T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing

Jae-Ho Nah* Jeong-Soo Park* Chanmin Park† Jin-Woo Kim* Yun-Hye Jung* Woo-Chan Park‡ Tack-Don Han*
*Yonsei University, Korea †Samsung Electronics, Korea ‡Sejong University, Korea

Abstract

Ray tracing naturally supports high-quality global illumination effects, but it is computationally costly. Traversal and intersection operations dominate the computation of ray tracing. To accelerate these two operations, we propose a hardware architecture integrating three novel approaches. First, we present an ordered depth-first layout and a traversal architecture using this layout to reduce the required memory bandwidth. Second, we propose a three-phase ray-triangle intersection architecture that takes advantage of early exit. Third, we propose a latency hiding architecture defined as the ray accumulation unit. Cycle-accurate simulation results indicate our architecture can achieve interactive distributed ray tracing.

CR Categories: Computer Graphics [I.3.7]: Computer Graphics—Three-Dimensional Graphics and Realism–Ray tracing

Keywords: ray tracing, ray tracing hardware, global illumination

Links: DL PDF

1 Introduction

Ray tracing [Whitted 1980; Cook et al. 1984] is the most commonly-used algorithm for photorealistic rendering. Ray tracing generates a more realistic image than does rasterization, but it requires tremendous computational power for traversal and ray-primitive intersections. For this reason, it has been used for offline rendering for most of the last decade.

For real-time ray tracing, many approaches utilizing CPUs, GPUs, or custom hardware have recently been studied. These approaches do not yet provide sufficient performance for processing 1G rays/s for real-time distributed ray tracing [Govindaraju et al. 2008].

Most performance bottlenecks in ray tracing are in traversal and intersection tests [Benthin 2006]. Traversal is the process of searching an acceleration structure (AS), such as a kd-tree or bounding volume hierarchy (BVH), to find a small subset of the primitives for testing by the ray. A ray-primitive intersection test determines the visibility of primitives found during the traversal.

We believe a dedicated hardware unit for traversal and the intersection test is a suitable solution for real-time distributed ray tracing. In this paper, we present a custom hardware architecture, called T&I (traversal and intersection) engine. This architecture can be integrated with existing programmable shaders, as with raster operations pipelines (ROPs) or texture mapping units. Also, it comprises three novel approaches that are applicable to the traversal and intersection test processes.

First, an ordered depth-first layout (ODFL) and its traversal architecture are presented. The ODFL is the enhancement of an eight-byte kd-tree node layout [Pharr and Humphreys 2010]. It arranges the child node, which has a larger surface area than its sibling, adjacent to its parent to improve parent-child locality. We apply this layout to our traversal architecture to effectively reduce the miss rate of the traversal cache. The ODFL also can be easily applied to other CPU or GPU ray tracers. This concept was previously announced in the extended abstract [Nah et al. 2010].

Second, we propose a three-phase intersection test unit, which divides the intersection test stage into three phases. Phase 1 is the ray-plane test, Phase 2 is the barycentric coordinate test, and Phase 3 is the final hit point calculation. This configuration reduces the need for further computation and memory requests for missed triangles that are identified in either Phases 1 or 2. Phases 1 and 2 are performed in a common module because they use roughly the same arithmetic operations.

Third, a ray accumulation unit is proposed for hiding memory latency. This unit manages memory requests and accumulates rays that induce a cache miss. While the waiting missed block is fetched, other rays can perform their operations. When the missed block is fetched, the accumulated rays are flushed to the pipeline.

We verify the performance of our architecture with a cycle-accurate simulator and evaluate resource requirements and performance. We also perform a simulation with three types of rays that have different coherence. The proposed architecture achieves 44-1188 Mrays/s ray tracing performance at 500 MHz on 65 nm process.

The remainder of this paper is structured as follows. Section 2 describes related work. Section 3 gives an overview of the proposed architecture. In Sections 4 to 6, we cover the details of our three approaches (a traversal unit with the ODFL, a three-phase intersection test unit, and a ray accumulation unit). In Section 7, we describe the experimental results of the proposed architecture simulation. Finally, we conclude the paper in Section 8.

2 Related Work

2.1 Dedicated ray tracing hardware

SaarCOR [Schmittler et al. 2004] is a ray tracing pipeline that consists of a ray generation/shading unit, a 4-wide SIMD traversal unit, a list unit, a transformation unit, and an intersection test unit. Woop et al. [2005] presented the programmable RPU architecture, which performs ray generation, shading, and intersection tests with programmable shaders. For dynamic scenes, D-RPU [Woop et al. 2006a; Woop 2007] has a node update unit [Woop et al. 2006b] unlike RPU. RTE [Davidovic et al. 2011] is an optimized version of D-RPU that uses tail recursive shaders with treelets.

SaarCOR, RPU, D-RPU, and RTE are based on packet tracing [Wald et al. 2001], but packet tracing may result in low SIMD efficiency with incoherent rays as described in [Gribble and Ramani

ACM Reference Format
Nah, J., Park, J., Park, C., Kim, J., Jung, Y., Park, W., Han, T. 2011. T & I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing. *ACM Trans. Graph. 30*, 6, Article 160 (December 2011), 10 pages. DOI = 10.1145/2024156.2024194 http://doi.acm.org/10.1145/2024156.2024194.

2008]. In order to compensate for this drawback, Gribble and Ramani [2008] proposed a wide-SIMD ray tracing architecture using stream filtering. For higher SIMD utilization, this architecture filters active rays in a ray packet during each step of the operation. In contrast, TRaX [Spjut et al. 2009] and MIMD threaded multiprocessors [Kopta et al. 2010] exploit single-ray thread execution rather than packet tracing. They have lightweight thread processors, shared functional units, and a shared cache in the core. Finally, CausticRT [Caustic Graphics 2009] is commercial ray tracing hardware, but detailed architecture for it has not been published.

2.2 General purpose many-core architecture

There have been some studies of new many-core architectures for effective ray tracing. Larrabee [Seiler et al. 2008] uses multiple in-order x86 CPU cores for full-software rendering. This project has been shifted to the Intel Many Integrated Core (MIC) for high performance computing [Wald 2010]. Copernicus [Govindaraju et al. 2008] is a tiled multicore processor for ray tracing. Mahesri et al. [2008] proposed a new xPU architecture and evaluated its performance in visual computing applications, including ray tracing. OptiX [Parker et al. 2010] is a programmable ray tracing system designed for nVIDIA GPUs. Aila and Karras [2010] proposed an enhanced architecture, a stack-top cache and treelets, for the Fermi GPU to reduce the memory traffic in incoherent ray tracing.

3 Overall System Architecture

3.1 Basic design decisions

Fixed logic design: We designed the T&I engine with a fixed pipeline. The traversal and intersection tests are the most resource-consuming stages. Fixed logic can achieve higher performance than programmable logic through reduction of unused resources and the removal of instruction fetching and decoding. Fixed logic is appropriate because traversal and intersection testing are repeating operations. Other stages that require a certain degree of flexibility, such as ray generation and shading, are performed by a programmable shader. D-RPU employs the same strategy.

Single-ray tracing: Our architecture is based on single-ray tracing because single-ray tracing is more robust for incoherent rays than SIMD packet tracing. According to Mahesri et al. [2008], SIMD architectures showed poorer performance per area than MIMD architectures. Therefore, our method processes each ray independently via a different data path as in the MIMD approaches.

