EUROGRAPHICS Workshop on ... '0x
N.N. and N.N.
(Editors)

Volume 0 (200x), Number 0

Memory Considerations for Low Energy Ray Tracing

D. Kopta[1], K. Shkurko[1], J. Spjut[1,2,3], E. Brunvand[1], and A. Davis[1]

[1]School of Computing, University of Utah, Salt Lake City, UT, USA
{dkopta, kshkurko, sjosef, elb, ald}@cs.utah.edu
[2]NVIDIA, Santa Clara, CA, USA
[3]Department of Engineering, Harvey Mudd College, Claremont, CA, USA

Abstract

We propose two hardware mechanisms to decrease energy consumption on massively parallel graphics processors for ray tracing. 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 L1 hit rates and reduces off-chip memory energy substantially through better management of off-chip memory access patterns. To evaluate this model we augment our architectural simulator with a detailed memory system simulation that includes accurate control, timing, and power models for memory controllers and off-chip DRAM. These details change the results significantly over previous simulations that used a simpler model of off-chip memory, indicating that this type of memory system simulation is important for realistic simulations that involve external memory. Second, we employ reconfigurable special-purpose pipelines that are constructed dynamically under program control. These pipelines use shared execution units that can be configured to support the common compute kernels that are the foundation of the ray tracing algorithm. This reduces the overhead incurred by on-chip memory and register accesses. These two synergistic features yield a ray tracing architecture that reduces energy by optimizing both on-chip and off-chip memory activity when compared to a more traditional approach.

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

1. Introduction

Ray tracing [Whi80] has traditionally been considered to require too much computation to be used in interactive rendering. With the advances in integrated circuit process technologies, more computation capabilities have become available, typically in the form of increasingly many computation cores tiled on a chip. Ray tracing scales well with increases in available computation, but the memory system quickly becomes a bottleneck, particularly when assaulted with the random memory access patterns exhibited by naive ray tracing algorithms. Also, with the increase in available computation comes an increase in power consumption. Power is increasingly becoming a primary issue both in large high-performance chips, where power and heat are limiting how much of the chip can be active at one time [Dal13], and for chips targeting the embedded space where power is directly related to battery life and device temperature [She13]. Studying power consumption of processors under various graphical workloads is becoming more popular in the research literature [CLAL07, JGDAM12, MLC06, PLS11, PLS10a, SP10, PLS10b].

Many researchers have explored ways to harness available hardware parallelism to enhance the speed of ray tracing. These studies can be very broadly broken down into those that leverage single instruction, multiple data (SIMD) parallelism (e.g. [WSBW01, DHS04, RSH05, WBB08, BSP06, PBD*10, Ima13, Sil13]) and those that opt for a multiple instruction multiple data (MIMD) or single-program multiple-data (SPMD) approach (e.g. [GDS*08, SCS*08, SKKB09, KJJ*09, KSBD10, SKBD12, LSL*13]) or a mix of these (e.g. [WFWB13, WWB*14]).

In this paper, we examine ray tracing on non-SIMD parallel hardware. We explore methods to reduce the power requirements while maintaining rendering speed and without compromising image quality. We assume that for the ray tracing algorithm, the arithmetic work load is already highly

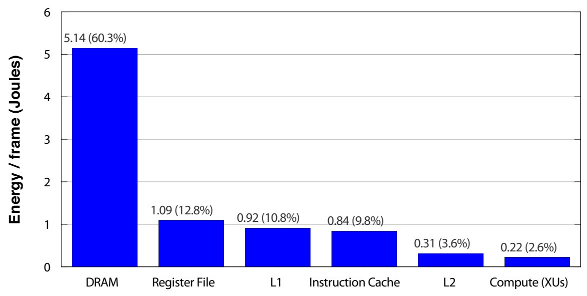

Figure 1: *Breakdown of energy consumption per frame, averaged over all test scenes in Figure 2, for our baseline architecture.*

optimized and leaves little room for energy reduction 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 access energy. From an energy and delay perspective, this is a compelling target because fetching an operand from main memory is both slower and three orders of magnitude more energy expensive than doing a floating point arithmetic operation [Dal13]. This is made clear by estimating the breakdown of energy consumption per frame using our baseline architectural simulation (see Figure 1). This estimate shows that DRAM energy takes up slightly over 60% of the entire energy for a frame for this system.

A natural approach would be to reduce the consumed bandwidth to main memory. This would decrease the number of main memory accesses, and thus reduce the energy cost related to gathering data. While this is certainly true, a more detailed look at DRAM energy costs reveals that because of the structure of DRAM circuits, changing the data access patterns is an equally effective means of reducing energy costs. This is true even in cases where the raw bandwidth consumption increases over the baseline system. This observation would not be possible without the addition of a detailed DRAM memory model to our architectural simulation. In this case we use the USIMM DRAM memory simulator which includes a detailed model of the complex timing and energy behavior of a modern DRAM memory system [CBS*12, MSC12]. The use of this memory simulator is a significant extension of our previous simulations [KSS*13] and is described in Section 4. 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 [NFLM07] and [AK10] but with specific hardware support for stream buffers to increase L1 cache hit rates and restructure the access patterns to the off-chip DRAM. This involves repurposing much of the L2 data cache as programmer-managed on-chip ray buffers with the goal of keeping ray data on-chip as much as possible (Section 3.1). Scene data is organized as treelets designed to work well both with the L1 data cache and with the internal DRAM row buffers (historically known as DRAM pages) (Section 4).

Second, we employ special-purpose pipelines which are dynamically configured under program control (Section 3.3). 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 in this work, though special purpose shading pipelines may also be effective. The result is that we construct two special purpose pipelines: one for bounding volume hierarchy 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 [MDP04, IPD04]. 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 power consumption for intersection and traversal when compared to a more traditional approach. Although power is the primary metric of interest, note that this technique has a processing overhead that can affect framerate. Once the system's bandwidth capability is saturated, the proposed technique always outperforms the baseline by allowing more efficient use of DRAM for large numbers of processors.

We use twelve ray tracing benchmark scenes, as shown in Figure 2, to evaluate the performance of our proposed technique. The scenes used represent a wide range of geometric complexities and data footprints: architectural models (Sibenik, Crytek, Conference, Sodahall, San Miguel), scanned models (Buddha, Dragon), and nature/game models (Fairy, Vegetation, Hairball). The laser scan models are unlikely to be used alone in empty space in a real situation such as a movie or game. Because of that we also include versions of them enclosed in a box, allowing rays to bounce around the environment.

