tracks the current state of traversal, including the *working set* of active scene segments, the mapping of TMs to scene segments, and the status of the scene and ray streams.

As discussed in Section 3, the scene is partitioned into a number of segments which can be traversed independently. With the exception of the first scene segment (e.g. the root treelet, in the case of BVH treelets), scene segments become eligible for traversal only after their parent has been traversed. When the traversal of a scene segment is completed (i.e. its ray queue is depleted), the stream scheduler replaces the scene segment with another. After adding a scene segment to the working set, the stream scheduler transfers the corresponding data from DRAM to the scene buffer.

Tracking scene segment streams incurs little overhead per segment: starting address and the number of cache lines transferred so far. We have found that streaming eight segments simultaneously performs well, and requires modest area within the stream scheduler: 40 bytes of storage and some counters.

Rays are partitioned into a number of queues, with one queue per scene segment. While only a small subset of all active rays can fit on chip, the rest are stored in DRAM until they can be processed. Each ray queue is stored as a linked list of small pages, called *ray buckets*. Within its header, each ray bucket stores the next bucket's address and a ray counter. Although some buckets may be filled only partially, buckets maintain constant size and row buffer alignment in DRAM. In our implementation four 2KB ray buckets perfectly fit within an 8KB DRAM row buffer to leverage the streaming behavior of DRAM.

Both scene segments and ray buckets are sized cognizant of the DRAM row buffer, to make sure each stream is stored in main memory as a continuous block, as shown in Figure 2.

TPs do not read rays directly from the queues. Instead, the stream scheduler fetches entire ray buckets from DRAM and forwards them to the appropriate TM's *ray staging buffer*. Similarly, TPs write rays into queues via the stream scheduler, which maintains a small queue of such requests. The stream scheduler drains the queue by writing each ray into the appropriate ray bucket (stored in DRAM).

Since rays are written into the queues of child segments as they exit their parent, the total number of potential write destinations equals the total number of children of all segments currently in the working set. Since segments can have many children, the number of destinations can be large, and for some of our test scenes it reached about a thousand. Maintaining pointers to the ray queues for such a large number of segments—along with the metadata capturing the parent-child and sibling relationships required for queue processing and scheduling—requires approximately 16KB of SRAM.

Each TP fetches individual rays to be processed from one of the ray staging buffers within its TM. There is no deterministic mapping between TPs in a TM and rays within the staging buffer. The staging buffer is sized to store exactly two ray buckets and is split into two halves: while one is being drained by TPs, the stream scheduler fills the other with another ray bucket from DRAM. Each ray stores the address of the node it is traversing. The TPs use this address to load scene data from L1, which accesses the scene buffer on misses. The scene data is fed into TM-wide intersection pipelines for traversal.

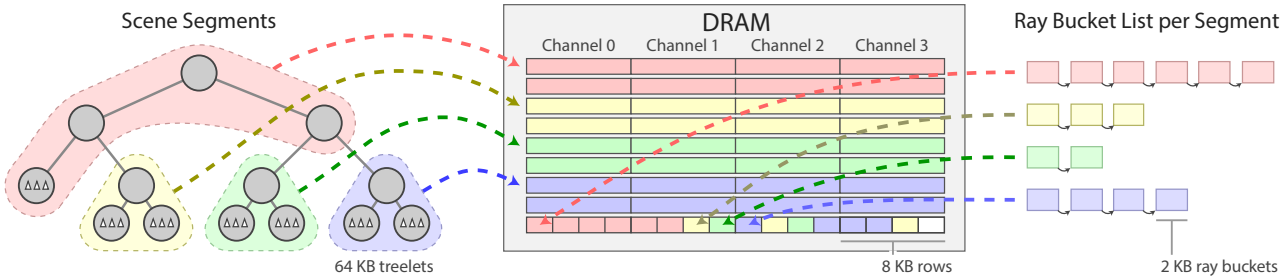

Figure 2: Layout of both streams in DRAM. Each scene segment (left) is placed contiguously in DRAM, including triangles. Ray stream (right) consists of a linked list of buckets, which may be stored in a fragmented fashion. Note that the stream scheduler stores pointers to scene segments and the first (head) ray bucket, depicted by dashed arrows.

After one half of a ray staging buffer runs out of rays, it is swapped for the other half. The stream scheduler polls which staging buffers are empty, and attempts to find another ray bucket from the same scene segment (to improve L1 cache reuse). If there are no more buckets for the currently processed scene segment, the scheduler attempts to find a ray bucket for a scene segment in the working set. If there are no more ray buckets left for any of the scene segments in the working set, the scheduler must wait for the traversal of a scene segment to complete, before evicting it from the working set and replacing it with another scene segment. To select the next scene segment for inclusion in the working set, the stream scheduler maintains a queue of segment IDs to be processed, ordered by depth-first segment traversal.

4.2 Scene Buffer

The scene buffer is a global, on-chip memory that holds the scene data for all scene segments currently in the working set. The scene buffer hides access latency much like the last-level caches in more traditional architectures, but operates simpler. Unlike a traditional cache, the scene buffer is read-only and is managed entirely by the stream scheduler, rather than responding to individual memory access requests.TPs access scene segments only through the scene buffer, which eliminates random access to DRAM for scene data. Recently accessed scene data is retained in the L1 cache of each TM, which reduces contention for the global scene buffer.