Acceleration structure (AS): We selected kd-trees for the AS. Although BVHs provide fast traversal performance in packet tracing [Wald et al. 2007] and GPU ray tracing [Aila and Laine 2009], the best AS is needed for single-ray tracing. In this case, kd-trees generally deliver faster ray intersection calculations than BVHs [Pharr and Humphreys 2010]. The reason for this is that only a single ray-plane intersection is required per kd-tree traversal step, and kd-trees do not suffer from node overlapping [Woop 2007].

Fully pipelined architecture: Our architecture is fully pipelined for high throughput. Therefore, our architecture requires shape data (e.g., node, list, and triangle) for each cycle. In order to reduce memory traffic, we assigned individual cache memories to all operation pipelines except for the second ray-triangle intersection unit, which does not require data fetching.

3.2 System organization

Figure 1 illustrates the system organization including the T&I engine consisting of a ray dispatcher (RD), traversal units (TRVs), list

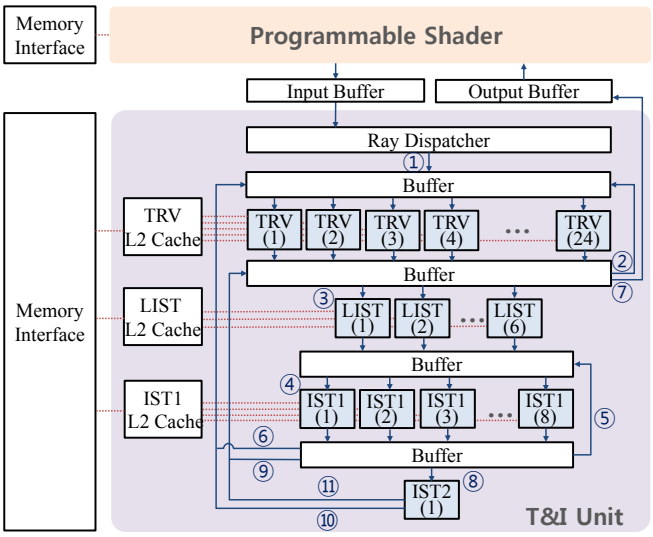

Figure 1: *Overall system architecture of the T&I engine.*

units (LISTs), the first intersection units (IST1s), and the second intersection unit (IST2). Each unit is connected by buffers that pass a ray from one unit to the others.

A TRV performs kd-tree traversal. We included four-entry short-stack memory [Horn et al. 2007] in each TRV. This feature has two benefits: a small SRAM size and stack overflow prevention without a limited tree depth.

A LIST searches the primitive list in a leaf node. This unit is necessary because a primitive can duplicate its reference in many leaf nodes. This occurs in spatial subdivisions such as kd-trees.

IST1s and the IST2 perform ray-triangle intersection tests. An IST1 performs a ray-plane test and a barycentric test, and the IST2 computes the final u, v, and t values of the hit point. The main goal of this separation is a reduction in area-expensive reciprocal units; an IST1 does not require reciprocal operations.

TRVs, LISTs, and IST1s require data fetching. For efficient memory access, we assigned a level-one (L1) cache memory and a ray accumulation buffer to each pipeline. Three level-two (L2) caches for TRVs, LISTs, and IST1s between off-chip memory and L1 caches reduce memory traffic.

The ratio of traversal to intersection testing is dependent on the ray and scene characteristics; but Schmittler et al. [2004] suggested that a ratio of 4:1 is reasonable. We therefore set 24 TRVs and six LISTs. Also, we assigned eight IST1s per core because an IST1 handles two operations (ray-plane and barycentric tests). Finally, we assigned only one IST2 per core because most triangles tested by IST1s do not intersect the ray and an IST2 only handles the hit triangles.

Ray generation and shading are processed by programmable shaders. Use of external DRAM as the interface between the T&I and the shader would be a bandwidth burden to the memory system. Hence, we configure on-chip FIFO (First In, First Out) buffers to store 128 rays for both input (ray generation to T&I) and output (T&I to shading), respectively. We assumed that the size of a ray is 128 bytes. Therefore, a total of 32 KB (256×128 bytes) of SRAM is required. To use these two buffers concurrently, programmable shaders should support concurrent kernel execution.

An RD gets a ray from the programmable shader and transfers the ray to the TRVs. The RD first calculates inverse direction vectors for traversal. It then performs an intersection test between the ray

and scene's axis-aligned bounding box (AABB) and gets the sceneMax value (the maximum t distance in the scene). Because only one ray can be supplied for each cycle, rays are supplied to the TRVs in a FIFO manner.

Table 1 illustrates how a ray moves from unit-to-unit. The numbers in Table 1 correspond to the circled numbers in Figure 1.

Table 1: *Data flow.*

Flow	From→To	Current state of the ray	Next process
1	RD→TRV	Detected free TRV stacks	Dispatching a new ray
2	TRV→TRV	Finished an inner node traversal	Going to the next node
3	TRV→LIST	Finished a leaf node traversal	Fetching the fist triangle index
4	LIST→IST1	Fetched the triangle index	Ray-plane test (Phase 1)
5	IST1→IST1	Passed the ray-plane test	Barycentric test (Phase 2)
6	IST1→TRV	Failed the barycentric test with the last triangle	Stack pop
		Passed the barycentric test (occlusion ray)	Freeing the stack
7	TRV→ Shader	Found the hit point or there are no hit points	End of the ray traversal
8	IST1→IST2	Passed the barycentric test (radiance ray)	Calculation of the final hit point values (Phase 3)
9	IST1→LIST	Failed either the ray-plane or barycentric test and subsequent triangles remain in the leaf	Fetching the subsequent triangle index
10	IST2→TRV	Found the nearest hit point (radiance ray)	Freeing the stack
11	IST2→LIST	Subsequent triangles remain in the leaf	Fetching the subsequent triangle index

4 Traversal with Ordered Depth-First Layout

4.1 Background

At each traversal step, node fetching and stack operations are required. These operations can cause significant memory traffic because a ray visits dozens of nodes to find its hit point. In order to alleviate this problem, we developed a cache-efficient layout called ODFL and applied the short-stack algorithm [Horn et al. 2007] to our traversal architecture.

Cache-efficient layouts can help reduce scene traffic. In kd-trees, a compact eight-byte depth-first layout (DFL) [Pharr and Humphreys 2010] is widely used for ray tracing because of its small memory footprint and the depth-first search manner of ray traversal. In this layout, the parent node and the left child node are adjacent, so they have parent-child locality. Therefore, this layout has only a single pointer for the right child node. In the DFL, the arrangement criterion of child nodes is a geometric position. We improve the DFL by utilizing parent-child locality.

A simple way to remove memory traffic for stack operations is the use of on-chip SRAM. However, this approach requires a large SRAM to prevent stack overflow. The short-stack algorithm can solve this problem. It uses a small, n-entry stack to maintain the last n pushes. If the stack is empty, the ray restarts the traversal on the restart node determined by the push-down method. Horn et al. [2007] reported that the overhead of short-stack was less than 3%.

