An Energy and Bandwidth Efficient Ray Tracing Architecture

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis*
School of Computing, University of Utah, Salt Lake City, UT

Abstract

We propose two hardware mechanisms to decrease energy consumption on massively parallel graphics processors for ray tracing while keeping performance high. First, we use a streaming data model and configure part of the L2 cache into a ray stream memory to enable efficient data processing through ray reordering. This increases the L1 hit rate and reduces off-chip memory accesses substantially. Second, we employ reconfigurable special-purpose pipelines than are constructed dynamically under program control. These pipelines use shared execution units (XUs) that can be configured to support the common compute kernels that are the foundation of the ray tracing algorithm, such as acceleration structure traversal and triangle intersection. This reduces the overhead incurred by memory and register accesses. These two synergistic features yield a ray tracing architecture that significantly reduces both power consumption and off-chip memory traffic when compared to a more traditional cache only approach.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: ray tracing, streaming, persistent pipelines, bandwidth reduction, energy reduction

1 Introduction

Ray tracing [Whitted 1980] has traditionally been considered too compute-intensive for interactive rendering. Primarily this is due to less predictable memory system access patterns and the attendant increase in system power consumption. As Moore's law continues to increase available computation resources, it has become more feasible to support algorithms like ray tracing even on mobile and embedded platforms where power consumption has become the primary bottleneck.

A number of researchers have leveraged SIMD GPU-style processing to enhance the speed of ray tracing (e.g. [Wald et al. 2001; Dmitriev et al. 2004; Reshetov et al. 2005; Wald et al. 2008; Bigler et al. 2006]), but ray tracing's divergent branching and irregular memory access patterns suggest that an alternate approach may be beneficial. Many have argued that a more decoupled parallel approach makes more sense for ray tracing [Govindaraju et al. 2008; Seiler et al. 2008; Spjut et al. 2009; Kelm et al. 2009; Kopta et al. 2010]. In these cases, researchers opt for a Multiple Instruction, Multiple Data (MIMD), or Single Program, Multiple Data (SPMD) style of execution which reduces the requirement of collecting and sorting rays into bundles suitable for SIMD processing.

*email: {dkopta, kshkurko, sjosef, elb, ald}@cs.utah.edu

In this paper, we examine ray tracing on non-SIMD parallel hardware and methods to reduce the power requirements while maintaining rendering speed without compromising image quality. For any specific algorithm, the arithmetic work load is fixed and leaves little room for energy optimization except at the circuit level. The primary opportunity lies in improving the memory system by restructuring data access patterns to increase cache hit rates and reduce off-chip memory bandwidth. From an energy and delay perspective, this is fortunate since getting an operand from main memory is both slower and three orders of magnitude more energy expensive than doing a floating point arithmetic operation [Dally 2013]. Additional improvements are possible by reducing register access, and instruction fetch and decode energy by algorithmic or architectural improvements.

Specifically, we propose two mechanisms to improve energy performance for ray tracing. First, we use a streaming data model and treelet decomposition of the acceleration structure similar to [Aila and Karras 2010] but with specific hardware support for stream buffers to increase L1 cache hit rates and reduce traffic to the off-chip DRAM. Note that the order in which the ray data entries in these "buffers" are accessed is not important. In this work, we employ single linked lists which are accessed in a LIFO manner. This choice minimizes hardware overhead, allows a large number of these LIFO structures to co-exist in a single memory block, and removes the need to keep each structure in a contiguous address space. The data streams for the traversal phase are marshaled using a program controlled SRAM in place of a traditional L2 cache.

Second, we employ special-purpose pipelines which are dynamically configured under program control. These pipelines consist of execution units (XUs), multiplexers (MUXs), and latches that are shared by multiple lightweight thread processors. Our focus in this work is on the traversal and primitive intersection phases. We do not attempt to optimize shading. The result is that we construct two special purpose pipelines: one for BVH box intersection and the other for triangle intersection. The essential benefit of this tactic is to replace a large number of conventional instructions with a single large fused box or triangle intersection instruction. This significantly reduces register and instruction memory accesses as well as reducing the instruction decode overhead. The energy efficiency of these pipelines is similar to an ASIC design except for the relatively small energy overhead caused by the MUXs and slightly longer wire lengths [Mathew et al. 2004]. However unlike ASICs, our pipelines are flexible since they are configured under program control. These two synergistic features yield a ray tracing architecture that significantly improves both power consumption and off-chip memory traffic for intersection and traversal when compared to a more traditional approach, while preserving frame rates, and the quality rendering that is the hallmark of ray tracing.