2. Background

Recent work in ray tracing has explored a variety of ways to increase efficiency. Software approaches to increase performance on existing platforms involve gathering rays into packets to better match the SIMD execution model [BSP06,

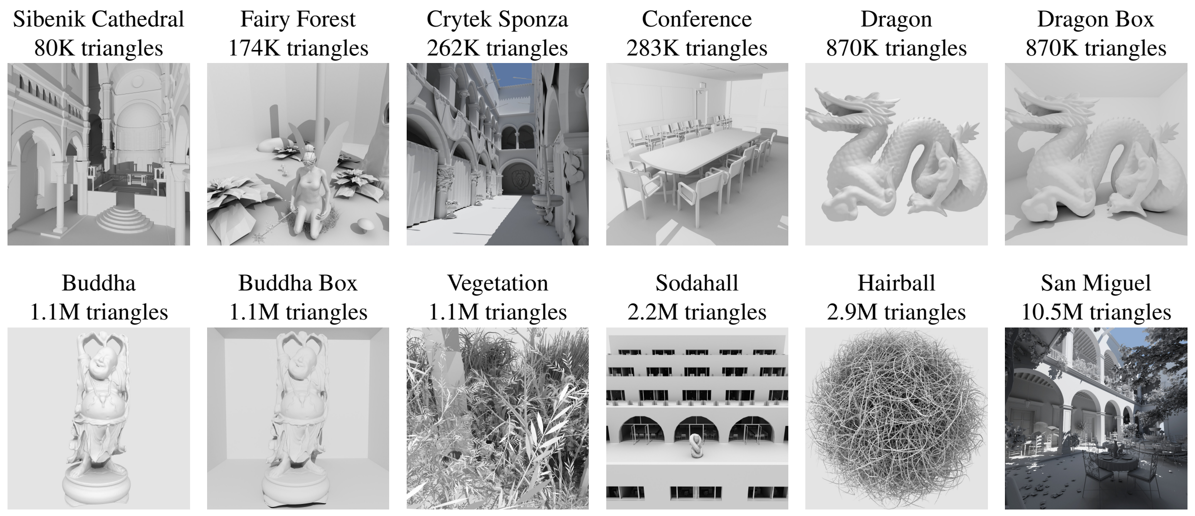

Figure 2: *Benchmark scenes used to evaluate performance.*

BEL*07, GPSS07, ORM08, WWB*14]. These systems can increase cache hit rates if they are able to assemble ray packets to operate on similar regions of interest. As an example of a SIMD approach targeted specifically to ray tracing, the Mobile Ray Tracing Processor [KKK12], 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, there are several approaches that attempt to reduce overall off-chip bandwidth requirements on more general architectures. These approaches can involve cache-conscious data organization [PH96, CLF*03, PKGH97, MMAM07], and ray reordering [SCL05, BWB08, NFLM07, MBK*10]. Some researchers specifically employ image-space rather than data-space partitioning for rays [IBH11, BFH12, BIH13]. Stream-based approaches to ray generation and processing have also been explored both in a ray tracing context [GR08, RG09, Tsa09, AK10, NFLM07] and a volume rendering context [DK00]. At least two commercial hardware approaches to ray tracing use some form of ray sorting and/or classification [Ima13, Sil13]. Our work has found that while an overall reduction in bandwidth consumption is certainly helpful, it can be equally helpful to carefully control the DRAM access patterns, even if that results in slight increases in bandwidth consumption in some cases.

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 [GDS*08, SCS*08, SKKB09, KJJ*09, KSBD10, SKBD12, LSL*13]. These efforts can be characterized broadly as tiled lightweight general purpose cores with relatively simple memory systems that do not support hardware shared memory. We use this type of parallel architecture as a starting point for our exploration.

3. Streaming Treelet Ray Tracing Architecture (STRaTA)

We start with our parallel MIMD architecture called TRaX because it is designed specifically for ray tracing [SKKB09], and because we also believe that the MIMD execution model is better suited to ray tracing than the SIMD execution of existing platforms [KSBP08, KSBD10, SKBD12]. We have made this model publicly available with a cycle-accurate simulator and LLVM-based compiler that can be modified for further architectural evaluation [HWR12]. Specifically, we 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 and a small local memory. The local memory acts as an extended register file for local stack operations. The exact size of these resources can be varied easily in our simulator. The generic TRaX thread multiprocessor (TM) aggregates a number of TPs which share more expensive XUs such as floating point and inverse square root units. The TPs in a TM also share separate multi-banked L1 instruction and data caches. The TM and multi-TM chip architectures are shown in Figure 3. The specifics of the size, number, and configuration of the processor resources are variable in the simulator. We use this infrastructure to explore how exploiting features of DRAM access patterns and access energy can lead to overall energy reduction in a parallel ray tracing architecture.

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. Individual TRaX TPs do not employ branch prediction or multi-threading and thus can operate effectively with relatively small register files. Note, however, the total register storage on a large chip is quite large in aggregate. Additional thread parallelism is achieved by adding more TPs, made possible by the simplicity of the TP core. The resulting small, simple TPs, can be tiled on a chip in a reasonable die area budget. Multiple TP cores 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 consumption rather than the lack of computational resources.

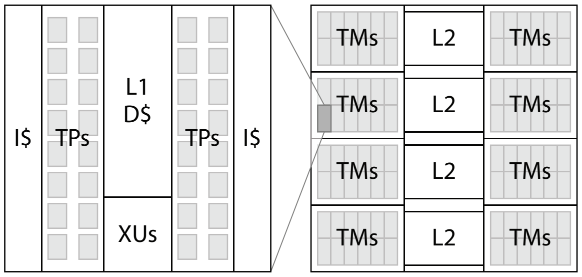

Figure 3: *Baseline TRaX architecture. Left: an example configuration of a single Thread Multiprocessor (TM) with 32 lightweight Thread Processors (TPs) which share caches (instruction I$, and data D$) and execution units (XUs). Right: potential TRaX chip organization with multiple TMs sharing L2 caches [SKKB09]. In a STRaTA configuration, the L2 caches are mostly replaced with ray stream buffer memory.*

3.1. Ray Stream Buffers

Recent work [NFLM07,AK10] focuses on reducing off-chip bandwidth consumption 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 ray 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 cache associated with that processor to increase cache hit rate. This technique requires many rays to be in flight at once in order to fill the treelet ray 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 consumption drastically, since reducing memory bandwidth is the end goal. In contrast to previous work, STRaTA stores the ray state in a buffer on-chip, therefore storing or retrieving rays does not affect the consumed off-chip bandwidth.