4.2 Proposed method

In contrast to the original DFL, the ODFL uses surface area rather than the geometric position to arrange child nodes. This means that the child node with the larger surface area is stored next to its parent node. Because the probability of a ray intersecting with a node is proportional to its surface area, the probability that a ray accesses

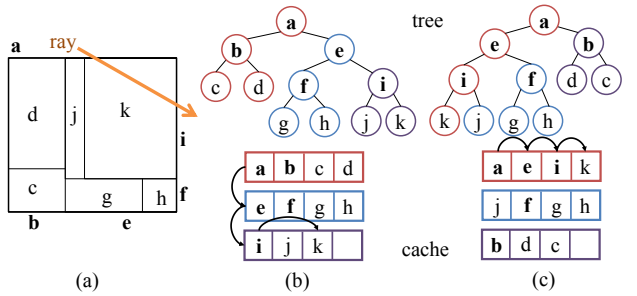

Figure 2: *(a) An example space, (b) original depth-first layout, and (c) ordered depth-first layout.*

the same cache line also increases. Figure 2 illustrates how our approach differs from the original DFL. For each case, leaf node k has the largest surface area compared with all other leaf nodes. If a ray visits leaf node k with the DFL (Figure 2-(b)), the ray accesses three cache lines. In contrast, the ray accesses only one cache line with the ODFL (Figure 2-(c)) because node k is stored in the same cache line as node a (root). Because most tree construction algorithms are based on the surface area heuristic (SAH) [MacDonald and Booth 1990], we used this surface area value with no overhead when determining the order of child nodes. We then add a one-bit flag to indicate whether the node is reversed in order from the original DFL. Because this flag can be embedded into an eight-byte node, our layout does not require additional memory space.

Tree traversal using the ODFL is almost the same as the DFL, aside from the use of the reorder flag. If the flag is true, the two child nodes are considered to be reversed in contrast to their geometric position. In front-to-back tree traversal, the reorder flag should be referenced.

Table 2 shows the kd-tree traversal pipeline with the ODFL and the short stack. We assumed that both a floating-point (FP) addition and multiplication requires two cycles, while an FP comparison requires a single cycle [Kopta et al. 2010]. We added a reorder flag-check routine in the fourth pipeline stage for the ODFL. This check routine requires only a one-bit NXOR operation. Also, this pipeline is based on the short-stack algorithm. A stack operation for on-chip SRAM can be performed in a single cycle. Note that in contrast to the pseudo-code in [Horn et al. 2007], we exploited the pre-computed inverse direction vector, as described in [Benthin 2006; Pharr and Humphreys 2010], to use a multiplier rather than a reciprocal unit. In summary, we assigned one FP adder, one FP multiplier, and three FP comparators for a traversal unit.

Table 2: *Kd-tree traversal pipeline. FP, INT, ADD, MUL, and CMP are the abbreviations for floating-point, integer, adder, multiplier, and comparator, respectively. The sign() function extracts a sign bit in an FP value. The pipelines in this table go from top to bottom.*

No	FP or INT ADD	FP MUL	FP CMP	Etc.
P1			tmax< sceneMax	check the hit result and flag
P2	so=node.split - ray.ori[axis])			stack allocation and free
P3				
P4		tplane = so * ray.invdir[axis]		leftChildFirst= (reorderFlag NXOR sign(so))
P5	leftChild = currentNode+1			
P6			tplane> tmax	sign (tplane) == 1
P7			tplane< tmin	
P8				stack push/pop

4.3 Comparison with other approaches

In Table 3, we compare our architecture with other traversal architectures. Our architecture has several benefits. First, our single-ray tracing approach is robust for incoherent rays in contrast to packet-based SIMD approaches. Second, the stack size is considerably reduced. Third, there are no reciprocal units due to the use of an inverse direction vector. Fourth, the ODFL can reduce cache misses.

Table 3: *Comparison to other traversal architectures. Throughput is the number of traversal steps per cycle.*

	SaarCOR [Schmittler et al. 2004]	RPU [Woop et al. 2005]	D-RPU [Woop 2007]	Ours
FP ADD	4	4	16	1
FP MUL	0	4	16	1
FP RCP	4	0	4	0
Stack entry	32	32	32	4
Peak throughput	4	4	4	1
Architecture	SIMD	SIMD	SIMD	MIMD
AS	kd-tree	kd-tree	B-KD tree	kd-tree
Special feature			node update	ODFL

5 Three-Phase Intersection Test Unit

5.1 Background

The majority of the ray-triangle intersection tests involve three-phase calculations, as illustrated in Figure 3. First, a ray-plane test determines whether the ray intersects with the triangle within the t intervals. After passing this test, a barycentric test is performed. This test checks barycentric coordinates (u, v) to determine whether the hit point is inside or outside of the triangle. If the ray passes these two tests, the final t, u, and v values can be calculated. This final phase divides intermediate values ($dett$, $detu$, and $detv$) calculated from the former two phases by the determinant value in Cramer's rule. Note that some algorithms [Wald 2004; Woop 2004; Kensler and Shirley 2006] change the order of these three phases; they execute this reciprocal earlier than the other processes.

Multi-phase intersection test algorithms result in early termination for the test of non-intersecting triangles. Previous one-phase intersection hardware architectures [Schmittler et al. 2004; Woop et al. 2006a; Kim et al. 2007] have not exploited this feature. In other words, the calculation costs of an intersecting triangle and a non-intersecting triangle are the same. To solve this problem, we implemented a three-phase intersection test for our proposed architecture.

To determine the optimal algorithm for our architecture, we analyzed the computational costs of various intersection algorithms (Table 4). In the early t version, which is optimized in this paper, the ray-plane test (t test) is executed first, by multiplying the t interval by the determinant value. According to Table 4, the early t version of Shevtsov et al.'s method [2007] has the lowest costs for non-intersecting triangles and Wald's method [2004] has the lowest costs for intersecting triangles. Because most of the tested triangles are not intersected by the ray, we chose the former algorithm.

Although Möller and Trumbore's method [1997] has advantages in terms of memory footprint and the cost of pre-computation, we did not choose it because its computational cost is too high and it requires fetching three vertices. The latter feature is not suitable for fully pipelined architectures because expensive three-port SRAM is required.

5.2 Proposed method

The proposed method consists of two kinds of architecture. First, the ray-plane and barycentric tests are performed in a common unit

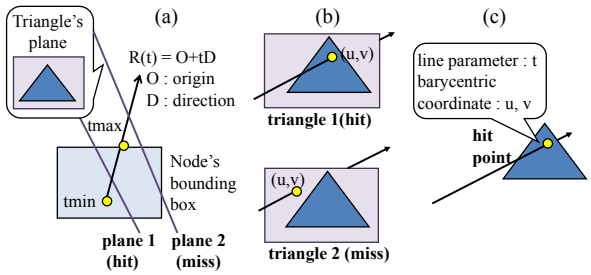

Figure 3: *(a) Ray-plane test, (b) barycentric test, and (c) final hit point calculation.*

Table 4: *Comparisons of intersection algorithms. MUL, RCP, tri, and vtx are abbreviations for multiplication, reciprocal, triangle, and vertex, respectively. (a), (b), and (c) are the same as in Figure 3. Pre-computation is performed either per triangle or per ray packet. In single-ray tracing, however, the latter type of pre-computation is performed per ray during the intersection test. The pre-compation cost of each algorithm can be varied by calculation methods.*