2 Background

Recent work in ray tracing has explored a variety of ways to increase efficiency. Software approaches to increase performance on traditional GPUs involve gathering rays into packets to better match the SIMD execution model [Bigler et al. 2006; Boulos et al. 2007; Günther et al. 2007; Overbeck et al. 2008]. These systems also tend to increase cache hit rates because the ray packets operate on similar regions of interest. As an example of a SIMD approach

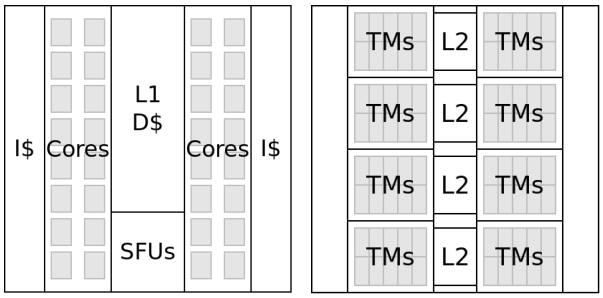
targeted specifically to ray tracing, the Mobile Ray Tracing Processor [Kim et al. 2012], traces packetized rays through the scene using four computation kernels, one for each step of the ray tracing algorithm. To effectively handle their Single Instruction, Multiple Thread (SIMT) execution model for ray tracing, the processor is able to dynamically switch between 12-way SIMT (12 processors each running the same instruction kernel on scalar data) and 4-way 3-vector processing (four threads, each using a 3-vector data path) for different phases of the algorithm. While different from the data path reconfiguration model we propose, it demonstrates that the different phases of the ray tracing algorithm can significantly benefit from hardware pipeline customization.

More directly related to this work, specific approaches to bandwidth reduction on more general architectures can involve cache-conscious data organization [Pharr and Hanrahan 1996; Christensen et al. 2003; Pharr et al. 1997; Mansson et al. 2007], and ray reordering [Steinhurst et al. 2005; Boulos et al. 2008; Navrátil and Mark 2006; Moon et al. 2010]. Some researchers specifically employ image-space rather than data-space partitioning for rays [Ize et al. 2011; Brownlee et al. 2012; Brownlee et al. 2013]. Stream-based approaches to ray generation and processing have also been explored both in a ray tracing context [Gribble and Ramani 2008; Ramani and Gribble 2009; Tsakok 2009; Aila and Karras 2010; Navratil et al. 2007] and a volume rendering context [Dachille and Kaufman 2000]. Although technical details are not published, at least two commercial hardware approaches to ray tracing appear to use some sort of ray sorting and/or classification [Imagination Technologies 2013; Silicon Arts Coproration 2013].

Architectural approaches for high-performance ray tracing have mostly involved the design and evaluation of non-SIMD parallel approaches that are better suited to the run-time branching behavior of ray tracing than wide SIMD processing [Govindaraju et al. 2008; Seiler et al. 2008; Spjut et al. 2009; Kelm et al. 2009; Kopta et al. 2010]. These efforts can mostly be characterized as lightweight general purpose cores with relatively simple memory systems that support the simple sharing model of the parallel ray tracing algorithm. We use this type of decoupled parallel architecture as a starting point for our exploration.

3 Streaming Treelet Ray Tracing Architecture (STRaTA)

We start with a parallel decoupled architecture called TRaX because it is designed specifically for ray tracing [Spjut et al. 2009], and because it is publicly available with a cycle-accurate simulator and LLVM-based compiler that we can modify for our own architectural evaluation [HWRT 2012]. We use this architecture as a starting point because we also believe that the MIMD execution model is better suited to ray tracing than the SIMD execution of traditional GPUs [Kopta et al. 2008; Kopta et al. 2010]. In addition, we can modify this architecture using the available tools to create STRaTA. The basic TRaX architecture is a collection of simple, in-order, single-issue integer thread processors (TPs) configured with general purpose registers (32 by default) and a small local memory (512B by default). The local memory acts as an extended register file for local stack operations. The generic TRaX thread multiprocessor (TM) aggregates a number of TP cores which share more expensive XUs such as floating point and inverse square root units. The TPs in a TM also share multi-banked L1 instruction and data caches. The TM and multi-TM chip architectures are shown in Figure 1. The specifics of the size, number, and configuration of the processor resources are variable in the simulator. We use this infrastructure to explore opportunities to decrease external memory bandwidth and energy usage in a parallel ray tracing architecture.



TRaX TM architecture with 32 lightweight cores and shared cache and compute resources

Potential TRaX chip organization with multiple TMs sharing L2 caches

Figure 1: *Overall TRaX Thread Multiprocessor (TM) and multi-TM chip organization [Spjut et al. 2009].*

In short, the TRaX MIMD sharing model is the opposite of what one finds in the SIMD structure of modern GPUs. TRaX shares data path XUs while allowing each TP to operate on a different instruction to better support the divergent branching associated with secondary ray processing. Individual TRaX TPs do not employ branch prediction and rely instead on thread parallelism to achieve high performance and to keep the shared floating point units busy. The result is small, simple TP cores, which can be tiled on a chip in a reasonable die area budget. Multiple TP cores (32 by default) form a TM block. Multiple TMs may then be aggregated onto a chip with larger shared L2 caches. The result is a very large number of small lightweight cores and a simple hierarchical memory system. The throughput of TRaX is primarily limited by power and bandwidth rather than the lack of computational resources.