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 [KSBD10], and save area and power over more complex associative caches. We assign treelets to be exactly the size of an L1 cache, and the BVH builder arranges the treelets into cache-aligned contiguous address 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 type of treelet (see Figure 4). Note that triangle treelets are not technically a "tree", but simply a collection of triangles in nearby leaf nodes. 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. Most importantly, we differ from Aila et. al.'s work by focusing our evaluation on data access patterns to the DRAM rather than simply reducing raw bandwidth consumption. DRAM chips have a complex internal structure and correspondingly complex access protocols. Carefully controlling the access patterns to match the internal behavior of the DRAM can have a larger impact on memory access energy than reducing bandwidth consumption alone.

The ray stream memory holds the ray buffers for every treelet. Any given ray 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 ray buffer is thus limited by the number of rays in every other buffer. Although the simulator models these dynamically-sized ray buffers as a simple collection data structure, we envision a hardware model in which they 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 requires 8% or less of the total capacity for our tested configurations. The energy cost of an address lookup for the head of the desired ray buffer, plus the simple circuitry to handle the constant time push and pop operations onto the end of 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 that these energy costs are roughly comparable, but this assumption will need to be more precisely quantified in future work.

Note that the order in which the ray data entries in these buffers are accessed within a TM is not important. All rays in a buffer will access the same treelet, which will eventually be cache-resident. Rays that exit that treelet will be transferred to a different treelet's ray buffer. In this work, we employ singly 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.

Contiguous treelet data which fits in the L1 cache is also contiguous in the DRAM. This means that data loaded from the DRAM into the cache can amortize the internal structure of the DRAM in a way that greatly reduces both latency and energy compared to more random DRAM accesses. There are more details about this in Section 4 but essentially each access to DRAM loads a large amount of data (typically 4KB to 8KB) into a static "row buffer" (historically known as a DRAM page). Data accessed from a row buffer (e.g. for an L1 cache load) has a dramatically better latency and power profile than random DRAM accesses.

The programmer fills the ray 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 ray 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, where secondary rays replace the ray that spawned them to avoid overflowing on-chip ray buffers. Managing ray generation is done by the programmer with the help of the API. For example, during shading (when a ray has completed traversal/intersection), if another ray must be generated as part of the shader, the programmer simply adds that ray with the same pixel ID and updated state (such as ray type) to the root treelet ray buffer instead of immediately invoking a BVH traversal routine.

In this work 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 the 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 memory buffers. In our case, the more rays present in a buffer, the longer a TM can operate on that treelet, increasing the energy savings by not accessing off-chip memory during that computation. 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 size of a traversal stack, at the cost of extra work during traversal or extra data associated with the BVH such as parent pointers [Smi98, Lai10, HDW*11].

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 child or right child) 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 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 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 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.

3.3. Reconfigurable Pipelines

One of the characteristics of ray tracing is that computation can be partitioned into distinct phases: traversal, intersection, and shading. The traversal and intersection phases have a small set of specific computations that dominate time and energy consumption. If the available XUs 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 (i.e. time, energy and area) is fairly low 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 [MDP04, IPD04, RD07, Ram12].

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 [Smi98, WBMS05]. 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 [KSBD10]. 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.

Ray-triangle intersection is typically determined based on barycentric coordinates [MT97] and is considerably more complex than the ray-box intersection. We remapped the computation as a data-flow 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 [SSKN07] 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 outside of the pipeline. 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 shades the ray without reconfiguring the TM pipeline. In our test scenes, Lambertian shading is a small portion of the total computation, and threads performing shading can take advantage of the leftover 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 buffer or be processed by a pipeline configured for shading.

The programmer invokes and configures these phase-specific pipelines with simple compiler intrinsics provided in the API. Once a TM is configured into a specific pipeline, all of the TPs within operate in the same mode until reconfigured. 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. Accurate DRAM Modeling

A fair amount of recent work aims to reduce off-chip bandwidth consumption (see Section 2), since incoherent access to DRAM can be the main performance bottleneck in a ray tracer, and is clearly a large energy consumer. Raw bandwidth consumption however does not tell the full story, since the internal structure of DRAM yields highly variable energy and latency characteristics depending on access patterns. A benchmark with a higher number of total accesses but a friendlier access *pattern* may out-perform another benchmark even if it consumes less raw bandwidth.

The internal structure of a DRAM chip is logically organized as a set of memory arrays or banks. Each bank holds a portion of the total DRAM data, and can be accessed relatively independently. Because of the circuit structure of the DRAM bank arrays, they are accessed at the granularity of an entire row of the array, which typically contains 4KB or 8KB of data [JNW10]. This large chunk of data is staged in a static buffer on the chip known as a "row buffer." Because of the size of a typical row buffer, each row buffer may contain multiple cache lines.[†]

A basic read from DRAM consists of two phases: reading a row into a row buffer (this operation is known as opening the row), then selecting and reading a column from that row. If the address requested lies in the row that happens to be already open, only a column read must be performed, which is much cheaper than opening a row, both in terms of energy and delay. This is called a row buffer hit, and amortizes the significant energy cost of opening the row. When an address in a different row is required, the current row must be closed, and the new one opened. Closing a row requires re-writing the data from the buffer back into the memory cells, since the initial row read is destructive. In the worst case, an access pattern will read only a single column before requiring a new row. The memory controller can attempt to increase row buffer hit rate by preferentially scheduling reads to open rows, but there is a limit to its effectiveness with overly chaotic access patterns.

