RayCore: A ray-tracing hardware architecture for mobile devices

JAE-HO NAH
Sejong University and University of North Carolina at Chapel Hill
and
HYUCK-JOO KWON and DONG-SEOK KIM
Sejong University
and
CHEOL-HO JEONG
Siliconarts
and
JINHONG PARK
LG Electronics
and
TACK-DON HAN
Yonsei University
and
DINESH MANOCHA
University of North Carolina at Chapel Hill
and
WOO-CHAN PARK
Sejong University

We present RayCore, a mobile ray-tracing hardware architecture. RayCore facilitates high-quality rendering effects, such as reflection, refraction, and shadows, on mobile devices by performing real-time Whitted ray tracing. RayCore consists of two major components: ray-tracing units (RTUs) based on a unified traversal and intersection pipeline and a tree-building unit (TBU) for dynamic scenes. The overall RayCore architecture offers considerable benefits in terms of die area, memory access, and power consumption. We have evaluated our architecture based on FPGA and ASIC evaluations and demonstrate its performance on different benchmarks. According to the results, our architecture demonstrates high performance per unit area and unit energy, making it highly suitable for use in mobile devices.

Categories and Subject Descriptors: Computing methodologies [**Computer graphics**]: Graphics systems and interfaces—*Graphics processors*; Computing methodologies [**Computer graphics**]: Rendering—*Ray tracing*

General Terms: ray tracing

Additional Key Words and Phrases: ray-tracing hardware, kd-tree, global illumination

ACM Reference Format:

Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. 2014. RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Trans. Graph.* 33, x, Article xxx (2014), 15 pages.
DOI:http://dx.doi.org/2601097.2601368

1. INTRODUCTION

Ray tracing [Whitted 1980] is a classic global illumination algorithm for photo-realistic rendering. Most applications that generate high-quality images using ray tracing perform off-line computations. However, with advances in semiconductor technology, recent research has focused on developing real-time ray tracing algorithms for CPUs [Wald et al. 2001; Reshetov et al. 2005; Wald et al. 2007; Djeu et al. 2011], GPUs [Aila and Laine 2009; Parker et al. 2010; Gribble and Naveros 2013], the Intel many integrated core (MIC) architecture [Benthin et al. 2012], and dedicated ray-tracing hardware [Schmittler et al. 2004; Woop et al. 2005; Nah et al. 2011; ImgTec 2013]. However, most of these techniques are designed for real-time rendering on desktop or laptop systems.

With the widespread use of mobile devices, including smartphones and tablets, there is considerable interest in generating photo-realistic images at low power cost. Moreover, it has been shown that ray tracing can be used to generate high-quality rendering while using less power than traditional multi-pass rasterization methods [Keller et al. 2013]. These recent studies on ray-tracing for mobile platforms can be classified into two types: the OpenGL ES-based software approach [Nah et al. 2010] and ray-tracing hardware architectures for mobile platforms [Kim et al. 2012; Spjut et al. 2012; Lee et al. 2012; Kim et al. 2013; Lee et al. 2013]. While these approaches are promising, most of them either do not provide sufficient performance for real-time ray tracing [Nah et al. 2010; Kim et al. 2012; Spjut et al. 2012; Kim et al. 2013] or are based