Algorithm	Computational cost (a)	(a)+(b)	(a)+(b)+(c)	Precom-putation	Required memory size
[Möller and Trumbore 1997]	MUL 24 ((b) only)		MUL 27 RCP 1	X	12B per tri 12B per vtx
(early t version)	MUL 20	MUL 26	MUL 29 RCP 1		
[Wald 2004]	MUL 5 RCP 1	MUL 11 RCP 1	MUL 11 RCP 1	MUL 22 RCP 2	36-48B per tri
[Woop 2004]	MUL 7 RCP 1	MUL 21 RCP 1	MUL 21 RCP 1	MUL 54 RCP 1	12B per tri 12B per vtx
[Kensler and Shirley 2006]	MUL 7 RCP 1	MUL 21 RCP 1	MUL 21 RCP 1	MUL 6	12B per tri 12B per vtx
[Benthin 2006]	MUL 9 ((b) only)		MUL 15 RCP 2	MUL 27	12B per tri 12B per vtx
(early t version)	MUL 4 RCP 1	MUL 13 RCP 1	MUL 15 RCP 2		
[Shevtsov et al. 2007]	MUL 12 ((b) only)		MUL 15 RCP 1	MUL 14 RCP 1	36-48B per tri
(early t version)	MUL 6	MUL 14	MUL 17 RCP 1		
[Havel and Herout 2010] non-SSE version	MUL 7	MUL 21	MUL 24 RCP 1	MUL 36 RCP 1	48B per tri

(IST1) with two modes. Because the two tests have similar computational costs and typically require triangle data fetching, this common unit is useful. Tables 5 and 6 depict the pipeline stages of the IST1 unit. Next, the last phase is performed in a separate unit (IST2) due to the need for a reciprocal and non-data fetching (Table 7). In this IST2 unit, an FP reciprocal requires 16 cycles. The definitions of the variables in Tables 5-7 are as follows: nu, nv, and np are normal data. pu and pv are vertex data. ci is the projection axis. $e0u$, $e0v$, $e1u$, and $e1v$ are edges data. ou, ov, and ow are the ray's origin. du, dv, and dw are the ray's direction.

With this three-phase strategy, we can markedly reduce the required arithmetic units for intersection tests. The number of floating-point units in an IST1 comprises only seven adders, seven multipliers, and three comparators while the number of floating-point units in an IST2 includes only one reciprocal unit and three multipliers. There is only one IST2 per T&I core, thus a T&I core includes only one reciprocal unit for intersection tests.

Our strategy is also effective for reducing memory access. First, we assumed that the 2-bit ci value could be embedded into other values. We then only allotted 16 bytes (np, nu, nv, and pu) for a ray-plane test. If the ray does not pass this test, then fetching of other data for the barycentric test in the triangle is not required.

Table 5: *Pipeline stages of the first intersection test unit in mode 0 (ray-plane test). The pipelines in this table go from top to bottom.*

No	FP ADD	FP MUL	FP CMP	Etc.
P1-2	dett=np-ow	dvnv=dv*nv dunu=du*nu		
P3-4	det=dunu+dw	ounu=ou*nu		
P5-6	dett=dett-ounu det=det+dvnv	ovnv=ov*nv		
P7-8		tmaxdet=tmax*det tmindet=tmin*det		
P9-10	dett=det-ovnv			
P11-12		dudett=du*dett	tmaxdet <=dett	N/A
P13-14	puou=pu-ou			
P15-16	N/A		tmindet >=dett	N/A
P17-18			N/A	

Table 6: *Pipeline stages of the first intersection test unit in mode 1 (barycentric test). The pipelines in this table go from top to bottom.*

No	FP ADD	FP MUL	FP CMP	Etc.
P1-2	pvov=pv-ov	dvdett=dv*dett puoudet=puou*det		
P3-4	Du=dudett -puoudet	pvovdet=pvov*det		
P5-6	Dv=dvdett -pvovdet	e1vDu = e1v*Du		
P7-8		e1uDv = e1u*Dv e0uDv = e0u*Dv		
P9-10	detu=e1vDu -e1uDv			
P11-12		e0vDu = e0v*Du	detu <=det	sign(det) ==sign(detu)
P13-14	detv= e0uDv-e0vDu			
P15-16	tmpdet0 =detu+detv		detv <=det	sign(det) ==sign(detv)
P17-18			tmpdet0 <=det	

Table 7: *Pipeline stages of the second intersection test unit.*

No	FP MUL	FP RCP
P1-16		rdet = 1.0f / det
P17-18	t=dett*rdet, u=detu*rdet, v=detv*rdet	

Our method permits effective configuration. When arranging triangle data to memory, the separation of data for the intersection test and for shading is effective in terms of data fetching [Wald et al. 2001]. In a fully pipelined architecture for intersection tests, all data for the intersection test of a triangle should be stored in a cache line since this architecture requires data fetching in each cycle. The data layout of algorithms requiring pre-computation [Wald 2004; Shevtsov et al. 2007; Havel and Herout 2010] uses 48 bytes per triangle. Because 48 is not a power of two, Benthin et al. [2006] stored normal and shader indices, along the data for the intersection, in a 64-byte cache line. This combination reduces cache efficiency at the intersection test stage, as 1/4 of the data in a cache is unnecessary for the intersection test. In contrast, our three-phase intersection does not need to fetch the entire 48 bytes each cycle. Thus, only intersection data is stored in a cache.

5.3 Comparison with other approaches

In Table 8, we compare our three-phase architecture with other one-phase dedicated intersection architectures. Our architecture greatly reduces the number of arithmetic units compared with the other architectures. Thus, we can conclude that our three-phase architecture has a higher performance/area than previous approaches.

Table 8: *Comparison to other intersection architectures. Throughput is the number of intersection tests per cycle. Because the ratio of IST1 to IST2 is 8:1, we divided the number of arithmetic units of IST2 by 8. The throughput of 0.76 for our architecture obtained by assuming that 70% of triangles are filtered in Phase 1.*

	SaarCOR	D-RPU	CDE	Ours	
	[Schmittler et al. 2004]	[Woop 2007]	[Kim et al. 2007]	IST1	IST2
FP ADD	12	17	12	7	0.375
FP MUL	11	21	27	7	0.375
FP RCP	1	1	1	0	0.125
Throughput	0.8	0.5	1.0	0.76	
Algorithm	[Wald 2004]	[Möller and Trumbore 1997]		[Shevtsov et al. 2007]	

6 Ray Accumulation Unit for Latency Hiding

6.1 Background

Each of the traversal and intersection steps in ray tracing requires memory access to obtain the shape data. In other words, ray tracing is not only computation-intensive, but also memory-intensive. Thus, efficient memory latency hiding techniques are required for fast ray tracing.

Existing GPUs hide memory latency using hardware multithreading [Fatahalian and Houston 2008]. This method is concurrently executed from several hundred threads. For example, the Fermi GPUs can perform 48 warps (=1536 threads) per streaming multiprocessor (SM). In GPU ray tracing, a ray is treated as a thread. Even if some threads stall, others can be executed during the next cycle. In many cases, this method can efficiently hide long-latency operations, such as global memory access or texture fetching.

Although the hardware multithreading in GPUs is enabled, memory traffic can be the primary factor limiting the performance of incoherent ray tracing [Aila and Laine 2009]. We expect that this occurs because existing GPUs' multithreading is processed on wide SIMD architectures. For example, NVIDIA GPUs concurrently execute 32 threads in a warp. If rays are coherent, the threads may request broadcast access or coalesced access. Thus, hardware multithreading can be effective in this case. However, if rays are incoherent, each thread in a warp may request different shape data. If one lane misses at that time, all other lanes will wait. This situation prevents the rays from exploiting temporal locality because the possibility of evicting useful data in the cache may increase during the waiting period.