Many architectural simulations, including previous incarnations of our simulator [KSS*13], focus on accurate modeling of the on-chip systems, but use a simplified approximation for DRAM performance, such as assuming an average latency and energy for all reads and writes. USIMM is a DRAM simulator with sophisticated modeling of timing and energy characteristics for the entire DRAM system [CBS*12], and has been used by a number of simulation systems as an accurate memory model [MSC12,NCQ13]. In this work, we incorporate USIMM into the TRaX simulator, and adapt it to operate with on-the-fly DRAM requests as they are generated, as opposed to operating on trace files. The result is a cycle accurate simulation of a complete GPU architecture which reveals many performance characteristics previously hidden by a simpler DRAM model. DRAM performance is subtle and complex when all the details are exposed. For example, detailed DRAM behavior requires modeling the following types of activity:

- Opening/closing rows, sometimes called a page access, and modeling row hits vs. row misses - These can result in drastic differences in energy and delay.
- Scheduling (memory controller) - Reads can be serviced out of order, which results in opportunities for increasing row hits.
- Write drain mode - Draining the write queue disables reads for a long period of time, introducing hiccups in DRAM access timing.
- Refresh - Memory cells must be re-written periodically or they lose data. This disables everything for a long period of time, introducing hiccups, and consuming a large amount of energy.
- Separate memory clock - A memory controller can make decisions between GPU/CPU cycles.
- Queueing delay - All of the above behaviors have a combined effect on queueing delay.
- Background energy - DRAM energy is not only a function of the number and pattern of accesses, but also of running time.
- Address mapping policy - How addresses map to channels/banks/rows has a direct impact on how efficiently the data is accessed.

[†] Historically, DRAM chips that included static memory to hold an entire row from the internal memory arrays were known as "page mode" DRAMs, and the data fetched from the array on a single read was known as a DRAM page. This static buffer is known as a "row buffer" in modern DRAM parlance.

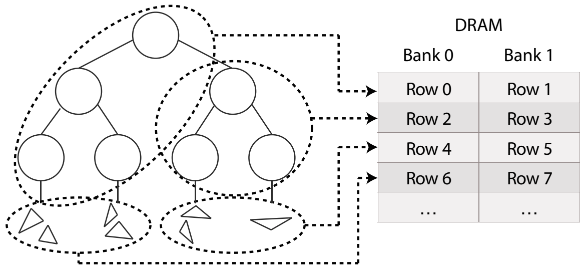

Figure 4: *Treelets are arranged in contiguous data blocks targeted as a multiple of the DRAM row size. In this example treelets are constructed to be the size of two DRAM rows. Primitives are stored in a separate type of "treelet" differentiated from node treelets, and subject to the same DRAM row sizes.*

To understand the key difference in DRAM access patterns between the baseline and STRaTA, we must examine the algorithmic source of these accesses. The baseline ray tracer's memory access pattern is determined by the nature of the BVH traversal algorithm. If no special care is taken in the implementation to govern memory access patterns this results in chaotic memory accesses when, for example, path tracing inevitably generates many incoherent rays. Accesses that miss in the L1 and L2 are thus both temporally and spatially incoherent, generating continuous moderate pressure on all channels, banks, and rows in DRAM.

STRaTA remaps the ray tracing algorithm to specifically target coherent L1 accesses. While a TM is operating on a certain treelet, all accesses will hit in the L1, except for the first to any given cache line. Ideally a TM will operate on the treelet for a prolonged period of time, generating no L2 or DRAM accesses. The accesses that do make it past the L1 occur right after a TM has switched to a new treelet. All threads within a TM will immediately begin reading cache lines for the new treelet, missing in the L1, and generating a very large burst of L2/DRAM accesses. While the small L2 cache in STRaTA may absorb some of this burst, the remainder that makes it to DRAM will have the same bursty structure. Intuitively this behavior may seem detrimental for

Table 1: *DRAM performance characteristics for baseline vs. STRaTA, where* ***bold*** *signifies the better performer. Read latency is given in units of GPU clock cycles. STRaTA DRAM energy is also shown as a percentage of baseline DRAM energy. For all columns except Row Buffer Hit Rate, lower is better.*

	Baseline					**STRaTA**				
Scene	DRAM Accesses (M)	Row Buffer Hit Rate (%)	Avg. Read Latency	ms / Frame	Total DRAM Energy (J)	DRAM Accesses (M)	Row Buffer Hit Rate (%)	Avg. Read Latency	ms / Frame	Total DRAM Energy (J)
Sibenik	39	69	39	**21**	1.7	**15**	**84**	**31**	23	**0.98 (58%)**
Fairy	22	62	49	**12**	1.1	**14**	**83**	**45**	16	**0.77 (70%)**
Crytek	59	44	60	**31**	3.5	**52**	**84**	**35**	34	**2.0 (57%)**
Conference	18	57	42	**17**	1.1	**9**	**83**	**35**	23	**0.84 (76%)**
Dragon	**70**	55	264	**22**	3.2	78	**80**	**63**	25	**2.5 (78%)**
Dragon Box	**168**	35	429	71	10.1	252	**80**	**65**	**57**	**7.3 (72%)**
Buddha	**47**	63	219	**13**	**1.9**	86	**77**	**83**	23	2.7 (142%)
Buddha Box	**133**	31	416	61	8.6	224	**78**	**63**	**54**	**6.8 (79%)**
Vegetation	**148**	43	346	56	8.2	160	**77**	**53**	**51**	**5.4 (66%)**
Sodahall	5	64	**41**	**8**	0.4	**4.5**	**72**	69	9	0.4 (100%)
Hairball	135	48	352	46	6.9	**126**	**75**	**62**	**40**	**4.3 (62%)**
San Miguel	**218**	27	352	108	14.8	323	**60**	**169**	**94**	**13.7 (93%)**

DRAM utilization; however, treelets are stored in a consecutive address block, mapping to as few DRAM row buffers as possible (in our configurations treelets are the size of two rows). Figure 4 shows an example of this. Once a DRAM row is open, the treelet data can flow quickly through the memory channel, at a very low energy cost compared to the more constant but incoherent accesses generated by the baseline. Table 1 shows a significant increase in row buffer hit rate for STRaTA on all benchmark scenes (Figure 2), as well as a reduction in read latency on all but one scene.

5. Results and Discussion

We start with a baseline TRaX system (based on previous area/performance explorations) with 128 TMs which results in 4096 total thread processors. Each thread processor is configured with 32 registers and a 512B local memory. Although these individual resources are small, the combination across all the TPs on a chip is large. Because individual TPs do not support multi-threading, we increase concurrency by adding TPs rather than increasing register file size. For the baseline non-STRaTA system we use a near-future capacity of 4MB of L2 cache shared among the TMs on the chip (current top-end GPUs have up to 1.5MB of on-chip L2). In this work each TM's L1 data cache (and thus maximum treelet size) is 16KB. To model near-future GPU DRAM capabilities, we configure USIMM for both the baseline and STRaTA to use GDDR5 with eight 64-bit channels, running at 2GHz (8GHz effective), for a total of 512GB/s maximum bandwidth. The chip configurations in this work use a 1GHz core clock rate.