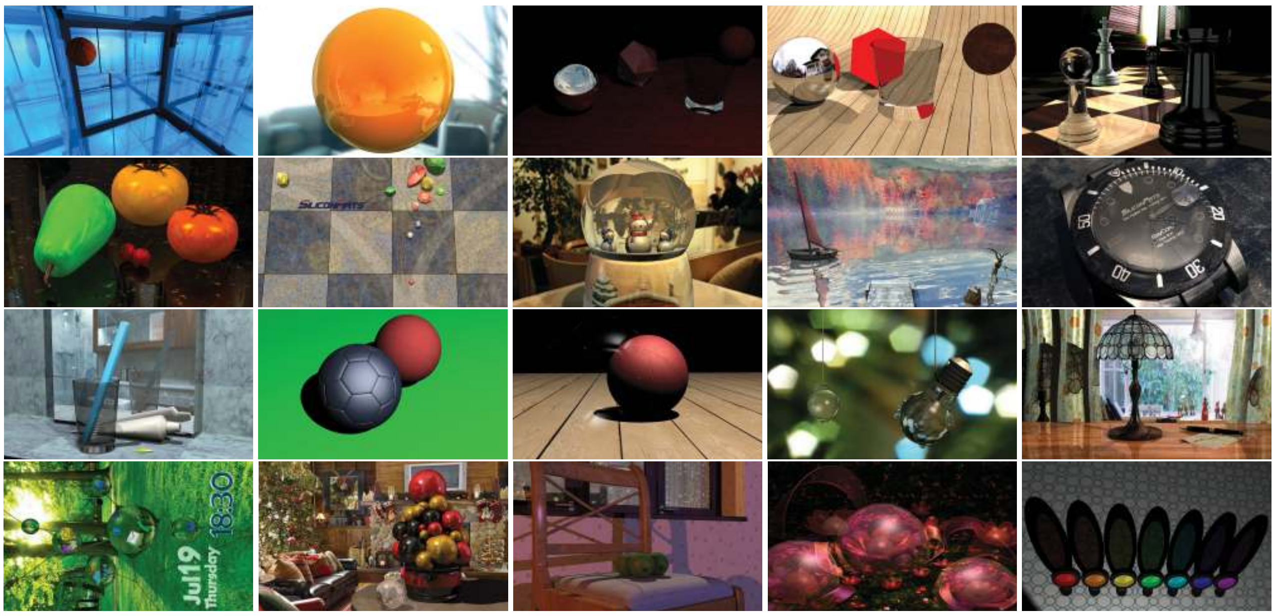

Fig. 1. Sample images rendered by RayCore. The scenes are composed of 0.6 K–64 K triangles, and also represent ray-tracing features supported by RayCore, such as specular reflection, refraction, and shadows. Our RayCore architecture also achieves real-time kd-tree construction for the scenes. Detailed information of these scenes is included in Table IV.

purely on software simulations [Nah et al. 2010; Spjut et al. 2012; Lee et al. 2012].

In this paper, we present RayCore, a dedicated hardware architecture for real-time mobile ray tracing. The RayCore architecture is based on the multiple instruction, multiple data (MIMD)-based ray-tracing architecture [Park et al. 2008], and has been developed as a ray-tracing hardware architecture that can be integrated into mobile application processors (APs). Mobile APs have many limitations as compared to desktop processors: smaller die areas, lower power resources, and lower memory bandwidth. For example, the Apple A7 processor has a die area of 102 mm^2, a maximum power consumption of less than 2-3 W, and 10.6 GB/s of the maximum memory bandwidth (dual-channel LPDDR3 at 1333 MHz) [NotebookCheck 2013], even though various components (CPUs, GPUs, DSPs, a memory controller, and so on) are integrated into a single chip. Given the constraints of mobile devices, our architecture is designed to achieve high power and area efficiency. Some of the main components of our hardware architecture include:

—**High ray-tracing performance on limited hardware resources:** According to NVIDIA's Tegra 4 Whitepaper [NVIDIA 2013], power efficiency (performance per watt) and area efficiency (performance per square millimeter) are very important design criteria for mobile processors. Our architecture includes a ray-tracing unit (RTU) that combines unified traversal and intersection (T&I) pipelines with a MIMD execution model to meet power and area efficiency goals. This hardware design simplifies the control logic and interfaces between each unit, and thereby enables high performance per unit area and unit energy; additionally, the RTU is designed to maintain ray-tracing performance regardless of ray coherence or scene characteristics. According to our ASIC evaluation, six RTUs achieve up to 239 Mrays/s using an area of 18 mm^2 and 1W power consumption using 28 nm process technology. This ray-tracing performance is comparable to other software-based ray tracing systems running on current GPUs [Aila and Laine 2009; Gribble and Naveros 2013] or the Intel MIC architecture [Benthin et al. 2012], but power consumption and the die area of RayCore are much lower than that of desktop platforms.