4.3 Hit Record Updater

The hit record updater is a unit that atomically updates the hit records stored in DRAM. Duplicated rays share a common hit record, requiring atomic updates whenever an intersection is found. When a TP finds a hit, it sends an update request to the hit record updater, which maintains a small on-chip queue of requests. If there is an update pending for the given ray index (i.e. duplicate ray found another hit), the closer of the two hits replaces the update; otherwise, the request is added to the queue.The hit record updater compares the hit distances to the values recorded in the DRAM and, updates hit information in DRAM only if the pending hit is closer. As long as the hit record updater queue is not full, TP execution is not blocked. Despite our initial apprehension, we have not found access to the hit record updater to be a bottleneck.

4.4 Comparison to STRaTA

Like STRaTA [Kopta et al. 2013, 2015] our hardware implementation uses the SPMD paradigm, utilizes fixed-function pipelines, and relies on treelets as a means of accessing scene data. However, beyond these similarities in basic components, which would be present in any parallel hardware, our dual streaming hardware design bears little resemblance to STRaTA. First of all, dual streaming fundamentally alters how data flows through the processor and how it is accessed: perfect prefetching for dual streaming vs on-demand cache loads for STRaTA. The traversal algorithm each architecture implements is also fundamentally different: dual streaming processes all rays within a given treelet before moving on to the next in a fixed order, whereas STRaTA processes treelets in an unknown order, on-demand as they fill with rays, and often some rays re-entering treelets (thereby re-fetching the same scene data from DRAM). Another fundamental difference is that dual streaming traces wavefronts starting with the primary rays followed by secondary rays in separate passes, organized into ray streams that reside in DRAM. This enables many more rays in flight simultaneously, and combined with the deterministic traversal order, ensures that any treelet is loaded at most once per pass. Even if STRaTA was capable of spilling rays from on-chip queues into DRAM to allow more rays in flight, its traditional depth-first traversal would force it to (potentially) reload treelets many times. In comparison to STRaTA, dual streaming uses a completely different scheduler with different tasks, it contains a read-only scene buffer instead of a large L2 cache, contains a dedicated hit record updater unit, and relies on an input ray staging buffer per TM. In other words, any similarity between the two architectures is limited to the design of individual TPs and TMs and the fact that treelets are used to split the scene data.

5 RESULTS

We use a cycle-accurate simulator to evaluate our dual streaming architecture and we compare our results to STRaTA [Kopta et al. 2015], a state-of-the-art ray tracing specific architecture. The choice of STRaTA for direct comparison is motivated by the fact that it also aims to optimize DRAM accesses (though using a traditional ray tracing paradigm) and thus we can design fair comparisons by simulating similar hardware parameters. We also provide limited comparisons against NVIDIA's OptiX GPU ray tracer [Parker et al.

Table 1: Hardware Configuration

	Dual Streaming	STRaTA
Common System Parameters		
Technology Node	65nm CMOS	
Clock Rate	1GHz	
DRAM Memory	4GB GDDR5	
Total Threads	2048	
On-Chip Memory		
L2 Cache	512KB, 32 banks	
On-Chip Ray Queues	*N.A.*	4MB
Scene Buffer	4MB	*N.A.*
TM Configuration		
TPs / TM	16	16
L1 Cache	16KB, 8 banks	32KB, 8 banks
Ray Staging Buffer	2×2KB	*N.A.*
Area (mm^2)		
Memory Controller	13.1	13.1
Scheduler	0.53	negligible
Caches / Buffers	190.4	159.7
Compute	57.1	57.1
Total	261.1	229.9

2010], and Intel's Embree CPU ray tracer [Wald et al. 2014], running on actual hardware.

In our comparisons we use no early termination for our dual streaming hardware, but we do use early termination for STRaTA, OptiX, and Embree. Therefore, our dual streaming hardware performs substantially more work without the benefits of early ray termination. Additionally, we provide results with STRaTA without early termination for comparison. We present our test results with early termination using dual streaming separately at the end of this section.

5.1 Hardware Specifications

Table 1 lists the hardware configurations for both dual streaming and STRaTA. On-chip cache and SRAM buffer areas are estimated using Cacti 6.5 [Muralimanohar et al. 2007]. Compute resource areas are estimated with synthesized versions of the circuits using Synopsys DesignWare/Design Compiler. We did not fully synthesize the logic circuitry for the memory controllers or the dual streaming scheduler. Instead, the area consumed by memory controllers is mostly dominated by its buffers and other SRAM components [Bojnordi and Ipek 2012]. The dual streaming scheduler is similar to a memory controller in terms of logic circuitry. Therefore, to make conservative area estimates for the memory controller and scheduler, we assume the area of these units is 2× the size of the SRAM components, which we model with Cacti. STRaTA's scheduler is reported as roughly zero area because its scheduling metadata is contained entirely within the ray queue, which is already accounted for. Since STRaTA has a much simpler scheduler, the additional logic circuitry would be negligible in area. We do not include area comparisons for Embree or OptiX, since STRaTA and dual streaming are imagined as accelerators, not a full-system CPU/GPU, and because the 65nm process technology we can simulate is much larger than current technologies.

In all simulated test results we present, the processor runs at 1GHz and has 4MB of on-chip memory (used differently with STRaTA and dual streaming), 128 thread multiprocessors each with 16 hardware threads (2048 threads total), each with 32 registers and 512B of local scratchpad memory. Note that this is a relatively moderate configuration compared to currently available discrete GPU hardware. Beyond these common parameters, hardware-specific parameters are specified after numerous tests for finding an optimal setup for each hardware design.