3.1 Stream Queues

Aila and Karras [Aila and Karras 2010] focus on reducing off-chip bandwidth by partitioning the Bounding Volume Hierarchy (BVH) tree into sub-groups called treelets, sized to fit comfortably in either the L1 or L2 data cache. Each node in the BVH belongs to exactly one treelet, and treelet identification tags are stored along with the node ID. During traversal, when a ray crosses a treelet boundary, it is sent to a corresponding buffer where its computation is deferred until a processor is assigned to that buffer. In this scheme, a processor will work for a prolonged period of time only on rays that traverse a single treelet. This allows that subset of BVH data to remain in the L1 cache associated with that processor to increase the L1 hit rate. This technique requires many rays to be in flight at once in order to fill the treelet buffers, as opposed to the typical single ray at a time per core model. The state of each ray must be stored in global memory and passed along to other processors as needed. Ideally, this auxiliary ray state storage should not increase off-chip bandwidth drastically, since overall reduced bandwidth is the end goal.

We adapt Aila's approach by partitioning a special purpose ray stream memory that replaces some or all of the L2 data cache. This avoids auxiliary traffic by never saving ray state off-chip, at the cost of a lower total number of rays in flight, which are limited by the size of the ray stream partition. The TRaX architecture uses very simple direct-mapped caches, which prove to work well enough for the ray tracing data access patterns [Kopta et al. 2010], and save area and power over a more complex associative caches. We assign treelets to be exactly the size of an L1 cache, and a preprocessing step arranges the treelets into cache-aligned contiguous address

Figure 2: *Benchmark scenes used to evaluate performance.*

spaces. Since the L1 only contains treelet data, this guarantees that while a TM is working on a specific treelet, each line in the TM's L1 cache will incur at most one miss, and will be transferred to the L1 only once. We also modify Aila's algorithm to differentiate triangle data from BVH data, and assign each to a separate treelet. This ensures that any TM working on a leaf or triangle treelet is doing nothing but triangle intersections, allowing us to configure a specialized pipeline for triangle intersection (see Section 3.3). Similarly, when working on a non-leaf BVH treelet, the TM is computing only ray-box intersections utilizing a box intersection pipeline.

The ray stream memory holds the buffers for every treelet. Any given buffer can potentially hold anywhere from zero rays up to the maximum number that fit in the stream memory, leaving no room for any of the other buffers. The capacity of each buffer is thus limited by the number of rays in every other buffer. Although the simulator models these dynamically-sized buffers as a simple collection data structure, we envision a hardware model in which the buffers are implemented using a hardware managed linked-list state machine with a pointer to the head of each buffer stored in the SRAM. Link pointers for the nodes and a free list could be stored within the SRAM as well. This would occupy a small portion of the potential ray memory: not enough to drastically affect the total number of rays in flight since it only requires 8% or less of the total SRAM capacity for our tested configurations. The energy cost of an address lookup to find the head of the desired buffer, plus the simple circuitry to manage the linked-list is assumed to be roughly equal to the energy cost of the tag and bank circuitry of the L2 cache that it is replacing. We believe this is a conservative assumption that will need to be more precisely quantified in future work.

The programmer fills the buffers with some initial rays before rendering begins, using provided API functions to determine maximum stream memory capacity. These initial rays are all added to the buffer for the top-level treelet containing the root node of the BVH. After the initial rays are created, new rays are added to the top treelet buffer but only after another ray has finished processing, thus new rays effectively replace old ones. When a ray completes traversal, the executing thread may either generate a new secondary shadow ray or global illumination bounce ray for that path, or a new primary ray if the path is complete. Rays are removed from and added to the buffers in a one-to-one ratio, and managing ray generation is left to the programmer with the help of the API. Data for each ray requires 48 bytes comprised of: ray origin and direction (24 bytes total), ray state (current BVH node index, closest hit, traversal state, ray type, etc. totaling 20 bytes), and a traversal stack (4 bytes, see section 3.2).

3.2 Traversal Stack

Efficient BVH traversal attempts to minimize the number of nodes traversed by finding the closest hit point as early as possible. If a hit point is known and it lies closer than an intersection with a BVH node, then the traversal can terminate early by ignoring that branch of the tree. To increase the chances of terminating early, most ray tracers traverse the closer BVH child first. Since it is non-deterministic which child was visited first, typically a traversal stack is used to keep track of nodes that need to be visited at each level. One can avoid a stack altogether by adding parent pointers to the BVH, and using a deterministic traversal order (such as always left first then right), this however eliminates the possibility of traversing the closer child first and results in less efficient traversal.

Streaming approaches such as the one used in this work typically require additional memory space to store ray state. Rays are passed around from core to core and are stored in some memory buffer. In our case, the more rays present in a buffer, the longer a TM can operate on that treelet, increasing the bandwidth savings. Storing the entire traversal stack with every ray has a very large memory cost, and would reduce the total number of rays in flight significantly. There have been a number of recent techniques to reduce or eliminate the storage cost of a traversal stack, at the cost of extra work during traversal or extra data associated with the BVH such as parent pointers [Smits 1998; Laine 2010; Hapala et al. 2011].