The temporal locality of each ray can be explained as follows. If a ray accesses a cache line, the ray is likely to access the cache line in the near future. This is because the child, grandchild, or sibling nodes of a particular node, the elements of a primitive list, or the data of adjacent primitives can be stored in the same cache line. As mentioned earlier, the hardware multithreading on wide SIMD architectures may not effectively exploit this locality due to increased cache miss.

Even if the shape data is fetched by global memory access without cache (e.g., triangles in [Aila and Laine 2009]), a similar problem occurs in incoherent ray tracing. Because many threads request random access instead of coalesced access in this case, the memory bandwidth can be saturated. In this case, the hardware multithreading in GPUs cannot efficiently hide memory latency.

6.2 Proposed method

In this section, we propose specialized hardware multithreading for ray tracing. If a ray causes cache miss, it should wait until the shape data is fetched. In this case, the ray is stored in a small ray

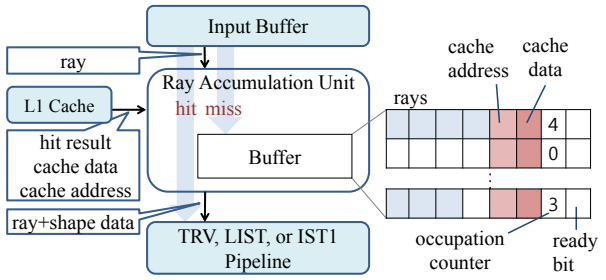

Figure 4: *Overall architecture of the ray accumulation unit.*

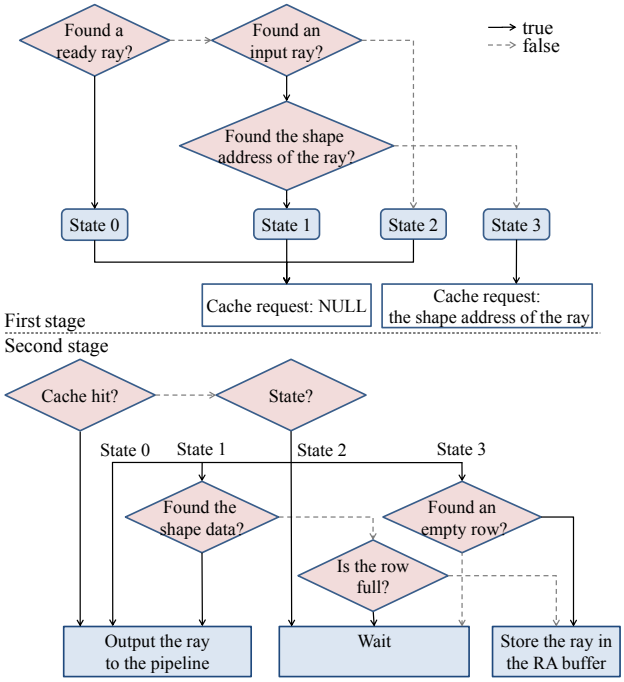

Figure 5: *Operational flow of the ray accumulation unit.*

accumulation (RA) buffer. Other rays can be processed during this period, so the long memory latency can be effectively hidden.

Our approach has a similar purpose to the instruction execution mechanism used in data flow architectures [Dennis 1980]. Both our approach and data flow scheduling aim to achieve a continuous process stream without waiting for the previous process to be completed. A different feature of our method compared with data flow scheduling is the RA buffer configuration. In our approach, the rays that reference the same cache line are accumulated in the same row in an RA buffer. When the shape data requested by the accumulated rays is fetched from the L2 cache, these rays with the shape data are transferred from the RA buffer to the operation pipeline. Note that a non-blocking cache is needed to support our method.

Figure 4 depicts the overall architecture of an RA unit. An RA unit consists of control logics and buffer spaces. Each RA unit is located between an input buffer, an L1 cache, and an operation pipeline (e.g., TRV, LIST, and IST1) and fetches shape data from L2 cache/memory.

In Figure 4, each row has four rays, a cache address (27 bits), cache line data (32 bytes), a ray counter (four bits), and a ready bit. A ray counter includes the number of rays in the row. Because a cache address does not include the cache line offset (five bits), a ray counter and a valid bit can be embedded within these five bits. We assume that the buffer size in our hardware is 4×8, and thus, a buffer can store up to 32 rays. This size was determined experimentally (see Section 7.3), but all sizes (e.g., the number of stored rays and cache line size) can be varied by a hardware designer.

A ready bit represents the existence or absence of "ready rays" in the row. When the cache line data in the row is transferred from the cache, all rays in the row are ready to be output to the operation pipeline. If there are ready rays in the buffer, the buffer preferentially outputs those rays to the operation pipeline. If and only if the number of ready rays in the row is greater than 0, the ready bit is 1.

Figure 5 illustrates the operational flow of the RA unit. This flow is divided into two steps: before a cache request and after a cache response. There are three states in the first stage and five states in the second stage.

In the first stage, the RA unit determines the output states for the second stages and the cache request address. If there are rows with a ready bit of 1, the RA unit sets the output state to 0. Otherwise, the RA unit checks whether an input ray has been transferred from the input buffer. If an input ray exists, the RA unit checks whether the shape address required by the input ray exists in the RA buffer. If so, the output state is 1. If not, the output state is 3. If an input ray does not exist, then the output state is 2. Output states of 0, 1, and 2 indicate that a cache request is NULL. If the output state is 3, the RA unit makes a cache request using the required shape address specified by the input.

In the second stage, the RA buffer identifies the output data using the state determined in the first stage. There are three possible cases. In the first case, the RA unit sends a ray to the operation pipeline. If requested data is contained in the cache (cache hit), the output data is a combination of a ray from either the input buffer or the RA buffer and shape data from the cache. If cache miss occurs but ready rays remain in the RA buffer (state 0), ray data and shape data are obtained from the RA buffer. If the state is 1 and the shape data required by an input ray exists in the RA buffer, the output data is a combination of the ray from the input buffer and the shape data from the RA buffer. In the second case, there is no output data and no content changes in either the input or RA buffers. This occurs when there is no empty space in the row (state 1), no input rays (state 2), or no empty rows in the RA buffer (state 3). In the third case, the state is 3 and an empty row exists. Thus, a ray from the input buffer is successfully stored in the RA buffer.

The main differences between existing hardware multithreading and the proposed method are related to the working set size and the exploitation of temporal locality. First, the buffer space of an RA unit is small; a 32-entry RA buffer requires 4 KB of memory. Next, each ray can effectively exploit its temporal locality because the period between the shape data fetching of the ray is much shorter than existing hardware multithreading. This feature results from the small working-set size of our architecture.

7 Simulation Results and Analysis

7.1 Setup

To verify the proposed architecture, we created a cycle-accurate simulator. This simulator collects scene and ray data from the files and simulates the execution of our architecture. After simulation, this simulator provides the total number of cycles used to generate a scene, hardware utilization, cache statistics, expected performance, and hit results.