—**Interactive ray-tracing of dynamic scenes:** Unlike static scenes, which require only high ray-tracing performance, ray tracing dynamic scenes requires both high performance and fast acceleration-structure updates [Wald et al. 2009]. However, in resource-limited mobile hardware, the run-time cost of acceleration-structure updates can be high. To solve this problem, we present a hardware architecture that relies upon kd-tree construction. Our compact tree-building unit (TBU) with a die area of 1.6 mm^2 is power and area efficient; it can construct a surface-area heuristic (SAH) kd-tree for 64 K triangles within 20 ms, which is comparable to the performance of previous software-based CPU/GPU SAH kd-tree construction methods [Shevtsov et al. 2007; Hou et al. 2011].

—**Efficient latency hiding and low off-chip memory accesses:** Reducing off-chip memory accesses is very important for power efficiency and high performance on mobile devices [NVIDIA 2013]. To reduce performance degradation due to off-chip memory accesses, we use a novel latency-hiding technique called "looping for the next chance" on the T&I units. This technique is combined with other features of our architecture, including efficient memory systems of T&I units and the TBU and texture mip-mapping, to minimize off-chip memory accesses. As a result, real-time Whitted ray tracing with six RTUs and kd-tree construction with one TBU only require up to 1.1 GB/s of memory bandwidth in our benchmarks, respectively. This value is much less than the maximum bandwidth of mobile LPDDR3 memory (12.8 GB/s) [Wagner 2013].

2. RELATED WORK

2.1 Hardware-accelerated Ray Tracing

Ray-tracing hardware architectures can be separated into two types: they are either dedicated architectures, designed from the ground up for ray tracing, or redesigned versions of programmable multi-core architectures that have been optimized for ray tracing.

Many dedicated ray-tracing architectures have been proposed over the last decade. SaarCOR [Schmittler et al. 2004] is a ray-tracing hardware architecture with a ray generation and shading unit, a kd-tree traversal unit, and a ray-triangle intersection unit. This architecture has been extended to RPU [Woop et al. 2005] for programmable shading and D-RPU [Woop et al. 2006a] based on BKD-trees [Woop et al. 2006b]. The StreamRay architecture [Ramani et al. 2009] includes a filter engine for incoherent rays and a ray engine. The T&I Engine [Nah et al. 2011] introduces three concepts: an ordered depth-first layout, a three-phase intersection-test unit, and a ray accumulation buffer for latency hiding. Nah et al. [2013] have recently created a hybrid architecture which extends the T&I engine to dynamic scenes by using both CPUs and dedicated hardware. This hybrid architecture is based on asynchronous bounding volume hierarchy (BVH) construction [Wald et al. 2008]; a BVH refitting procedure is performed on a hardware unit while a BVH is rebuilt on a CPU. Recently, Doyle et al. [2013] describe a BVH construction hardware unit for ray tracing dynamic scenes. The Caustic Series2 [ImgTec 2013] is a commercial ray-tracing acceleration board for high-quality rendering on desktop PCs.

Other recent research focuses on redesigning programmable multi-core architectures. Copernicus [Govindaraju et al. 2008] is a tile-based parallel ray-tracing system running on 128 programmable cores. Mahesri et al. [2008] propose a many-core architecture for visual computing, including ray tracing. TRaX [Spjut et al. 2009] and MIMD threaded multiprocessors (TMs) [Kopta et al. 2010] are built on MIMD processor cores for incoherent ray tracing. Aila and Karras [2010] propose a new hardware architecture based on NVIDIA Fermi GPUs in order to reduce memory traffic via a treelet-based approach and a stack-top cache architecture. Kopta et al. [2013] improve the TRaX architecture's power efficiency by using a treelet-based approach and reconfigurable pipelines.

2.2 Mobile Ray Tracing

Mobile ray-tracing hardware and software architectures have received considerable attention. MobiRT [Nah et al. 2010] is a software ray tracer using OpenGL ES. MRTP [Kim et al. 2012] is a reconfigurable processor that supports both MIMD architectures and single instruction, multiple thread (SIMT) models. Kim et al. [2013] present a reconfigurable SIMT processor to improve the MRTP architecture. Spjut et al. [2012] extend the MIMD TMs to mobile environments and measure their performance with cycle-accurate simulation. SGRT [Lee et al. 2012; Lee et al. 2013] combines the T&I Engine [Nah et al. 2011] and Samsung reconfigurable processors (SRPs) for both high performance and flexibility.