The global scene buffer for dual streaming is 4MB in size and can store at most 64 treelets, each 64KB in size. Ray buckets are 2KB in size and store up to 64 rays. The sizes of these components are chosen based on our experiments with different configurations. Our earlier tests revealed that we can achieve slightly higher performance for almost all scenes when the scene buffer size is 4MB, as compared to 2MB. However, the optimal scene buffer size depends on the number of TMs. Our tests with different ray buffer sizes provided only slightly elevated performance for most scenes with 2KB, as compared to 1KB.

For the STRaTA results we use treelets of size 32KB, which produced the best performance in our tests. The on-chip memory for STRaTA is split into a 512KB L2 cache and a 4MB ray buffer. The execution units of the original STRaTA multiprocessors dynamically reconfigure into either a ray-box or two ray-triangle intersection pipelines. For a fair comparison to dual streaming, we generated the STRaTA results using fixed-function pipelines, which have slightly elevated performance.

The pipeline intersecting rays against boxes relies on inverted ray directions [Williams et al. 2005]. Each TP uses the TM-wide shared division unit to compute this inverse immediately after fetching a ray from the staging buffer. The inverse is reused when traversing an individual scene segment. The ray-triangle intersection pipeline relies on Plücker coordinates [Shevtsov et al. 2007] to delay the division until intersection is found. Both architectures include a single ray-box (1 cycle initiation interval, 8 cycle latency) and two ray-triangle (18 cycle initiation interval, 31 cycle latency) pipelines shared by all TPs in each TM.

Our evaluation setup includes a GDDR5 DRAM subsystem with 16 32-bit channels, running at an effective clock rate of 8GHz for a total of 512 GB/s maximum bandwidth. The DRAM row buffer is 8KB wide. We rely on a sophisticated memory system simulator, USIMM [Chatterjee et al. 2012], to accurately model DRAM accesses. Note that using a full memory system simulator is essential for producing reliable results, since the ray tracing performance is tightly coupled with the highly complex behavior of DRAM.

The OptiX (v3.9) results are obtained on an NVIDIA GTX TITAN GPU with 2688 cores running at 876 MHz and 6144 MB GDDR5 memory with 288.4 GB/s bandwidth. The Embree (v2.10) results are obtained with its example path tracer (v2.3.2) running on an Intel Core i7-5960X processor with 20 MB L3 cache and 8 cores (16 threads) over-clocked to 4.6GHz.

5.2 Test Scenes

We used eight test scenes, shown in Figure 3, to represent a range of complexities and scene sizes. They are rendered using path tracing [Kajiya 1986] with five bounces, producing a highly incoherent

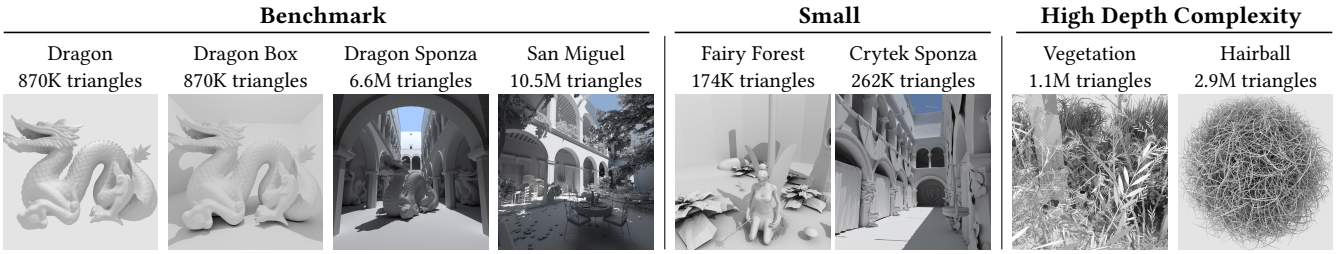

Figure 3: Scenes used for all performance tests and comparisons.

Table 2: The performance results comparing OptiX, Embree, STRaTA and our dual streaming architecture. Note $ means cache, M means millions. Values highlighted in **red** indicate the best performance for that metric.