We use a traversal technique in which parent pointers are included with the BVH so full node IDs are not required for each branch decision. We do, however, need to keep track of which direction (left or right) was taken first at each node. To reduce the memory cost of keeping this information we store the direction as a single bit on a stack and thus the entire stack fits in one 32-bit integer. Furthermore, there is no need for a stack pointer, as it is implied that the least significant bit (LSB) is the top of the stack. Stack operations are simple bitwise integer manipulations: shift left one bit to push, shift right one bit to pop. In this scheme, after a push, either 1 is added to the the stack (setting the LSB to 1, corresponding to left), or it is left alone (leaving the LSB as 0, corresponding to right). After visiting a node's subtree we examine the top of the stack. If the direction indicated on the top of the stack (left or right) is equal to which "side" the visited child was on, then we traverse the other child if necessary, otherwise we are done with both children and pop the stack and continue moving up the tree.

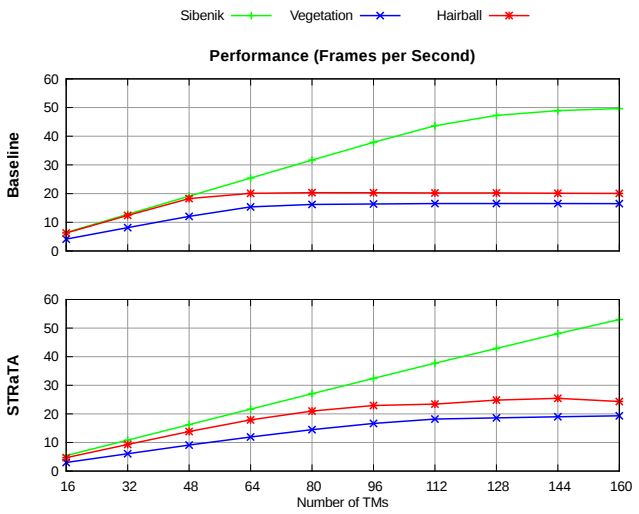


Figure 3: *Performance on three benchmark scenes with varying number of TMs. Each TM has 32 cores. Top graph shows baseline performance and bottom graph shows the proposed technique. Performance plateaus due to the 256GB/s bandwidth limitation.*

3.3 Reconfigurable Pipelines

One of the characteristics of ray tracing is that computation can be partitioned into distinct phases: traversal, intersection, and shading. Each phase has a small set of specific computations that dominate time and energy consumption. If the available functional units in a TM could be connected so that data could flow directly through a series of XUs without fetching new instructions for each operation, a great deal of instruction fetch and register file access energy could be saved. We propose repurposing the XUs by temporarily reconfiguring them into a combined ray-triangle or ray-box intersection test unit using a series of latches and MUXs when the computation phase can make effective use of that functionality. The overhead for this reconfigurability is fairly small as the MUXs and latches are small compared to the size of the floating point XUs, which themselves occupy a small portion of the circuit area of a TM.

Consider a hardware pipeline test for a ray intersection with an axis-aligned box. The inputs are four 3D vectors representing the two corners of the bounding box, the ray origin, and ray direction (12 floats total). Although the box is stored as two points, it is treated as three pairs of planes – one for each dimension in 3D [Smits 1998; Williams et al. 2005]. The interval of the ray's intersection distance between the near and far plane for each pair is computed, and if there is overlap between all three intervals, the ray hits the box, otherwise it misses. The bulk of this computation consists of six floating point multiplies and six floating point subtracts, followed by several comparisons to determine if the intervals overlap.

The baseline TRaX processor has eight floating point multiply, and eight floating point add/subtract units shared within a TM, which was shown to be an optimal configuration in terms of area and utilization for simple path tracing [Kopta et al. 2010]. Our ray-box intersection pipeline uses six multipliers and six add/subtract units, leaving two of each for general purpose use. The comparison units are simple enough that adding extra ones as needed for the pipeline to each TM has a negligible effect on die area. The multiply and add/subtract units have a latency of two cycles in 65nm at 1GHz, and the comparisons have a latency of one cycle. The box-test unit can thus be fully pipelined with an initiation interval of one and a latency of eight cycles.

Table 1: *Estimated energy per access in nanojoules for various memories. Estimates are from Cacti 6.5.*

L2/Stream memories

512KB	1MB	2MB	4MB	8MB	16MB
0.524	0.579	0.686	0.901	1.17	1.61

Inst. Cache	Reg. File	Off-Chip
4KB	128B	DRAM
0.014	0.008	16.3

Ray-triangle intersection is typically determined based on barycentric coordinates [Möller and Trumbore 1997] and is considerably more complex than the ray-box intersection. We remapped the computation as a dataflow graph, and investigated several potential pipeline configurations. Because an early stage of the computation requires a high-latency divide (16 cycles), all of the options have prohibitively long initiation intervals and result in poor utilization of execution units and low performance. An alternative technique uses Plücker coordinates to determine hit/miss information [Shevtsov et al. 2007] and requires the divide at the end of the computation, but only if an intersection occurs. If a ray intersects a triangle we perform the divide as a separate operation. Of the many possible ray-triangle intersection pipelines, we select one with a minimal resource requirement of four multipliers and two adders, which results in an initiation interval of 18, a latency of 31 cycles, and an issue width of two.