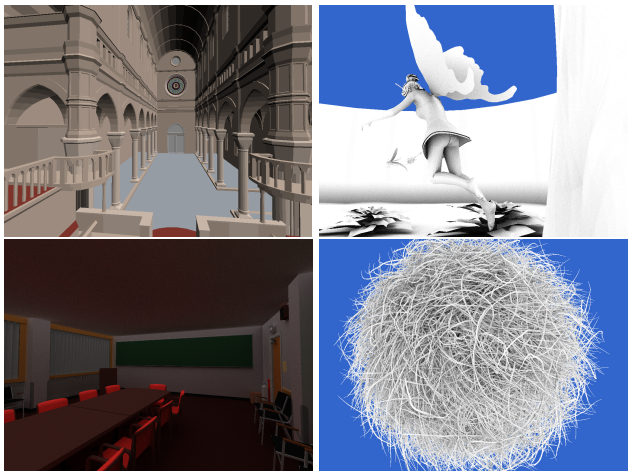

Figure 6: *Test scenes: Sibenik with ray casting, Fairy with AO, Conference with path tracing, and Hairball with AO.*

The software setup was structured as follows. When we constructed kd-trees, we used the SAH [MacDonald and Booth 1990] with on-the-fly pruning [Soupikov et al. 2008]. We chose four test scenes (Figure 6): Sibenik (80 K triangles), Fairy (174 K triangles), Conference (282 K triangles), and Hairball (2.9 M triangles). To examine the influence of the ray coherence on performance, we created three different types of ray data from Aila's ray tracer [Aila and Laine 2009]: a primary ray, an ambient occlusion (AO) ray, and a diffuse inter-reflection ray. Primary rays are extremely coherent. In contrast, AO and diffuse inter-reflection rays are incoherent because they are distributed according to a Halton sequence on the hemisphere. The number of samples per pixel was 32. AO rays were terminated by the cut-off value; we used AO cut-off values of 5.0, 0.3, 5.0, and 0.3 for the Sibenik, Fairy, Conference, and Hairball scenes, respectively. All scenes were rendered at a 1024×768 resolution. The performance values are averages from five representative viewpoints per scene except for the Hairball scene. This setup is the same as that in [Aila and Laine 2009] (Sibenik, Fairy, and Conference) and [Laine and Karras 2010] (Hairball) except for the type of acceleration structure.

The hardware setup was structured as follows. We supplied rays to the TRV units from the RD unit when the number of executing rays in a T&I core was less than 480 (24 TRVs×20 stacks per TRV). We also adopted a GDDR3 memory simulator from GPGPU-Sim [Bakhoda et al. 2009] in order to execute accurate memory access. The settings of GDDR3 were as follows: 1 GHz clock, 8 channels, tCL=9, tRP =13, tRC=34, $tRAS$=21, $tRCD$=12, and $tRRD$=8. These settings are similar to those of GTX280 (65 nm GPU) and [Mahesri et al. 2008]. We used the First-Ready First-Come-First-Serve method [Rixner et al. 2000] for effective memory access scheduling. The latency of the L1 and L2 caches was set to 1 and 20 cycles, respectively.

7.2 Hardware complexity and area estimation

Table 9 depicts the hardware complexity of a T&I core. As mentioned in Section 3.1, this T&I core is fully pipelined and independently executed. If a ray requires 48 traversal steps and 12 intersection steps, the peak throughput of a T&I core in this case is 0.5 rays per cycle.

We assigned an independent L1 cache for each TRV, LIST, and IST unit. Each TRV, LIST, and IST1 unit has an 8 KB 2-way, 4 KB 2-way, and 16 KB 4-way set associative cache, respectively. Because traversal and intersection tests have read-only access patterns, each independent cache requires only a single read port. We also assigned a 128 KB 2-way TRV L2 cache, a 32 KB 2-way L2 cache, and a 128 KB 2-way IST1 L2 cache per T&I core. Each TRV, LIST, and IST1 L2 cache has eight, four, and eight banks, respectively, for multiple requests. All caches have a line size of 32 B. Register files (RF) are needed for input buffers, output buffers, RA buffers, traversal stacks, I/O buffers to programmable shaders, and pipeline registers.

Table 9: *Complexity of a T&I core measured by the number of floating-point units and the required on-chip memory.*

	ADD	MUL	RCP	CMP	RF	L1 Cache	L2 Cache
1 RD	6	9	1	12	2 KB		
24 TRV	24	24		72	271 KB	192 KB	128 KB
6 LIST					43 KB	24 KB	32 KB
8 IST1	56	56		24	117 KB	128 KB	128 KB
1 IST2		3	1		9 KB		
I/O buffer					32 KB		
Total	86	92	2	108	476 KB	344 KB	288 KB

Table 10: *Area estimates of a T&I core.*

Functional Unit	Area (mm^2)	Total Area (mm^2)	Memory Unit	Area (mm^2)	Total Area (mm^2)
FP ADD	0.003	0.26	TRV L1	0.030	0.73
FP MUL	0.01	0.92	LIST L1	0.028	0.17
FP RCP	0.11	0.22	IST1 L1	0.082	0.66
FP CMP	0.00072	0.08	TRV L2		0.64
INT ADD	0.00066	0.01	LIST L2		0.24
Control/Etc.		0.35	IST1 L2		0.64
			4 K RF	0.019	2.26
Wiring overhead					4.95
Total					12.12

To predict the performance of our T&I engine, we carefully estimated the area of our architecture. First, we assumed that the hardware specifications were similar to TRaX [Spjut et al. 2009]: 65 nm technology, a 200 mm^2 die size, and the 500 MHz clock. Unlike TRaX, however, our architecture separates T&I cores and programmable shader cores. As with D-RPU [Woop 2007], we assumed that T&I cores occupy less than 29.5% of the total die area. The remaining area was assigned to programmable shaders and memory interfaces. Second, we used the area estimates for arithmetic units and caches using the 65 nm library from [Kopta et al. 2010] and the CACTI 6.5 tool [Muralimanohar et al. 2007], respectively.

Table 10 summarizes the area estimation of a single T&I core. Stack operations, pipeline controls, RA buffer controls, and other controls require hardware resources. We estimated the area of this part to be 23.3% of the total area for arithmetic units. We obtained this percentage from the ratio of the front-end area to that of the execution area in [Mahesri et al. 2008]. We think this assumption is conservative because control parts of our architecture do not need instruction fetching and decoding in contrast to the architecture in [Mahesri et al. 2008]. Also, wiring overhead should be taken into, due to place and route, choice of logic gates, and other optimizations. Woop [2007] and Mahesri et al. [2008] used 29-33% wiring overhead. In contrast, our estimation requires two levels of wiring overhead (arithmetic units → TRV/LIST/IST units → a T&I core), so we added 69% (1.69 is the square of 1.3) wiring overhead into our area estimation. Finally, we concluded that a T&I core occupies a 12.12 mm^2 area with 65 nm technology.

One of our assumptions was that the maximum area for T&I cores is less than 29.5% of the total area, as mentioned before. Hence, four T&I cores (48.50 mm^2) can be allocated in a 200 mm^2 die. We believe that this area ratio is suitable for preventing performance bottlenecks caused by shaders.