All benchmark scenes (Figure 2) are rendered using a single point-light source and path tracing [Kaj86] 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. Our focus in this work is on the traversal and primitive intersection phases, and we do not attempt to optimize shading, so we use a simple Lambertian material on all scenes.

Our test renderer is limited to shading with non-branching ray paths. To support more advanced shaders, the programmer could add more information to the per-ray state to determine the remaining rays yet to be generated. When one shading ray finishes, a new ray could be generated with updated state for the associated shading point. Increasing the data footprint of rays will reduce the number of them that fit in the stream memory, but our results indicate that the number of rays in flight could decrease by a fair amount without being detrimental to the system. Another option is to allow the on-chip ray buffers to overflow to main memory when full (for example [AK10] store rays in main memory), although this would likely impact our DRAM access patterns negatively. Note that texture accesses could have similar DRAM access patterns if combined into texture "treelets" (with associated ray buffers) in an augmented STRaTA implementation.

5.1. Off-chip Memory Access

Starting from our baseline configuration, we first investigate the effects on DRAM access energy and performance 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 consist of 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. The STRaTA treelet-streaming model improves L1 hit rates significantly (Figure 5), 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.

Table 1 shows a breakdown of various DRAM characteristics on each scene, as well as total running time in $ms/frame$, for the baseline and STRaTA techniques. Note that although STRaTA increases L1 hit rates, the lack of a large L2 cache can result in a greater number of total DRAM accesses and thus bandwidth consumption on some scenes. However, the coherent pattern of STRaTA's accesses increases the row buffer hit rate significantly on all scenes, and drastically on some (San Miguel, Buddha Box, Dragon Box). Raw bandwidth consumption, while an interesting metric, does not reveal other subtleties of DRAM access; the increase in row buffer hit rate reduces DRAM energy consumed on all but two outlier scenes (Buddha increases by 42% and Sodahall is tied), discussed further below.

As a secondary effect, increased row buffer hit rate can also lead to greatly reduced read latency, up to 85% on the Dragon Box scene. This can result in higher performance, even though STRaTA introduces some overhead in the traversal phase due to its lack of a full traversal stack (Section 3.2) and the need to detect treelet boundaries.

There are two notable outlier scenes: Buddha and Sodahall. Buddha is the only scene in which STRaTA consumes more DRAM energy than the baseline. The major reason for this is that Buddha requires the fewest total rays to render. The Buddha is the only object in the scene so over half of the primary rays immediately hit the background and don't generate secondary bounces. The few rays that do hit the Buddha surface are likely to bounce in a direction that will also terminate in the background. Because of this, a disproportionate number of rays never leave the top level (root) treelet, and Buddha does not reach a critical mass of rays required for our ray buffers to function effectively. Hence, we also consider a more realistic scene by placing Buddha in a box.

When a TM switches to a treelet ray buffer, if there are not enough rays to keep all of its threads busy, many of the compute resources sit idle, effectively reducing parallelism. Even though STRaTA increases row buffer hit rates on Buddha, the increase in DRAM energy is partly background energy caused by the nearly doubled running time while threads sit idle. We note that DRAM energy is not only a function of the number and pattern of accesses, but it also has a dependency on the total running time (e.g. $ms/frame$ in Table 1), mostly

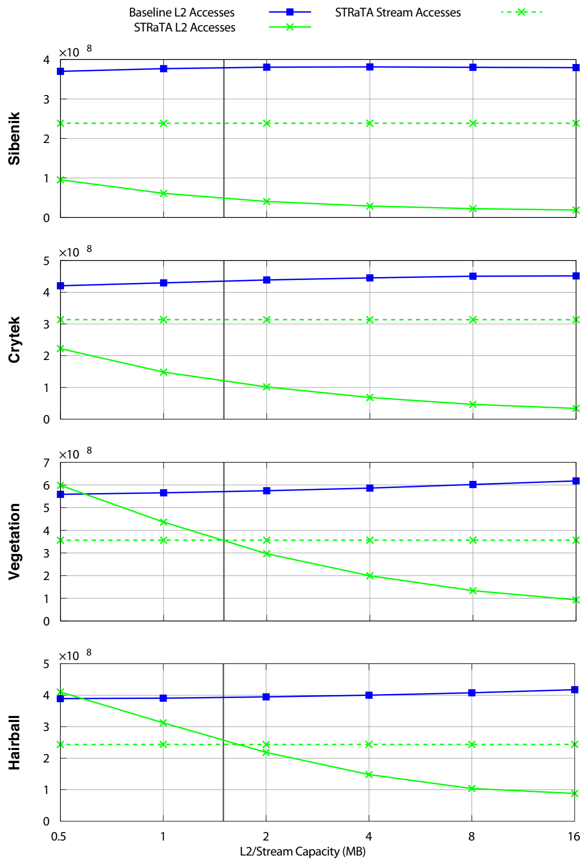

Figure 5: *Number of L1 misses (solid lines) for the baseline, and the proposed STRaTA technique, and stream memory accesses (dashed line) on a selection of benchmark scenes. L1 hit rates range 91% - 94% for the baseline, and 97% - 99% for STRaTA. The vertical dashed gray line corresponds to 1.5MB, the L2 cache size of NVIDIA's GK110 GPU.*

due to the need for continuous refreshing of the DRAM data even when no read/write activity occurs.

Also note that the baseline has a relatively high row buffer hit rate on Buddha, so STRaTA is unable to make as large of a difference. The Dragon scene is similar to Buddha, but does not exhibit this problem. Note that the baseline takes almost twice as long to render Dragon than Buddha, since Dragon fills a larger portion of the frame. This closes the gap in background energy between the two techniques. Dragon also results in more total rays, and has a smaller data footprint with fewer unique treelets and thus more rays on average in each buffer.