The final stage of our traversal shades the ray without reconfiguring the TM pipeline. In our test scenes, shading is a small portion of the total computation, and threads performing shading can take advantage of the general purpose XUs without experiencing severe starvation. Alternatively, if shading were more computationally intensive or if the data footprint of the materials is large, the rays could be sent to a separate queue or be processed by a pipeline configured for shading. The programmer invokes and configures these fixed-function pipelines with simple compiler intrinsics provided in an API. Since the pipelines have many inputs, the programmer is also responsible for loading the input data (a ray and a triangle/box) into special input registers via compiler intrinsics. This methodology keeps the instruction set simple and avoids any long or complex instruction words.

4 Results

We use three ray tracing benchmark scenes to evaluate the performance of our proposed STRaTA technique versus the TRaX baseline, as shown in Figure 2. All scenes are rendered using a single point-light source with simple path tracing [Kajiya 1986] because it generates incoherent and widely scattered secondary rays that provide a worst-case stress test for a ray tracing architecture. We use a resolution of 1024×1024, and a maximum ray-bounce depth of five resulting in up to 10.5 million ray segments per frame. Vegetation and Hairball have extremely dense, finely detailed geometry. This presents challenges to the memory system as rays must traverse a more complex BVH, and incoherent rays access large regions of the geometry footprint in unpredictable patterns. Sibenik is a much smaller scene with simpler architectural geometry, but is an enclosed scene forcing ray paths to reach maximum recursion depth before terminating.

We start with a baseline TRaX processor with 4MB of L2 cache shared among the TMs on the chip. The off-chip memory channels are capable of delivering a max bandwidth of 256GB/s from DRAM, similar to high-end GPUs. Figure 3 shows performance in frames per second using this baseline configuration for a varying

Table 2: *Instruction fetch energy (IFE) and register file energy (RegE) per frame (in Joules) using 128 TMs and an 8MB L2/Stream. Total (fetch + register) energy as a percentage of the baseline shown as Diff (lower is better). The STRaTA streaming technique uses more instructions than the baseline due to stream management overhead, but persistent pipelines are able to more than make up for that increase.*

	Sibenik			Vegetation			Hairball		
	IFE	RegE	Diff	IFE	RegE	Diff	IFE	RegE	Diff
Baseline	0.927	1.63	-	1.60	2.81	-	1.07	1.88	-
Streams	1.13	1.99	122%	1.69	2.99	106%	1.10	1.93	103%
Streams + Pipelines	0.911	1.60	98%	1.05	1.84	66%	0.805	1.41	76%

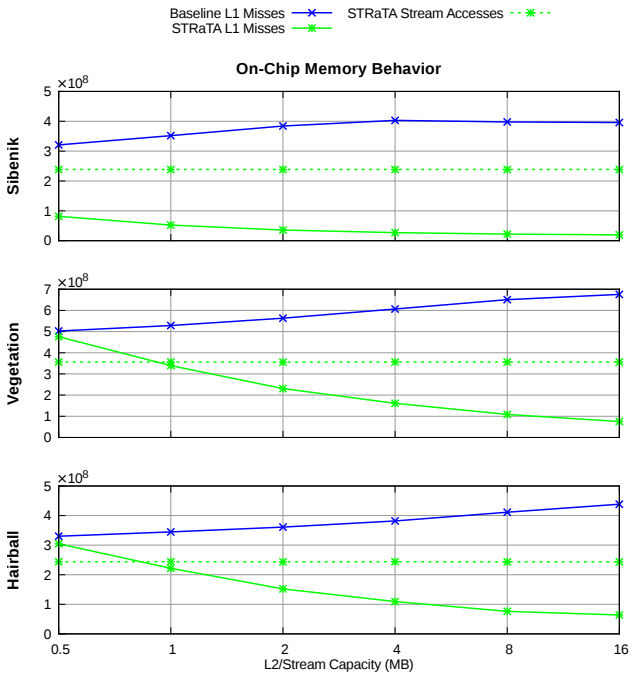


Figure 4: *Number of L1 misses (solid lines) for the baseline, and the proposed STRaTA technique and stream memory accesses (dashed line) on the three benchmark scenes. L1 hit rates range from 93% - 94% for the baseline, and 98.5% to 99% for the proposed technique.*

number of TMs. Recall that each TM consists of 32 thread processors, shared L1 instruction and data caches, and a set of shared functional units. On Hairball and Vegetation, performance quickly plateaus at 48 - 64 TMs for the basic non-streaming path tracer, and on Sibenik begins to level off rapidly around 112 TMs. After these plateau points, the system is unable to utilize any more compute resources due to data starvation from insufficient off-chip DRAM bandwidth.