Table 11: *Simulation results. TRV steps include the number of stack allocation/free/pop operations.*

Ray type	TRV/LIST/IST1/IST2 utilization (%)	TRV/LIST/IST1 L1 cache hit (%)	TRV/LIST/IST1 L2 cache hit (%)	Average TRV/IST steps per ray	Memory traffic (GB/s)	Memory traffic reduction by the ODFL (%) (L1→L2 / L2→Main)	Simulated performance (Mrays/s)
Sibenik (80 K triangles)							
Primary	67 / 69 / 62 / 24	99 / 99 / 99	93 / 76 / 77	72 / 21	0.9	1.3 / 0.1	461
AO	87 / 65 / 56 / 00	98 / 99 /99	99 / 88 / 91	52 / 10	1.3	1.9 / 15.1	819
Diffuse	70 / 66 / 60 / 24	89 / 95 / 89	87 / 53 / 69	78 / 18	27.1	0.5 / 3.8	437
Fairy (174 K triangles)							
Primary	85 / 55 / 53 / 21	97 / 99 / 98	91 / 71 / 71	127 / 22	3.3	5.7 / 1.6	363
AO	83 / 60 / 53 / 00	97 / 99 / 97	95 / 69 / 79	52 / 09	4.0	1.7 / 3.6	813
Diffuse	66 / 51 / 48 / 16	91 / 96 / 90	82 / 51 / 62	91 / 18	25.9	7.5 / 1.8	372
Conference (282 K triangles)							
Primary	70 / 84 / 78 / 43	99 / 99 / 99	92 / 74 / 74	53 / 18	3.5	0.6 / 1.4	792
AO	81 / 71 / 62 / 00	98 / 99 / 97	96 / 76 / 93	34 / 07	1.7	2.1 / 2.4	1188
Diffuse	57 / 63 / 61 / 35	92 / 96 / 88	87 / 58 / 64	47 / 13	22.6	3.8 / 1.9	605
Hairball (2.9 M triangles)							
Primary	83 / 68 / 64 / 07	93 / 98 / 92	92 / 72 / 76	216 / 43	12.6	4.6 / 2.9	186
AO	47 / 59 / 56 / 00	87 / 95 / 86	84 / 59 / 65	46 / 14	30.5	3.6 / 4.6	503
Diffuse	12 / 14 / 13 / 01	76 / 91 / 70	60 / 39 / 37	133 / 38	31.6	3.7 / 3.4	44

Table 12: *Performance comparison for the Conference scene against previous approaches.*

		CPU	GPU		Many-core	Ray Tracing H/W			
		Intel X7460 × 4 [Tsakok 2009]	NVIDIA GTX285 [Aila and Laine 2009]	NVIDIA GTX480 [Aila and Laine 2009]	Intel MIC [Wald 2010]	D-RPU [Woop 2007]	RTE [Davidovic et al. 2011]	MIMD TM [Kopta et al. 2010]	T&I Engine (ours)
Mrays/s	(Primary)		142	253	211[a]	122		387	792
	(AO)		134	269			117		1188
	(Diffuse)	60	60	121				355	605
Process (nm)		45	55	40	45	90	90	65	65
Number of cores		24	240	480	32	8	1	80	4 (T&I)
Area (mm^2)		503 × 4	470	529	-	186	15	175	48 (T&I), 200 (total)
Clock (MHz)		2666	648 (core) 1476 (shader)	700 (core) 1401 (shader)	1000	400	2000	1000	500

[a] 1920 (width) × 1200 (height) × 46 (FPS) × 2 (one primary ray + one shadow ray).

7.3 Simulation results

Before the simulation, we conducted an experiment result to measure the optimal size of an RA buffer (Figure 7). We changed the row height and column width in the RA buffer. When we increased the size of the RA buffer, performance increased due to increased unit utilization. However, it also increased the total area due to additional registers. Based on the results, 4×8 was considered the optimal size. Large RA buffers including 64-128 elements only provided slight performance improvements, but required significant additional area.

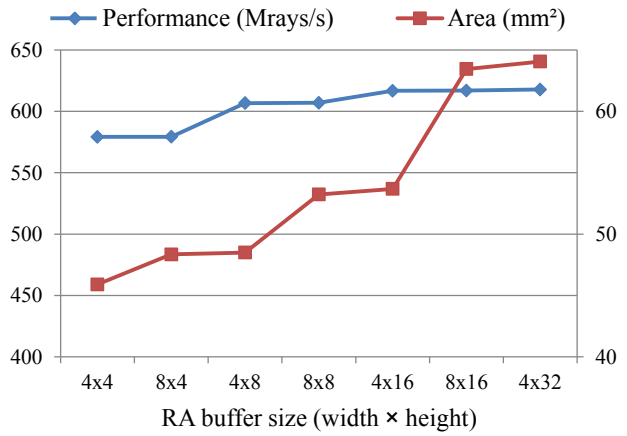

Figure 7: *Trade-off between performance and area by changing the size of the RA buffer. In this experiment, we used the conference scene with the viewpoint 1 and diffuse inter-reflection rays.*

Table 11 summarizes the simulation results. Our T&I engine achieved 44-1188 Mrays/s on four T&I cores. Table 12 shows that our architecture performed better than existing ray tracing platforms. This improved performance was due to the optimized algorithms [Horn et al. 2007; Shevtsov et al. 2007; Soupikov et al. 2008; Pharr and Humphreys 2010], fixed logic, MIMD architecture, and the three novel concepts described in this paper.

The AO rays exhibited much better performance than other types of rays. This can likely be explained by early termination. First, if the t value is higher than the cut-off, the traversal of the AO ray is terminated. Second, AO rays are treated as shadow rays, so they terminate their traversal as soon as they find a hit primitive. The reason that the AO rays results indicated zero utilization of IST2 units is this "any hit" termination process. Thus, the fewer traversal and intersection steps were required of an AO ray compared with those of other types of rays requiring the closest hit.

The diffuse inter-reflection ray performance was limited by memory bandwidth. In random access, DRAM utilization is approximately 30% [Rixner et al. 2000; Bakhoda et al. 2009]. Although 8-channel GDDR3 provides a peak bandwidth of 128 GB/s, we can only exploit 30-40 GB/s because the cache miss data makes random access patterns. According to the results in Table 11, the diffuse inter-reflection rays showed lower cache hit ratios than other types of rays. These cache misses increased memory traffic. Memory bandwidth usage increased by up to 31 GB/s and T&I utilization decreased due to the memory latency. Nevertheless, the performance of our memory system is comparable that of a GPU-based architecture for incoherent ray tracing [Aila and Karras 2010]. In the Hairball scene, our architecture required a bandwidth of 31 GB for 44 M inter-reflection rays and the architecture in [Aila and Karras 2010] required a bandwidth of 2.5 GB for 3 M inter-reflection rays.

We predict that a higher memory bandwidth with GDDR5 can alleviate the memory bottleneck of our architecture. We also suspect that GPU-based ray reordering [Garanzha and Loop 2010] can increase path tracing performance of our architecture. Garanzha and Loop [2010] reported that the GPU path tracing performance increased up to 1.9× when ray sorting was enabled.

The ODFL only slightly reduced memory traffic and this finding is different from the results in [Nah et al. 2010] (a gain in bandwidth of 16-30%). The main reason for this is that we used a smaller cache line size (32 B) than they did (64 B). The importance of exploiting parent-child locality is proportional to the cache line size. Another reason is that we measured memory traffic in TRVs and LISTs because a tree layout changes the order of the primitive list. The ODFL does not positively affect the primitive list because fetching of the list does not involve parent-child locality. Finally, the RA unit in our hardware absorbed some cache misses because the rays that referenced the same cache line were accumulated together in the RA buffer.

Although the ODFL had few benefits in our hardware, we think the ODFL has some promise. The benefits of the ODFL may be magnified on processors with large cache lines (64-128 B). Also, the use of the ODFL requires very small overhead for both tree traversal (a one-bit NXOR operation) and tree construction (simple ordering).