		Benchmark				Small		High Depth	
		Dragon	Dragon Box	Dragon Sponza	San Miguel	Fairy Forest	Crytek Sponza	Vegeta-tion	Hairball
OptiX	Render Time (ms/frame)	24.50	92.97	151.83	397.47	84.05	140.11	266.92	229.79
	Rays Traced per sec (M)	135.02	112.6	65.98	24.42	78.88	72.04	26.1	27.54
Embree	Render Time (ms/frame)	38.13	103.81	118.05	143.64	83.6	150.63	178.99	113.32
	Rays Traced per sec (M)	99.54	89.08	70.62	50.48	96.08	62.04	41.66	45.28
STRaTA	Render Time (ms/frame)	*23.12*	*91.27*	*70.98*	*125.51*	**16.1**	**39.0**	**48.23**	**36.2**
	Rays Traced per sec (M)	*89.7*	*128.9*	*135.4*	*72.6*	**365.6**	**233.3**	**121.4**	**111.2**
	DRAM Energy (J)	*2.34* (55%)	*10.17* (56%)	*5.32* (46%)	*15.08* (60%)	**0.87** (32%)	**2.26** (32%)	*5.38* (52%)	*4.61* (59%)
	On-Chip Memory Energy (J)	*1.84* (43%)	*7.67* (42%)	*6.03* (52%)	*9.76* (39%)	*1.76* (65%)	*4.55* (65%)	**4.70** (46%)	**3.02** (39%)
	Compute Energy (J)	*0.08* (2%)	*0.29* (2%)	*0.21* (2%)	*0.33* (1%)	*0.09* (3%)	*0.24* (3%)	*0.23* (2%)	*0.14* (2%)
	Avg. Bandwidth (GB/s)	219.33	266.65	137.48	219.34	99.01	101.95	229.59	254.53
	$ Lines Transferred (M)	79.2	380.1	152.5	430.1	24.91	62.14	173	144
STRaTA no early termination	Render Time (ms/frame)	*47.12*	*154.08*	*117.97*	*363.16*	*21.64*	*63.56*	*115.30*	*247.75*
	Rays Traced per sec (M)	*44.0*	*76.3*	*81.5*	*25.1*	*272.2*	*143.3*	*50.8*	*16.2*
	DRAM Energy (J)	*5.63* (66%)	*21.61* (67%)	*11.49* (60%)	*48.50* (75%)	*1.36* (40%)	*4.46* (41%)	*14.48* (61%)	*35.47* (78%)
	On-Chip Memory Energy (J)	*2.72* (32%)	*9.89* (31%)	*7.31* (38%)	*15.3* (24%)	*1.97* (57%)	*6.07* (56%)	*8.78* (37%)	*9.82* (21%)
	Compute Energy (J)	*0.15* (2%)	*0.53* (2%)	*0.38* (2%)	*0.76* (1%)	*0.12* (3%)	*0.35* (3%)	*0.46* (2%)	*0.49* (1%)
	Avg. Bandwidth (GB/s)	251.00	327.41	185.16	221.77	125.52	137.07	280.86	245.41
	$ Lines Transferred (M)	184.8	788.2	341.3	1258	42.4	136.1	506.0	950.0
Dual Streaming	Render Time (ms/frame)	**18.08**	**66.3**	**40.93**	**79.61**	*17.05*	*44.6*	*68.56*	*63.27*
	Rays Traced per sec (M)	**114.8**	**177.4**	**234.8**	**114.6**	*345.6*	*204.11*	*85.4*	*63.5*
	DRAM Energy (J)	**1.15** (42%)	**4.66** (40%)	**4.47** (57%)	**8.12** (50%)	*1.61* (53%)	*4.51* (50%)	**4.41** (41%)	**4.56** (43%)
	On-Chip Memory Energy (J)	**1.52** (55%)	**6.54** (57%)	**3.10** (40%)	**7.63** (47%)	**1.30** (43%)	**4.14** (46%)	*5.96* (56%)	*5.75* (54%)
	Compute Energy (J)	*0.09* (3%)	*0.36* (3%)	*0.23* (3%)	*0.53* (3%)	*0.10* (3%)	*0.32* (4%)	*0.37* (3%)	*0.33* (3%)
	Avg. Bandwidth (GB/s)	140.21	114.98	271.30	255.61	230.35	237.40	142.75	142.88
	$ Lines Transferred (M)	39.6	119.2	173.5	317.9	58.7	165.4	152.9	141.3
	Ray Stream $ Lines (M)	11.92	45.81	43.94	146.34	18.4	80.56	94.03	76.22
	Scene Stream $ Lines (M)	7.54	8.0	54.31	72.27	1.50	2.26	8.20	19.18
	Shading $ Lines (M)	17.43	42.38	45.00	46.22	30.53	45.42	32.46	26.99
	Hit Record $ Lines (M)	2.74	22.97	31.04	53.93	8.33	37.21	18.28	18.89
	Ray Duplication	4.55	3.14	4.18	15.19	3.00	8.55	15.15	16.02

collection of secondary rays, which is both challenging for high-performance ray tracing and typical for realistic rendering. Each image is rendered at 1024 × 1024 resolution, resulting in at most 10.5 million primary and secondary rays. Dual streaming traces at most two million rays (and their duplicates) per wavefront, while STRaTA traces 80,000 rays, many potentially at different depths. We use a simple Lambertian material on all scenes, so that the results are not skewed by expensive shading operations.

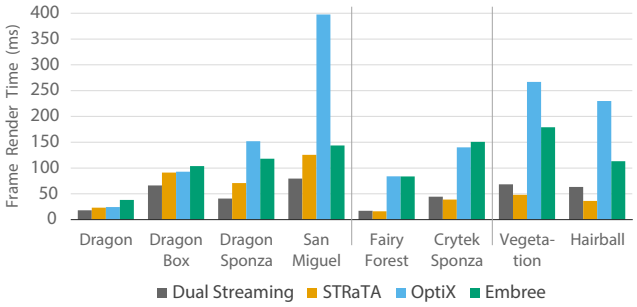

Figure 4: Render time per frame. Lower is better.

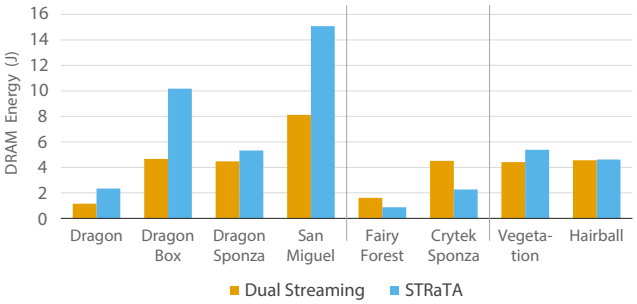

Figure 5: DRAM energy per frame. Lower is better.

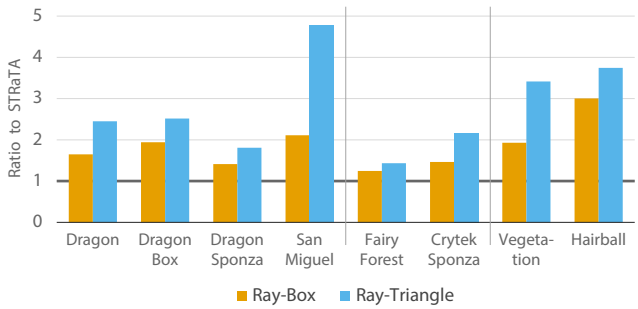

Figure 6: Number of box and triangle tests performed by our dual streaming hardware per frame. Lower is better.