2.3 Kd-tree Construction Algorithms

Ray tracing dynamic scenes is especially challenging, as dynamic scenes require fast tree construction for real-time rendering. The well-known SAH kd-tree construction has $O(n \log n)$ complexity [Wald and Havran 2006].

Some researchers have tried to improve the tree-construction time by designing approximations to the SAH tree-construction algorithm. Hunt et al. [2006] present a scanning (a.k.a. binning) approach for the SAH approximation. Shevtsov et al. [2007] present a parallel kd-tree construction algorithm on multi-core CPUs. This method combines object median for top-level nodes, binned SAH construction for mid-level nodes, and the exact SAH construction for bottom-level nodes. Zhou et al. [2008] describe the first GPU kd-tree construction method using the top-level spatial median and bottom-level SAH construction. Hou et al. [2011] improve Zhou et al.'s work by using a partial depth-first approach for large models. Karras [2012] improves the space-filling curve tree construction method proposed by Lauterbach et al. [2009] for better scalability on GPUs.

On the other hand, to maintain kd-tree quality, several researchers have tried to parallelize the exact SAH kd-tree construction. Choi et al. [2010] present two parallelization algorithms on multi-core CPUs (nested and in-place). Wu et al. [2011] describe a GPU-based algorithm for parallel SAH kd-tree construction with split clipping.

Finally, some researchers have investigated scene-graph hierarchies for tree construction. A gkDtree [Kang et al. 2013] is a scene-graph-based multi-level hierarchy for parallelization and partial updates of dynamic subtrees. Razor [Djeu et al. 2011] presents a parallel lazy-update method using scene-graph hierarchies.

3. RAYCORE HARDWARE ARCHITECTURE

In this section, we describe the overall architecture and some of the design criteria. Figure 2 illustrates various components of our RayCore architecture. A RayCore unit consists of a tree-building unit (TBU) and ray-tracing units (RTUs). The TBU performs SAH kd-tree construction for dynamic scenes, and the RTU performs ray tracing. We describe our design decisions in Section 3.1, the RTU architecture in Sections 3.2–3.7, the TBU architecture in Section 3.8, and the ray tracing API in Section 3.9.

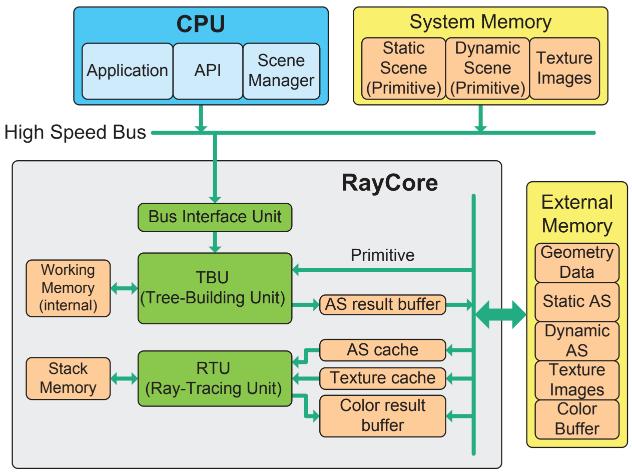

Fig. 2. Our ray-tracing architecture. In the current version, the type of the acceleration structure (AS) is a kd-tree.

3.1 Design Decisions

In this section, we describe some of the underlying goals that drove our design choices for the RayCore architecture, including high performance per unit area and unit energy, good support for dynamic scenes, and efficient memory systems.

Fixed-function units vs programmable units: Modern GPU architectures are based on standard APIs (OpenGL or DirectX) and consist of programmable shader units with several fixed-function special-purpose units (for rasterization, texture mapping, tessellation, etc.). In the case of mobile ray-tracing hardware architectures, however, no similar standards or APIs are known. Therefore, the previous architectures have been differently designed as fully programmable [Spjut et al. 2012; Kim et al. 2013] or partially programmable [Kim et al. 2012; Lee et al. 2013] architectures.