Finally, although we assigned sufficient area to programmable shaders (as described in Section 7.2), ray generation and shading (RGS) can act as a bottleneck. According to a state-of-the-art GPU ray tracer [Garanzha and Loop 2010], RGS performance on GTX285 was 774 Mrays/s on the Conference scene (17 ms elapsed for 133 M rays). If the simulated T&I performance is higher than 774 Mrays/s, then RGS is obviously a bottleneck. However, we expect that higher RGS performance can be achieved for two reasons. First, we fetch input rays and store output rays using FIFO buffers instead of off-chip global memory. Second, modern GPUs (e.g., GTX580) provide a higher computing power capacity than the GTX285. Thus, we think that ray generation and 'simple' Phong shading do not limit overall rendering performance.

8 Conclusions, Limitations, and Future Work

In this paper, we proposed a novel hardware architecture for efficient tree traversal and intersection tests, which we called the T&I engine. This architecture includes three novel concepts: the tree traversal unit with ordered depth-first layout, the three-phase intersection test unit, and the ray accumulation unit for hiding memory latency. Simulation results using a cycle-accurate simulator indicated that our architecture performs better than other approaches.

Our architecture has certain limitations. First, our architecture limits the primitive type to a triangle. In order to support other primitive types, primitives should be converted into triangles, such as in rasterization. Another potential solution is for programmable shaders to take charge of ray-primitive intersection tests, as with RPU [Woop et al. 2005], but this method has poor intersection performance. In our future work, we plan to extend our intersection test unit to support various primitive types.

Second, we focused on ray tracing of static scenes in this paper. In dynamic scenes, acceleration structures must be updated in each frame. If acceleration structures are constructed on CPUs and transferred to the T&I cores, the bandwidth of a PCI Express bus can be a bottleneck. If programmable shaders on GPUs are used for acceleration structure construction, dozens of milliseconds would be required, as described in [Hou et al. 2011]. Thus, we hope to develop special hardware architecture for tree construction.

Third, complex shading can be a major cost, although we expected that simple shading would not limit overall performance in Section 7.3. We think that shading filter stacks [Gribble and Ramani 2008] could be a solution to facilitate the concurrent use of various material shaders.

Finally, our architecture was only verified with cycle-accurate simulations. In the near future, we will perform ASIC verification and carefully investigate power consumption and temperatures.

Acknowledgements

This work was supported by Samsung Electronics Co., Ltd. Sibenik Cathedral model courtesy of Marko Dabrovic, Fairy forest model courtesy of Ingo Wald, Conference room model courtesy of Greg Ward, and Hairball model courtesy of Samuli Laine. We thank anonymous reviewers for constructive comments.

References

AILA, T., AND KARRAS, T. 2010. Architecture considerations for tracing incoherent rays. In *HPG' 10: Proceedings of the Conference on High Performance Graphics*, 113–122.

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *HPG '09: Proceedings of the Conference on High Performance Graphics*, 145–149. http://www.tml.tkk.fi/ timo/HPG2009/index.html.

BAKHODA, A., YUAN, G., FUNG, W., WONG, H., AND AAMODT, T. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software 2009*, 163–174.

BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. 2006. Ray tracing on the cell processor. In *Proceedings of the 2006 IEEE/EG Symposium on Interactive Ray Tracing*, 15–23.

BENTHIN, C. 2006. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Sarrland University.

CAUSTIC GRAPHICS. 2009. Introduction to CausticRT. Tech. rep. http://www.caustic.com/pdf/Introduction to CausticRT.pdf.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, 137–145.

DAVIDOVIC, T., MARSALEK, L., AND SLUSALLEK, P. 2011. Performance considerations when using a dedicated ray traversal engine. In *WSCG 2011 Full Paper Proceedings*, 65–72.

DENNIS, J. 1980. Data flow supercomputers. *Computer 13*, 11, 48 –56.

FATAHALIAN, K., AND HOUSTON, M. 2008. GPUs: A closer look. *ACM Queue 6*, 2, 18–28.

GARANZHA, K., AND LOOP, C. 2010. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum 29*, 2, 289–298. http://garanzha.com/Documents/GPU-RayTracing.ppt.

GOVINDARAJU, V., DJEU, P., SANKARALINGAM, K., VERNON, M., AND MARK, W. R. 2008. Toward a multicore architecture for real-time ray-tracing. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 176–187.

GRIBBLE, C., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 59–66.

HAVEL, J., AND HEROUT, A. 2010. Yet faster ray-triangle intersection (using SSE4). *IEEE Transactions on Visualization and Computer Graphics 16*, 3, 434–438.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, 167–174.

HOU, Q., SUN, X., ZHOU, K., LAUTERBACH, C., AND MANOCHA, D. 2011. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 3, 466–474.

KENSLER, A., AND SHIRLEY, P. 2006. Optimizing ray-triangle intersection via automated search. In *Proceedings of the 2006 IEEE/EG Symposium on Interactive Ray Tracing*, 33–38.

KIM, S.-S., NAM, S.-W., KIM, D.-H., AND LEE, I.-H. 2007. Hardware-accelerated ray-triangle intersection testing for high-performance collision detection. *Journal of WSCG 15*, 17–24.

KOPTA, D., SPJUT, J., BRUNVAND, E., AND DAVIS, A. 2010. Efficient MIMD architectures for high-performance ray tracing. In *ICCD 2010: Proceedings of the 28th IEEE International Conference on Computer Design*, 9–16.

LAINE, S., AND KARRAS, T. 2010. Two methods for fast ray-cast ambient occlusion. *Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering 2010) 29*, 4, 1325–1333.

MACDONALD, D. J., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3, 153–166.

MAHESRI, A., JOHNSON, D., CRAGO, N., AND PATEL, S. J. 2008. Tradeoffs in designing accelerator architectures for visual computing. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 164–175.

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

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 3–14.

NAH, J.-H., PARK, J.-S., KIM, J.-W., PARK, C., AND HAN, T.-D. 2010. Ordered depth-first layouts for ray tracing. In *ACM SIGGRAPH ASIA 2010 Sketches*, 55:1–55:2.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010) 29*, 4, 1–13.

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering*, second ed. Morgan Kaufmann.

RIXNER, S., DALLY, W., KAPASI, U., MATTSON, P., AND OWENS, J. 2000. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, 128–138.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 95–106.

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 (Proceedings of SIGGRAPH 2008) 27*, 3, 18:1–18:15.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Ray-triangle intersection algorithm for modern CPU architecture. In *Proceedings of GraphiCon 2007*, 33–39.

SOUPIKOV, A., SHEVTSOV, M., AND KAPUSTIN, A. 2008. Improving kd-tree quality at a reasonable construction cost. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing*, 67–72.

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 of Integrated Circuits and Systems 28*, 12, 1802–1815.

TSAKOK, J. A. 2009. Faster incoherent rays: Multi-BVH ray stream tracing. In *HPG '09: Proceedings of the Conference on High Performance Graphics*, 151–158.

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

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1, 6:1–6:18.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Sarrland University.

WALD, I. 2010. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics*. (to appear).

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

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*, 3, 434–444.

WOOP, S., BRUNVAND, E., AND SLUSALLEK, P. 2006. Estimating performance of a ray-tracing ASIC design. In *Proceedings of the 2006 IEEE/EG Symposium on Interactive Ray Tracing*, 7–14.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 67–77.

WOOP, S. 2004. *A Ray Tracing Hardware Architecture for Dynamic Scenes*. Master's thesis, Sarrland University.

WOOP, S. 2007. *A Programmable Hardware Architecture for Realtime Ray Tracing of Coherent Dynamic Scenes*. PhD thesis, Sarrland University.