Starting from this baseline configuration, we first investigate the effects on performance and bandwidth by repurposing the L2 cache as a dedicated ray stream memory. This involves replacing the L2 cache with a programmer-managed SRAM for storing and retrieving treelet streams. Treelet streams are rays associated with a particular BVH treelet. The size of the stream memory directly controls how many rays can be in flight at any given time. There is some overhead for using treelet streams and we want to make sure that this data structuring model does not adversely impact performance in terms of frames per second.

The STRaTA treelet-streaming model improves L1 hit rates significantly, but rather than remove the L2 cache completely we include a small 512KB L2 cache in addition to the stream memory to absorb some of the remaining L1 misses. Figure 3 also shows performance for the proposed STRaTA technique with increasing numbers of TMs. Performance does not differ drastically between the two techniques, and in fact the STRaTA technique has higher performance once the baseline is bandwidth constrained. The baseline performance will always be slightly higher if neither technique is bandwidth constrained, since the baseline has no treelet overhead. For the remainder of our experiments, we use 128 TMs (4K TPs), representing a bandwidth constrained configuration with a reasonable number of cores for current or near-future process technology.

Figure 4 shows the on-chip memory access behavior for each scene. The solid lines show total number of L1 misses (and thus L2 cache accesses), while the dotted lines show the total number of accesses to the stream memory for our proposed STRaTA technique. The size of the L2 cache (baseline) and stream memory (STRaTA) are the same. Reducing the number of accesses to these relatively large on-chip memories reduces energy consumption. The significant increase in L1 hit rate also decreases off-chip memory bandwidth which has an even more dramatic energy impact.

Note in Figure 4 that the number of L1 misses for the baseline technique increases (and thus L1 hit rate decreases) as the L2 capacity and frame rate increases. While this initially seems counterintuitive, there is a simple explanation. The L1 cache is direct mapped and shared by 32 threads which leads to an increased probability of conflict misses. As the size of the L2 cache increases, each thread has a reduced probability of incurring a long-latency data return from main memory since it is more likely that the target access will be serviced by the L2 cache. The increased performance of each thread generates a higher L1 access *rate* causing more sporadic data access patterns. The result is an increase in the number of L1 conflict misses. The number of stream accesses is constant with regards to the size of the stream memory because it is only affected by the number of treelet boundaries that an average ray must cross during traversal. Since the treelet size is held constant, the stream access patterns are only affected by the scene. Increasing the stream size does however increase the average number of rays in each treelet buffer, which allows a TM to spend more time processing while the treelet's subset of BVH data is cached in L1.

Figures 5 through 7 show the energy consumption per frame considering the L2 cache vs. stream memory accesses and off-chip memory accesses for each scene, based on the energy per access estimates in Table 1. All energy estimates are from Cacti 6.5 [Muralimanohar et al. 2007]. Not surprisingly, the baseline L2 cache energy consumption increases as larger capacities cause not only higher L1 miss rates (Figure 4), but the larger caches consume more energy per access. The proposed STRaTA technique consumes significantly less energy, but follows a similar curve. Note that the L1 misses (L2 accesses) for the proposed STRaTA technique in Figure 4 are to a fixed small, low energy 512KB L2 cache. The bulk of the energy comes from the stream memory accesses, which are constant with regards to stream size.

In addition to reducing memory traffic from the treelet-stream approach, we propose configuring the shared XUs into phase-specific

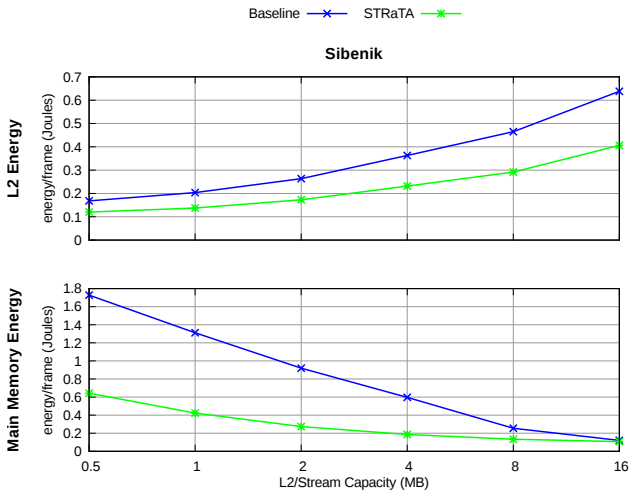


Figure 5: *Effect of L2 cache size (Baseline) and stream memory size (STRaTA) on memory system energy for the Sibenik scene.*