The other interesting outlier is Sodahall. Even though it has a large data footprint (2.2M triangles), it generates by far the fewest DRAM accesses. Most of the geometry is not visible from any one viewing angle since it is separated into

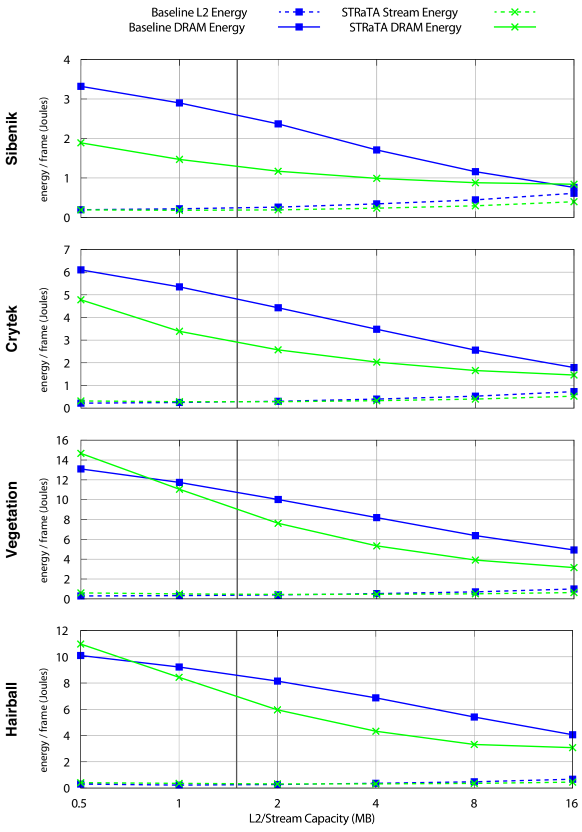

Figure 6: *Energy consumption for the L2 cache/stream memory and DRAM on a selection of benchmark scenes. The vertical dashed gray line corresponds to 1.5MB, the L2 cache size of NVIDIA's GK110 GPU.*

Table 2: *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
4KB	128B
0.014	0.008

many individual rooms. Only a small percentage of the total data is ever accessed. The pressure on DRAM is so low that background energy is the dominant factor for both STRaTA and the baseline. The viewpoint shown (Figure 2) has similar results to viewpoints inside the building.

5.2. On-chip Memory Access

Figure 5 shows the on-chip memory access behavior for a subset of our test scenes. All other scenes except Buddha have similar results. The solid lines show the 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. The significant increase in L1 hit rate allows STRaTA to do away with all but a very small L2 cache without detrimentally increasing DRAM accesses. More importantly, STRaTA's DRAM access *pattern* yields a much higher row buffer hit rate.

Note in Figure 5 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 counter-intuitive, 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 chance 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 a function of 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 which scene is being rendered. 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 the L1.

Figure 6 shows the energy consumption per frame considering the L2 cache/stream memory accesses and off-chip memory accesses for a subset of scenes. DRAM energy is reported by USIMM, and Table 2 shows energy estimates for the cache/stream memories from Cacti 6.5 [MBJ07]. Not surprisingly, the baseline L2 cache energy consumption increases as larger capacities consume more energy per access. The proposed STRaTA technique follows a similar curve, but both techniques are dominated by DRAM energy. Note that the L1 misses (L2 accesses) for the proposed STRaTA technique in Figure 5 are to a fixed small 512KB L2 cache. Table 1 shows that even though STRaTA sometimes requires more DRAM accesses, the energy consumed is lower in almost all scenes due to higher row buffer hit rates.

5.3. Phase-Specific Pipelines

In addition to the treelet-stream approach, we propose configuring the shared XUs into phase-specific pipelines to per-

Table 3: *Energy consumption (in Joules) for components affected by phase-specific pipelines. "RF" column refers to register file and "iCache" column refers to instruction cache. "Diff" column shows STRaTA as a percentage of the baseline for the sum of all energies shown.*

	Baseline		**STRaTA**		
Scene	RF	iCache	RF	iCache	Diff
Sibenik	1.18	0.9	0.88	0.73	77%
Fairy	0.67	0.51	0.49	0.41	76%
Crytek	1.85	1.42	1.33	1.11	75%
Conference	1.0	0.76	0.71	0.59	74%
Dragon	0.53	0.4	0.45	0.36	87%
Dragon Box	1.27	0.97	1.02	0.83	83%
Buddha	0.36	0.28	0.32	0.25	89%
Buddha Box	0.98	0.75	0.8	0.65	84%
Vegetation	1.74	1.33	1.25	1.04	75%
Sodahall	0.49	0.38	0.36	0.3	76%
Hairball	1.11	0.84	0.77	0.64	72%
San Miguel	1.96	1.5	1.38	1.15	73%

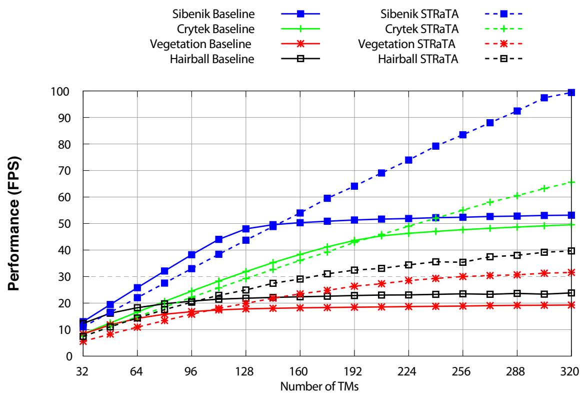

Figure 7: *Performance on a selection of benchmark scenes with varying number of TMs. Each TM has 32 cores. Performance plateaus due to DRAM over-utilization.*

form 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. This reduction in energy is largely independent of the other STRaTA features (treelets and ray buffers), since it simply reduces the energy cost associated with ray-triangle and ray-box intersection, and does not affect main memory traffic behavior. However, without treelets enabling the batch processing of rays performing the same task, reconfiguring the pipelines for individual rays would add significant overhead.

Table 3 shows the effect of STRaTA's phase-specific pipeline techniques on all test scenes. By engaging the phase-specific pipelines we see a reduction in instruction fetch and register file energy of between 11% and 28%.

In addition to reducing energy consumption, STRaTA can also increase performance scalability with the number of cores. Figure 7 shows performance for STRaTA with increasing numbers of TMs (dotted lines), compared to the baseline (solid lines) for a subset of benchmark scenes. With few cores when neither technique is memory bound, STRaTA initially has slightly lower performance due to the overhead added by using treelets during traversal. However, memory becomes a bottleneck much more quickly for the baseline than for STRaTA, which is able to utilize more cores to achieve significantly higher performance.

The total energy used per frame for a path tracer in this TRaX-style architecture is a strong function of the size of the L2 cache or stream memory, and whether the phase-specific pipelines are used. If we combine the treelet streaming and phase-specific pipeline enhancements we see a total system-wide reduction in energy (including all caches, DRAM, register files, and compute) of up to 30%. These reductions in energy come from relatively simple modifications to the basic parallel architecture with negligible overhead. 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.