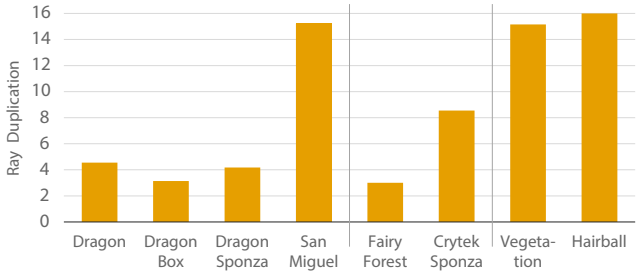

Figure 7: Dual streaming ray duplication. Lower is better.

Some scenes are chosen to present the performance of our dual streaming hardware in atypical cases that it is not designed to optimize. The first two, Fairy Forest and Crytek Sponza, are small scenes that can fit in the on-chip memory. Therefore, the improvements that dual streaming introduces for better DRAM access provide no benefits. The other two scenes, Vegetation and Hairball, are not as small, but have extreme depth complexity, where early termination, which is disabled for dual streaming, can provide tremendous savings in traversal and ray duplication.

5.3 Performance Comparisons

Table 2 provides detailed test results. The results for our dual streaming hardware and STRaTA are obtained from hardware simulations, so they include detailed information. For dual streaming, we also report a breakdown of the memory traffic and average ray duplication rates, which measure the ratio of the total number of rays enqueued into any treelet to the number of unique rays generated. The OptiX and Embree results only include render time and rays traced per second, measured on actual hardware.

Figure 4 compares the render times per frame between dual streaming, STRaTA, OptiX and Embree. Figure 5 compares DRAM energy per frame between dual streaming and STRaTA. Notice that for all benchmark scenes our dual streaming hardware provides substantially superior performance as compared to STRaTA, and the difference is more substantial in larger scenes. It achieves lower render times (up to almost twice as fast in large scenes) and consumes less DRAM energy (about half of STRaTA in some scenes).

In our small scenes, Fairy Forest and Crytek Sponza, which STRaTA can fit in the on-chip memory, our dual streaming implementation can still achieve a similar render time, but the additional burden of streaming rays costs extra DRAM energy.

On the other hand, the lack of early termination in our implementation of dual streaming hurts the render time in scenes with high depth complexity, Hairball and Vegetation. This is due to the extra work that our dual streaming implementation endures (for finding potentially all hits) and STRaTA can avoid via early termination (to find the first hit). This extra work can be clearly seen in the elevated ray-triangle intersection counts, shown in Figure 6, and the rates of ray duplication, shown in Figure 7. However, in the San Miguel scene, even though it also has substantial depth complexity causing several times more triangle intersections, the savings of dual streaming more than make up for the extra computation. This result confirms that dual streaming has the potential to provide more savings for larger scenes.

Even though dual streaming requires more intersection tests than STRaTA, all compute makes up around 3-5% of total energy spent per frame, as can be seen in Table 2. Thus, a five-fold increase in the number of intersection tests generates a tiny increase in compute energy. The remaining 95-97% of frame energy is spent by on-chip memories and DRAM, which is by far the single largest consumer.

The breakdown of the memory traffic for dual streaming and STRaTA are shown in Figure 8. Notice that the scene stream only takes up a relatively small portion, even in large scenes. Although the total traffic generated by dual streaming is smaller than that of STRaTA for all but Dragon Sponza and small scenes, dual streaming substantially reduces scene traffic for all scenes. Comparing Dragon Sponza to San Miguel, which is almost twice the size, we can see

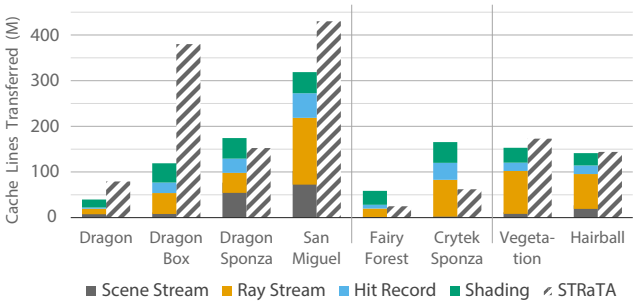

Figure 8: Memory traffic generated by dual streaming and STRaTA, in millions of cache lines (64B each). Lower is better.

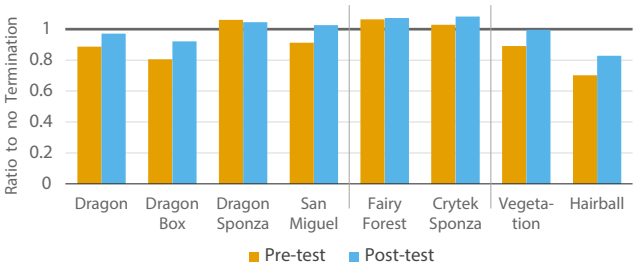

Figure 9: Effect of early ray termination on render times (ms/frame). Lower is better.

that they have similar scene stream costs (Figure 8). However, the ray stream can contribute a substantial portion of the memory traffic in all scenes. In the case of San Miguel, ray duplications not only cause extra computation but also a substantial amount of ray stream traffic and extra hit record updates, though it still renders almost twice as fast with dual streaming and consumes almost half the DRAM energy, as compared to STRaTA. Note that the magnitude of the memory traffic is related to, but does not directly correlate with the DRAM energy or performance, because they are also influenced by the order in which the memory requests are generated.