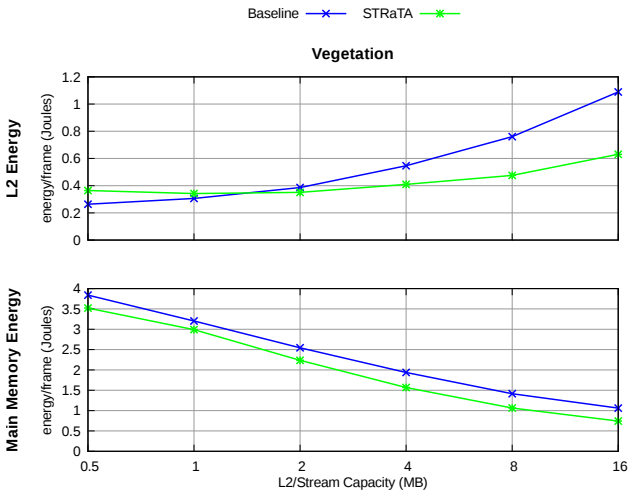


Figure 6: *Effect of L2 cache size (Baseline) and stream memory size (STRaTA) on memory system energy for the Vegetation scene.*

pipelines to perform box and triangle intersection functions. The effect of these pipelines is a reduction in instruction fetch and decode energy since a single instruction is fetched for a large computation, and a reduction in register file accesses since data is passed directly between pipeline stages. Table 2 shows the effect of the persistent pipeline techniques in STRaTA on the three test scenes. Note that the instruction fetch and register file energy actually increases for the basic STRaTA technique because of the overhead of the extra instructions used to manage the treelet streams. By engaging the phase-specific pipelines we see a reduction in instruction fetch and register file energy of between 2% and 34%.

The total energy used per frame for a path tracer in this TRaX-style architecture clearly is a function of the size of the L2 cache and stream memory, and whether the phase-specific pipelines are used. If we combine the two enhancements we see a total reduction in energy of the memory system (on- and off-chip memory and register file) and the instruction fetch of up to 38%. These reductions in energy come from relatively simple modifications to the basic parallel architecture with negligible overhead. They also have almost no impact on the frames per second performance, and actually increase the performance slightly in some cases. Although the functional unit energy has not changed, the significant reductions in energy used in the various memory systems, combined with low hardware overhead, implies that these techniques would be welcome additions to any hardware architecture targeting ray tracing.

5 Conclusions

The STRaTA design presented in this work demonstrates two improvements for ray tracing that can be applied to throughput oriented architectures. First, we provide a memory architecture to support smart ray reordering when combined with software that implements BVH treelets. By deferring ray computations through streaming rays, we can greatly increase our cache hit rates, and reduce the number of off-chip memory accesses by up to 70% on the Sibenik scene, and up to 27% on the larger scenes. Second, STRaTA allows shared XUs to be dynamically reconfigured into phase-specific pipelines to support the dominant computational phase for a particular treelet type. When these phase-specific pipelines are active, they reduce instruction fetch and register usage by up to 34%. The two STRaTA techniques are a more compelling competitor to current GPUs for ray trace acceleration.

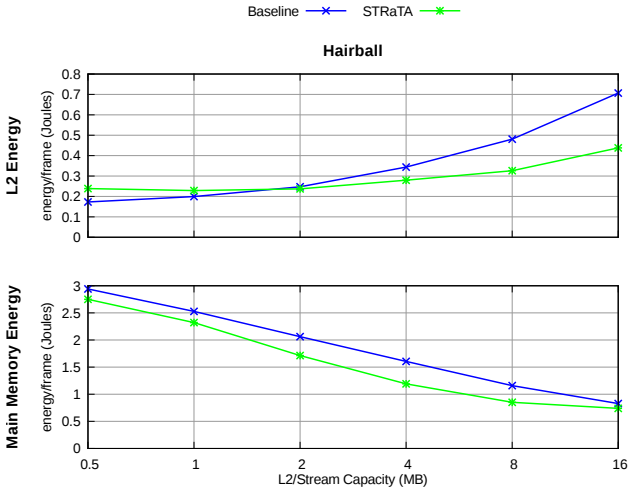


Figure 7: *Effect of L2 cache size (Baseline) and stream memory size (STRaTA) on memory system energy for the Hairball scene.*

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1017457. The Vegetation and Hairball models are from Samuli Laine, and the Sibenik Cathedral model is from Marko Dabrovic.

References

AILA, T., AND KARRAS, T. 2010. Architecture considerations for tracing incoherent rays. In *Proc. High Performance Graphics*.

BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for parallel interactive ray tracing systems. In *Symposium on Interactive Ray Tracing (IRT06)*.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive ray packet reordering. In *Symposium on Interactive Ray Tracing (IRT08)*.

BROWNLEE, C., FOGAL, T., AND HANSEN, C. D. 2012. GLuRay: Enhanced ray tracing in existing scientific visualization applications using OpenGL interception. In *EGPGV*, Eurographics, 41–50.

BROWNLEE, C., IZE, T., AND HANSEN, C. D. 2013. Image-parallel ray tracing using OpenGL interception. In *EGPGV*, Eurographics, 65–72.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Eurographics 2003*, 543–552.

DACHILLE, IX, F., AND KAUFMAN, A. 2000. Gi-cube: an architecture for volumetric global illumination and rendering. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, 119–128.