6. 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 cache hit rates, and improve the off-chip memory access patterns, resulting in row buffer hit rates increasing from 35% to 80% in the best case, DRAM energy up to 43% lower, and DRAM read latencies up to 85% faster. Second, STRaTA allows shared XUs to be dynamically reconfigured into phase-specific pipelines to support the dominant computational kernel for a particular treelet type. When these phase-specific pipelines are active, they reduce instruction fetch and register usage by up to 28%.

More generally we show that understanding DRAM circuits is critical to making evaluations of energy and performance in memory-dominated systems. DRAM access protocols, and the resulting energy profiles, are complex and subtle. We show that managing DRAM access patterns (e.g. to optimize row buffer hit rates) can have a significantly greater impact on energy than simply reducing overall DRAM bandwidth consumption. These effects require a high-fidelity DRAM simulation, such as USIMM, that includes internal DRAM access modeling, and detailed modeling of the memory controller. The interaction between compute architectures and DRAM to reduce energy is an under-explored area. We plan to continue to explore how applications like ray tracing interact with the memory system. Especially interesting is the DRAM subsystem because DRAM access is the primary consumer of energy in a memory-constrained application such as ray tracing, or graphics rendering in general. In particular, one might develop a memory controller scheduler that is ray-tracing aware, and hide DRAM access optimizations from the programmer.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1017457. Manjunath Shevgoor, Niladrish Chatterjee, and Tim Purcell provided many useful discussions. Sibenik Cathedral is from Marko Dabrovic, Fairy Forest is from the University of Utah, Crytek Sponza is from Frank Meinl at Crytek and Marko Dabrovic, Conference is from Anat Grynberg and Greg Ward, Dragon and Buddha are from the Stanford Computer Graphics Laboratory, Vegetation and Hairball are from Samuli Laine, and San Miguel is from Guillermo Leal Laguno.

References

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proc. High Performance Graphics* (2010). 2, 3, 4, 8

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface* (2007). 2

[BFH12] BROWNLEE C., FOGAL T., HANSEN C. D.: GLuRay: Enhanced ray tracing in existing scientific visualization applications using OpenGL interception. In *EGPGV* (2012), Eurographics, pp. 41–50. 3

[BIH13] BROWNLEE C., IZE T., HANSEN C. D.: Image-parallel ray tracing using OpenGL interception. In *EGPGV* (2013), Eurographics, pp. 65–72. 3

[BSP06] BIGLER J., STEPHENS A., PARKER S. G.: Design for parallel interactive ray tracing systems. In *Symposium on Interactive Ray Tracing (IRT06)* (2006), pp. 187–196. 1, 2

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive ray packet reordering. In *Symposium on Interactive Ray Tracing (IRT08)* (2008). 3

[CBS*12] CHATTERJEE N., BALASUBRAMONIAN R., SHEVGOOR M., PUGSLEY S., UDIPI A., SHAFIEE A., SUDAN K., AWASTHI M., CHISHTI Z.: *USIMM: the Utah SImulated Memory Module*. Tech. Rep. UUCS-12-02, University of Utah, 2012. See also: http://utaharch.blogspot.com/2012/02/usimm.html. 2, 7

[CLAL07] CHANG C.-H., LOHRMANN P. J., AGU E. O., LINDEMAN R. W.: ENCORE: Energy-Conscious Rendering for Mobile Device. In *GPGPU* (2007). 1

[CLF*03] CHRISTENSEN P. H., LAUR D. M., FONG J., WOOTEN W. L., BATALI D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Eurographics 2003* (2003), pp. 543–552. 3

[Dal13] DALLY B.: The challenge of future high-performance computing. Celsius Lecture, Uppsala University, Uppsala, Sweden, 2013. http://media.medfarm.uu.se/play/video/3261. 1, 2

[DHS04] DMITRIEV K., HAVRAN V., SEIDEL H.-P.: *Faster Ray Tracing with SIMD Shaft Culling*. Tech. Rep. MPI-I-2004-4-006, Max-Planck-Institut für Informatik, December 2004. 1

[DK00] DACHILLE IX F., KAUFMAN A.: Gi-cube: an architecture for volumetric global illumination and rendering. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2000), HWWS '00, ACM, pp. 119–128. 3

[GDS*08] GOVINDARAJU V., DJEU P., SANKARALINGAM K., VERNON M., MARK W. R.: Toward a multicore architecture for real-time ray-tracing. In *IEEE/ACM Micro '08* (2008). 1, 3

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Symposium on Interactive Ray Tracing (IRT07)* (2007), pp. 113–118. 2

[GR08] GRIBBLE C., RAMANI K.: Coherent ray tracing via stream filtering. In *Symposium on Interactive Ray Tracing (IRT08)* (2008). 3

[HDW*11] HAPALA M., DAVIDOVIC T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011* (2011), pp. 29–34. 5

[HWR12] HWRT: SimTRaX a cycle-accurate ray tracing architectural simulator and compiler. http://code.google.com/p/simtrax/, 2012. Utah Hardware Ray Tracing Group. 3

[IBH11] IZE T., BROWNLEE C., HANSEN C. D.: Real-time ray tracer for visualizing massive models on a cluster. In *EGPGV* (2011), Eurographics, pp. 61–69. 3

[Ima13] IMAGINATION TECHNOLOGIES: Caustic professional, 2013. http://www.imgtec.com/caustic/. 1, 3

[IPD04] IBRAHIM A., PARKER M., DAVIS A.: Energy Efficient Cluster Co-Processors. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (2004). 2, 6

[JGDAM12] JOHNSSON B., GANESTAM P., DOGGETT M., AKENINE-MÖLLER T.: Power efficiency for software algorithms running on graphics processors. In *ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (2012), EGGH-HPG'12, Eurographics Association, pp. 67–75. 1

[JNW10] JACOB B., NG S., WANG D.: *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010. 6

[Kaj86] KAJIYA J. T.: The rendering equation. In *Proceedings of SIGGRAPH* (1986), pp. 143–150. 8

[KJJ*09] KELM J. H., JOHNSON D. R., JOHNSON M. R., CRAGO N. C., TUOHY W., MAHESRI A., LUMETTA S. S., FRANK M. I., PATEL S. J.: Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09* (2009). 1, 3