5.4 Early Termination

Here we provide our test results for evaluating the two early ray termination approaches discussed in Section 3. Figure 9 compares the render times with pre-test and post-test early termination approaches to no early termination. Notice that pre-test can provide some improvement, especially for scenes with high depth complexity. The post-test, however, is less effective in our test scenes. Our experiments also revealed that combining tests does not improve on using pre-test alone. We attribute the slightly better performance achieved using pre-test (as compared to post-test) to its ability to skip ray traversal through the current scene segment, while post-test is only helpful in preventing unnecessary ray duplication when a closer hit has already been found.

It is also important to note that in some scenes our early termination strategies can even impact the overall render time and DRAM energy negatively. This is not surprising, since early termination requires random memory accesses that can cause TPs to stall. As such, optimizing early ray termination in dual streaming is left for future work.

We also compare the effect of early ray termination on STRaTA, shown in Table 2. Disabling it can incur a significant increase in frame times of up to 3× for San Miguel and almost 7× for Hairball. The total number of cache line transfers from DRAM at least doubles. These increases are expected because STRaTA reloads scene data as rays traverse back to parent treelets. Note that compared to STRaTA without early ray termination, dual streaming has lower frame render times for all scenes. For all but small scenes, dual streaming uses less DRAM energy and the number of cache lines transferred is also significantly smaller. In depth exploration of early ray termination within dual streaming is left for future work.

Table 3: Main memory utilization measured in bytes per ray for comparison architectures (lower is better). GB/s is the total main memory bandwidth and MRPS is millions of rays per second.

Architecture	GB/s	MRPS	Bytes/Ray
[Aila and Karras 2010]	2.0 - 3.8 GB/fr.	3.3 MR/fr.	598 - 1137
[Nah et al. 2014]	0.4 - 0.6	18 - 20	20 - 33
[Liktor and Vaidyanathan 2016]	not given	155 - 335	61 - 298
STRaTA (bench)	138 - 267	73 - 135	1015 - 3021
STRaTA (small)	99 - 192	233 - 366	271 - 437
STRaTA (depth)	230 - 255	112 - 121	1891 - 2269
Dual Streaming (bench)	115 - 271	115 - 235	648 - 2230
Dual Streaming (small)	230 - 237	204 - 346	667 - 1163
Dual Streaming (depth)	143	64 - 85	1672 - 2250

6 LIMITATIONS

The main feature of the dual streaming algorithm is to refactor ray tracing into two predictable data streams where the scene data is loaded at most once per ray pass. At a very high level this is in some ways a reversal from traditional algorithms. Instead of tracing a ray to completion while loading scene data (treelets) on demand, a portion of the scene is loaded once and all rays that intersect with it are streamed through. While the memory bandwidth required for scene data is reduced significantly, dual streaming requires bandwidth for ray data (Figure 8). This, in turn, becomes an interesting challenge for the current version of the proposed hardware - how to manage the memory bandwidth required for the ray traffic?

One way to compare to other systems is by considering how much data the system fetches per ray, shown in Table 3. For example, in terms of Bytes/Ray our dual streaming implementation is within a factor of two from the seminal treelet architecture [Aila and Karras 2010]. One reason for this difference is that dual streaming can not perform early termination. This can generate a lot of extra ray traffic between treelets. For scenes with high depth complexity (Vegetation and Hairball), this problem is increased because the number of treelets a ray intersects is proportional to the number of spatially-distant leaf nodes it must visit.

We can compare against another architecture targeting mobile platforms, RayCore [Nah et al. 2014]. It benchmarks using small scenes that fit nicely into the cache and therefore traditional algorithms that keep rays on chip result in very small traffic to main

memory. While dual streaming still reduces the scene traffic significantly, small scenes show the overhead of ray streams.

While dual streaming reduces scene traffic by introducing fixed traversal order, other researchers have made good progress in compressing the BVH data and thus reducing Bytes/Ray, for example [Liktor and Vaidyanathan 2016]. This method compresses BVH layouts in a manner similar to [Yoon and Manocha 2006], but adds modifications to compress treelet-interior pointers and optimize layout for caches. A direct comparison of the Bytes/Ray metric is complicated by the different memory architecture and heavy instancing in some of their test scenes. Additionally, they use index nodes to achieve a 50% reduction in L2 to L1 bandwidth, which we do not attempt. Dual streaming is largely orthogonal to such memory optimizations. Applying a similar scene data compression technique could help reduce scene traffic further, but more importantly it would reduce the number of scene segments, thus reducing ray duplication and ray traffic.

These comparisons offer only partial evaluation because raw memory traffic does not consider DRAM management like row buffer hit rates (see Section 2.1) which can have a huge impact on the actual latency and power of the memory system. In fact, it is possible for memory bandwidth to increase while reducing access latency [Kopta et al. 2015]. The comparisons also point out the main challenge in extending dual streaming: managing, minimizing, and compressing ray traffic (see discussions of future work in Section 7).

Because dual streaming processes wavefronts of rays, renderers and scenes that generate many ray bounces would require many passes, which could result in undesired drop in processor utilization and increase in memory traffic per ray. A limit to the number of ray bounces would bound this, but also introduce rendering bias, which, depending on the scene and light transport, may or may not be negligible.

Dual streaming architecture does not directly address building the acceleration structure and scene segments on chip. Because dual streaming is envisioned as a graphics accelerator, an external process would generate the scene stream and load it into DRAM before rendering a frame. However, dual streaming could be modified to rely on its general purpose execution units for this task.