DALLY, B., 2013. The challenge of future high-performance computing. Celsius Lecture, Uppsala University, Uppsala, Sweden, Feb. `http://media.medfarm.uu.se/play/video/3261`.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster ray tracing with SIMD shaft culling. Tech. Rep. MPI-I-2004-4-006, Max-Planck-Institut für Informatik.

GOVINDARAJU, V., DJEU, P., SANKARALINGAM, K., VERNON, M., AND MARK, W. R. 2008. Toward a multicore architecture for real-time ray-tracing. In *IEEE/ACM Micro '08*.

GRIBBLE, C., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *Symposium on Interactive Ray Tracing (IRT08)*.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *Symposium on Interactive Ray Tracing (IRT07)*, 113–118.

HAPALA, M., DAVIDOVIC, T., WALD, I., HAVRAN, V., AND SLUSALLEK, P. 2011. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, 29–34.

HWRT, 2012. SimTRaX a cycle-accurate ray tracing architectural simulator and compiler. `http://code.google.com/p/simtrax/`. Utah Hardware Ray Tracing Group.

IMAGINATION TECHNOLOGIES, 2013. Caustic professional. `http://www.imgtec.com/caustic/`.

IZE, T., BROWNLEE, C., AND HANSEN, C. D. 2011. Real-time ray tracer for visualizing massive models on a cluster. In *EGPGV*, Eurographics, 61–69.

KAJIYA, J. T. 1986. The rendering equation. In *Proceedings of SIGGRAPH*, 143–150.

KELM, J. H., JOHNSON, D. R., JOHNSON, M. R., CRAGO, N. C., TUOHY, W., MAHESRI, A., LUMETTA, S. S., FRANK, M. I., AND PATEL, S. J. 2009. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09*.

KIM, H.-Y., KIM, Y.-J., AND KIM, L.-S. 2012. MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization. *IEEE JSSC 47*, 2, 518–535.

KOPTA, D., SPUJT, J., BRUNVAND, E., AND PARKER, S. 2008. Comparing incoherent ray performance of TRaX vs. Manta. In *Symposium on Interactive Ray Tracing (IRT08)*, 183.

KOPTA, D., SPJUT, J., BRUNVAND, E., AND DAVIS, A. 2010. Efficient mimd architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design (ICCD)*.

LAINE, S. 2010. Restart trail for stackless bvh traversal. In *Proc. High Performance Graphics*, 107–111.

MANSSON, E., MUNKBERG, J., AND AKENINE-MOLLER, T. 2007. Deep coherent ray tracing. In *Symposium on Interactive Ray Tracing (IRT07)*.

MATHEW, B., DAVIS, A., AND PARKER, M. 2004. A Low Power Architecture for Embedded Perception Processing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 46–56.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools 2*, 1 (October), 21–28.

MOON, B., BYUN, Y., KIM, T.-J., CLAUDIO, P., KIM, H.-S., BAN, Y.-J., NAM, S. W., AND YOON, S.-E. 2010. Cache-oblivious ray reordering. *ACM Trans. Graph. 29*, 3 (July), 28:1–28:10.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO '07*, 3–14.

NAVRÁTIL, P. A., AND MARK, W. R. 2006. An analysis of ray tracing bandwidth consumption. Tech. Rep. TR-06-40, The University of Texas at Austin.

NAVRATIL, P., FUSSELL, D., LIN, C., AND MARK, W. 2007. Dynamic ray scheduling for improved system performance. In *Symposium on Interactive Ray Tracing (IRT07)*.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large ray packets for real-time whitted ray tracing. In *Symposium on Interactive Ray Tracing (IRT08)*, 41–48.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop*, 31–40.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97*, 101–108.

RAMANI, K., AND GRIBBLE, C. 2009. StreamRay: A stream filtering architecture for coherent ray tracing. In *ASPLOS '09*.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (SIGGRAPH '05) 24*, 3, 1176–1185.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (August).

SHEVTSOV, M., SOUPIKOV, A., KAPUSTIN, A., AND NOVOROD, N. 2007. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Procedings of GraphiCon'2007*.

SILICON ARTS COPRORATION, 2013. RayCore series 1000. http://www.siliconarts.co.kr/gpu-ip.

SMITS, B. 1998. Efficiency issues for ray tracing. *J. Graph. Tools 3*, 2 (Feb.), 1–14.

SPJUT, J., KENSLER, A., KOPTA, D., AND BRUNVAND, E. 2009. TRaX: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design 28*, 12, 1802 – 1815.

STEINHURST, J., COOMBE, G., AND LASTRA, A. 2005. Reordering for cache conscious photon mapping. In *Proceedings of Graphics Interface 2005*, 97–104.

TSAKOK, J. A. 2009. Faster incoherent rays: Multi-BVH ray stream tracing. In *Proc. High Performance Graphics*, 151–158.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (EUROGRAPHICS '01) 20*, 3, 153–164.

WALD, I., BENTHIN, C., AND BOULOS, S. 2008. Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs. In *Symposium on Interactive Ray Tracing (IRT08)*, 49–57.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.

WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. 2005. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools 10*, 1.