[KKK12] KIM H.-Y., KIM Y.-J., KIM L.-S.: MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization. *IEEE JSSC 47*, 2 (February 2012), 518–535. 2

[KSBD10] KOPTA D., SPJUT J., BRUNVAND E., DAVIS A.: Efficient MIMD architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design (ICCD)* (2010). 1, 3, 4, 6

[KSBP08] KOPTA D., SPUJT J., BRUNVAND E., PARKER S.: Comparing incoherent ray performance of TRaX vs. Manta. In *Symposium on Interactive Ray Tracing (IRT08)* (2008), p. 183. 3

[KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An energy and bandwidth efficient ray tracing architecture. In *High-Performance Graphics (HPG 2013)* (2013). 2, 7

[Lai10] LAINE S.: Restart trail for stackless BVH traversal. In *Proc. High Performance Graphics* (2010), HPG '10, Eurographics Association, pp. 107–111. 5

[LSL*13] LEE W.-J., SHIN Y., LEE J., KIM J.-W., NAH J.-H., JUNG S., LEE S., PARK H.-S., HAN T.-D.: Sgrt: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), ACM, pp. 109–119. 1, 3

[MBJ07] MURALIMANOHAR N., BALASUBRAMONIAN R., JOUPPI N.: Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO '07* (2007), pp. 3–14. 10

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

[MDP04] MATHEW B., DAVIS A., PARKER M.: A Low Power Architecture for Embedded Perception Processing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (2004), pp. 46–56. 2, 6

[MLC06] MOCHOCKI B., LAHIRI K., CADAMBI S.: Power analysis of mobile 3D graphics. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings* (2006), vol. 1, pp. 1 –6. 1

[MMAM07] MANSSON E., MUNKBERG J., AKENINE-MOLLER T.: Deep coherent ray tracing. In *Symposium on Interactive Ray Tracing (IRT07)* (2007). 3

[MSC12] MSC: 2012 memory scheduling championship, 2012. http://www.cs.utah.edu/~rajeev/jwac12/. 2, 7

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools 2*, 1 (October 1997), 21–28. 6

[NCQ13] NAIR P., CHOU C.-C., QURESHI M. K.: A case for refresh pausing in DRAM memory systems. In *IEEE Symposium on High Performance Computer Architecture (HPCA)* (2013), HPCA '13, IEEE Computer Society, pp. 627–638. 7

[NFLM07] NAVRATIL P., FUSSELL D., LIN C., MARK W.: Dynamic ray scheduling for improved system performance. In *Symposium on Interactive Ray Tracing (IRT07)* (2007). 2, 3, 4

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time whitted ray tracing. In *Symposium on Interactive Ray Tracing (IRT08)* (2008), pp. 41–48. 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers* (2010), SIGGRAPH '10, ACM, pp. 66:1–66:13. 1

[PH96] PHARR M., HANRAHAN P.: Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop* (1996), Springer, pp. 31–40. 3

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97* (1997), pp. 101–108. 3

[PLS10a] POOL J., LASTRA A., SINGH M.: An energy model for graphics processing units. In *Computer Design (ICCD), 2010 IEEE International Conference on* (2010), pp. 409 –416. 1

[PLS10b] POOL J., LASTRA A., SINGH M.: A per-unit breakdown of the energy consumption in a graphics processing unit. In *International Conference on Computer Design ICCD* (2010). 1

[PLS11] POOL J., LASTRA A., SINGH M.: Power-gated arithmetic circuits for energy-precision tradeoffs in mobile graphics processing units. *Journal of Low Power Eletronic Design 7*, 2 (April 2011). 1

[Ram12] RAMANI K.: *CoGenE: An Automated Design Framework for Domain Specific Architectures*. PhD thesis, University of Utah, 2012. 6

[RD07] RAMANI K., DAVIS A.: Application driven embedded system design: a face recognition case study. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (2007). 6

[RG09] RAMANI K., GRIBBLE C.: StreamRay: A stream filtering architecture for coherent ray tracing. In *ASPLOS '09* (2009). 3

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Transactions on Graphics (SIGGRAPH '05) 24*, 3 (July 2005), 1176–1185. 1

[SCL05] STEINHURST J., COOMBE G., LASTRA A.: Reordering for cache conscious photon mapping. In *Proceedings of Graphics Interface 2005* (2005), GI '05, Canadian Human-Computer Communications Society, pp. 97–104. 3

[SCS*08] 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., HANRAHAN P.: Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (August 2008). 1, 3

[She13] SHEBANOW M.: An evolution of mobile graphics. Keynote talk, HPG 2013, 2013. http:

//highperformancegraphics.org/wp-content/uploads/Shebanow-Keynote.pdf. 1

[Sil13] SILICON ARTS COPRORATION: RayCore series 1000, 2013. http://www.siliconarts.co.kr/gpu-ip. 1, 3

[SKBD12] SPJUT J., KOPTA D., BRUNVAND E., DAVIS A.: A mobile accelerator architecture for ray tracing. In *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)* (2012). 1, 3

[SKKB09] SPJUT J., KENSLER A., KOPTA D., BRUNVAND E.: TRaX: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design 28*, 12 (2009), 1802 – 1815. 1, 3, 4

[Smi98] SMITS B.: Efficiency issues for ray tracing. *J. Graph. Tools 3*, 2 (February 1998), 1–14. 5, 6

[SP10] SILPA B., PANDA P.: Introducing energy efficiency into graphics processors. In *Electronic System Design (ISED), 2010 International Symposium on* (2010), p. 10. 1

[SSKN07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A., NOVOROD N.: Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Procedings of GraphiCon'2007* (2007). 6

[Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-BVH ray stream tracing. In *Proc. High Performance Graphics* (2009), ACM, pp. 151–158. 3

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *Symposium on Interactive Ray Tracing (IRT08)* (2008), pp. 49–57. 1

[WBMS05] WILLIAMS A., BARRUS S., MORLEY R. K., SHIRLEY P.: An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools 10*, 1 (2005). 6

[WFWB13] WOOP S., FENG L., WALD I., BENTHIN C.: Embree ray tracing kernels for CPUs and the Xeon Phi architecture. In *SIGGRAPH Talks* (2013), p. 44. 1

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349. 1

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum (EUROGRAPHICS '01) 20*, 3 (September 2001), 153–164. 1

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree - a kernel framework for efficient CPU ray tracing. In *(to appear) ACM SIGGRAPH 2014 papers* (2014), SIGGRAPH '14, ACM. 1, 2