7 FUTURE WORK

Because dual streaming completely re-orders the traversal within ray tracing to address the fundamental problems of the traditional traversal order, it also exposes new and interesting challenges. They represent fertile ground for additional optimizations and future research. We discuss some of these challenges here.

Treelet Assignment: Optimizing treelet assignment to limit ray duplication and thus ray bandwidth is likely to yield better performance than optimizing to accelerate the traversal of individual rays. For example, it is unclear if treelets should prefer a shallow structure or if they should be constructed in a depth-first fashion producing deeper treelets. We would expect a combination of the two approaches to deliver superior performance.

Traversal order: Once a scene segment is processed, any and all of its child segments can be selected into the working set. Adjusting the predefined segment traversal order based on the structure of the BVH or altering the traversal order on-the-fly based on information gathered during the traversal of the parent segments can provide substantial performance benefits. For example, in our current implementation it is likely that rays are not traversed through the closer segment first, reducing the effectiveness of any early ray termination scheme. Modifying the traversal order to be more amenable for early ray termination without significant increase in memory traffic is an important open problem.

Early Termination: Unlike traditional ray tracing, implementing early ray termination is not trivial with dual streaming. Since ray stream excludes unique hit information shared by ray duplicates, the hit information must be gathered from the hit record separately. There are several alternatives to our current implementation. For example, if the hit record is not already on chip, it might be more beneficial for pre-test to traverse the ray anyway, instead of stalling until the read is serviced. Alternatively, a ray waiting for the hit record data can be simply skipped until the data arrives. Also, the hit information can be requested and cached for the entire bucket of rays before processing or even scheduling it.

Data Compression: The ray stream is the major component of the memory bandwidth used by dual streaming. Therefore, compressing the duplicated ray data might significantly improve the performance and reduce the energy cost [Keely 2014]. While the scene stream uses only a fraction of the memory bandwidth, compressing the scene data can still be helpful in reducing the number of segments and thereby reducing the number of duplicated rays. This would reduce the total memory bandwidth further.

Memory Optimizations: It may be possible to partition streams between on-chip and off-chip DRAM, in order to reduce energy further. In particular, because the bandwidth requirements for the scene stream are low and the access latency is hidden by prefetching, the scene data can reside on a slower off-chip memory. Furthermore, it may be possible to lower the operating frequency of the off-chip memory serving the scene stream to significantly reduce the energy use without impacting performance negatively. Moreover, the dual streaming architecture has the potential to take full advantage of the upcoming high bandwidth memory (HBM) systems [JDEC Standard 2015] and hide their additional latency through streaming.

DRAM Modifications: Ray streams effectively convert DRAM into a temporary staging buffer that writes and reads rays only once. Thus, after ray data is read from DRAM, there is no need to preserve it, which requires DRAM to write the contents of the row buffer back thus consuming energy and contributing to memory latency. A DRAM modified for ray streams could benefit from "destructive reads" which would avoid these costs.

Additional Streaming Opportunities: Our traversal guarantees that the scene geometry is accessed at most once per pass. This structure can be used for rendering extremely large scenes that cannot fit in memory by streaming them from a disk or other high latency locations [Eisenacher et al. 2013].

8 CONCLUSION

We introduced the dual streaming approach that restructures ray traversal into a predictable process that allows both scene data and

ray data to be streamed from main memory in a highly structured way. This approach is tailored to the fundamental operation of DRAM memory, where data accessed sequentially from an open row buffer is dramatically more efficient in both energy and latency than more random accesses. This streaming approach also eliminates some major bottlenecks inherent in traditional ray tracing order.

We also provided a first hardware implementation of dual streaming and test results using cycle-accurate hardware simulator, showing that our implementation of dual streaming already outperforms STRaTA, a highly optimized architecture for traditional ray tracing, in typical large scenes. Finally, we included an extensive discussion for potential future improvements on dual streaming implementation, providing a new avenue for further research on hardware accelerated ray tracing.

ACKNOWLEDGMENTS

This material is supported in part by the National Science Foundation under Grant No. 1409129. Josef Spjut and Elena Vasiou provided helpful feedback. Fairy Forest is from the University of Utah, Crytek Sponza is from Frank Meinl at Crytek and Marko Dabrovic, Dragon is from the Stanford Computer Graphics Laboratory, Vegetation and Hairball are from Samuli Laine, and San Miguel is from Guillermo Leal Laguno. Cem Yuksel combined Sponza atrium by Marko Dabrovic with the Stanford Dragon for Dragon Sponza.

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proc. High Performance Graphics*.

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics*. ACM, New York, NY, USA, 145–149.

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

R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. 2000. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proceedings of MICRO-33*. 245–257.

Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic ray stream traversal. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 151.

James Bigler, Abe Stephens, and Steven G. Parker. 2006. Design for Parallel Interactive Ray Tracing Systems. In *Symposium on Interactive Ray Tracing (IRT06)*.

Jacco Bikker. 2012. Improving Data Locality for Efficient In-Core Path Tracing. In *Computer Graphics Forum*, Vol. 31. 1936–1947.

Mahdi Nazm Bojnordi and Engin Ipek. 2012. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. In *International Symposium on Computer Architecture (ISCA '12)*.

Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. 2007. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*.

Erik Brunvand, Daniel Kopta, and Niladrish Chatterjee. 2014. Why Graphics Programmers Need to Know About DRAM. In *ACM SIGGRAPH 2014 Courses*.

N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. 2012. *USIMM: the Utah SImulated Memory Module.* Technical Report UUCS-12-02. University of Utah.

C. Eisenacher, G. Nichols, A. Selle, and B. Burley. 2013. Sorted Deferred Shading for Production Path Tracing. *Computer Graphics Forum* 32, 4 (2013).

Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 2008. Toward A Multicore Architecture for Real-time Ray-tracing. In *IEEE/ACM International Conference on Microarchitecture*.

Christiaan Gribble and Karthik Ramani. 2008. Coherent Ray Tracing via Stream Filtering. In *Symposium on Interactive Ray Tracing (IRT08)*.

Bruce Jacob, Spencer Ng, and David Wang. 2008. *Memory Systems - Cache, DRAM, Disk*. Elsevier.

JDEC Standard. 2015. *High Bandwidth Memory (HBM) DRAM.* Technical Report JESD325A. JDEC Solid State Technology Association.

James T. Kajiya. 1986. The Rendering Equation. In *Proceedings of SIGGRAPH*. 143–150.

Sean Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *High-Performance Graphics (HPG 2014)*.

John Kelm, Daniel Johnson, Matthew Johnson, Neal Crago, William Tuohy, Aqeel Mahesri, Steven Lumetta, Matthew Frank, and Sanjay Patel. 2009. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09*.

Hong-Yun Kim, Young-Jun Kim, and Lee-Sup Kim. 2010. Reconfigurable mobile stream processor for ray tracing. In *Custom Integrated Circuits Conference (CICC)*.

Hong-Yun Kim, Young-Jun Kim, and Lee-Sup Kim. 2012. MRTP: Mobile Ray Tracing Processor With Reconfigurable Stream Multi-Processors for High Datapath Utilization. *IEEE Journal of Solid-State Circuits* 47, 2 (feb. 2012), 518–535.

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2013. An energy and bandwidth efficient ray tracing architecture. In *Proc. High-Performance Graphics*. ACM, 121–128.

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2015. Memory Considerations for Low Energy Ray Tracing. *Computer Graphics Forum* 34, 1 (2015), 47–59.

Daniel Kopta, Josef Spjut, Erik Brunvand, and Alan Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design (ICCD)*.

Won-Jong Lee, Shi-Hwa Lee, Jae-Ho Nah, Jin-Woo Kim, Youngsam Shin, Jaedon Lee, and Seok-Yoon Jung. 2012. SGRT: a scalable mobile GPU architecture based on ray tracing. In *ACM SIGGRAPH 2012 Posters (SIGGRAPH '12)*.

Won-Jong Lee, Youngsam Shin, Seok Joong Hwang, Seok Kang, Jeong-Joon Yoo, and Soojung Ryu. 2015. Reorder buffer: an energy-efficient multithreading architecture for hardware MIMD ray traversal. In *Proc.High-Performance Graphics*. ACM, 21–32.

Gábor Liktor and Karthik Vaidyanathan. 2016. Bandwidth-efficient BVH Layout for Incremental Hardware Traversal. In *Proc. High Performance Graphics*. ACM.

B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon. 2010. Cache-oblivious ray reordering. *ACM Trans. Graph.* 29, 3 (2010).

N. Muralimanohar, R. Balasubramonian, and N. Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*.

Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. 2014. RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices. *ACM Trans. Graph.* 33, 5 (Sept. 2014).

Paul Navrátil, Donald Fussell, Calvin Lin, and William Mark. 2007. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing, 2007. IEEE Symposium on*. 95–104.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers (SIGGRAPH '10)*.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97*. 101–108.

Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. 2002. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics* 21, 3 (2002).

Karthik Ramani and Christiaan Gribble. 2009. StreamRay: A Stream Filtering Architecture for Coherent Ray Tracing. In *ASPLOS '09*.

J. Schmittler, I. Wald, and P. Slusallek. 2002. SaarCOR – A Hardware Architecture for Realtime Ray-Tracing. In *EUROGRAPHICS Workshop on Graphics Hardware*.

J. Schmittler, S. Woop, D. Wagner, W. Paul, and P. Slusallek. 2004. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Graphics Hardware Conference*. 95–106.

Maxim Shevtsov, Alexei Soupikov, Alexander Kapustin, and Nizhniy Novorod. 2007. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Procedings of GraphiCon'2007*. Moscow, Russia.

Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing. *IEEE Trans. on CAD* 28, 12 (2009).

Josef Spjut, Daniel Kopta, Solomon Boulos, Spencer Kellis, and Erik Brunvand. 2008. TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In *IEEE Symposium on Application Specific Processors (SASP)*.

Ingo Wald, Christiaan P. Gribble, Solomon Boulos, and Andrew Kensler. 2007. *SIMD Ray Stream Tracing-SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Technical Report UUSCI-2007-012. SCI Institute, University of Utah.

I. Wald, S. Woop, C. Benthin, G. Johnson, and M. Ernst. 2014. Embree - A Kernel Framework for Efficient CPU Ray Tracing. In *ACM SIGGRAPH*.

Amy Williams, Steve Barrus, R.K̆eith Morley, and Peter Shirley. 2005. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphics Tools* 10, 1 (2005).

Sven Woop, Erik Brunvand, and Philipp Slusallak. 2006. Estimating Performance of a Ray Tracing ASIC Design. In *IRT06*.

Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. on Graphics* 24, 3 (July 2005).

Wm. A. Wulf and S.A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News* 23, 1 (March 1995), 20–24.

Sung-Eui Yoon and Dinesh Manocha. 2006. Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Computer Graphics Forum*, Vol. 25. 507–516.