In contrast to the prior architectures, we chose a fully-hardwired approach for the overall RayCore architecture for the following reasons. First, a fully-hardwired approach is beneficial in terms of power and area efficiency. According to Borkar and Chien [2011], hardwired units can provide up to 50-500 times greater energy efficiency than general-purpose register organization in some cases. Additionally, instruction fetch-and-decode logic with instruction caches for programmability increases the required area. Thus, we feel that a fixed pipeline is much better suited for mobile devices with limited die areas and power resources. Second, modern mobile APs already include programmable GPUs. If the ultimate goal is to design an architecture that supports programmable shading and real-time ray tracing, we feel that they can both can be achieved by combining programmable GPUs in the same AP with our fixed-function pipelines.

MIMD vs SIMD: Single instruction, multiple data (SIMD) architectures have been widely used in modern CPUs and GPUs. For high SIMD efficiency, data-level parallelism is exploited. If multiple ray data is mapped into a SIMD unit (packet tracing [Wald et al. 2001]), the SIMD efficiency decreases when the ray coherence is low. Because high ray recursion depth or stochastic ray tracing decreases ray coherence, additional stream compaction or reordering methods [Ramani et al. 2009; Aila and Karras 2010; Nah et al. 2012] are needed to increase SIMD efficiency. In contrast, MIMD architectures exploit thread-level parallelism, so they can be more robust with respect to ray coherence [Mahesri et al. 2008; Spjut et al. 2009; Nah et al. 2011] than ray-packet-based SIMD architectures. Therefore, we use a MIMD-style architecture in RayCore.

A drawback of MIMD architectures is that they require more hardware resources than SIMD architectures. This is because all the control flow logic of the core should be replicated for each MIMD pipeline [Mahesri et al. 2008]. In other words, each MIMD pipeline should have its own independent front-end (instruction fetch-and-decode logic, an instruction cache, etc.). However, this drawback is not a serious issue in our architecture because our fixed-function hardware architecture does not perform instruction fetching and decoding.

Unified T&I units vs separate T&I units: Several ray-tracing architectures [Schmittler et al. 2004; Woop et al. 2006a; Nah et al. 2011; Lee et al. 2013] have separate hardware units for traversal and intersection tests. For example, SaarCOR [Schmittler et al. 2004] consists of a 4-way SIMD traversal unit for processing four rays and an intersection unit for processing a single ray. This 4:1 ratio originates from the expected workloads; there will be approximately four times as many traversal operations as intersection test operations. However, this assumption is not always true because the workload between the traversal and intersection tests can change according to the scene properties, the tree construction method, or the ray type (primary rays, secondary rays, shadow rays, etc.). Cases such as these can create a load imbalance problem, similar to that caused by separate shader architectures in traditional graphics hardware (vertex and pixel shaders) [Tamasi 2008]. In order to overcome this load imbalance problem, we propose a unified T&I pipeline (Section 3.4). Our architecture performs traversal and intersection operations in a single pipeline.

Multi-threading method: Modern GPU architectures support hardware multi-threading to achieve massive parallelism. Current GPU multi-threading systems use large register files for latency hiding; according to Kopta et al. [2010], the register area per streaming multiprocessor (SM) in NVIDIA GPUs is much larger than the compute area per SM. To minimize additional costs for hardware multi-threading, we present a novel multi-threading technique called "looping for the next chance" for our hardware architecture (Section 3.5). This technique prevents pipeline stalls by reusing existing input/output buffers and registers in a pipeline, so that it requires fewer memory resources than hardware multi-threading on GPUs.

Acceleration structure (AS): The use of acceleration structures enables fast ray tracing by reducing the number of ray-primitive intersection tests. Kd-trees and BVHs have been widely used as acceleration structures for ray tracing [Wald et al. 2009]. When SIMD instructions are not used, kd-trees are generally faster at single-ray traversal than BVHs due to early termination [Pharr and Humphreys 2010; Nah and Manocha 2014]. In addition, kd-trees exhibit better cache efficiency than BVHs due to the size of a kd-tree node (8 bytes). Four times as many kd-tree nodes as BVH nodes can be stored in a cache block because the size of a BVH node is typically 32 bytes. For these reasons, we use kd-trees in RayCore.

A drawback of kd-trees is a longer tree build time than BVHs. Because a kd-tree is a data structure based on spatial splits, a kd-tree has usually an order of magnitude more nodes than a BVH [Ize and Hansen 2011]. Thus, a SAH kd-tree has a proportionally longer build time than a SAH BVH on the same scene [Pharr and Humphreys 2010]. In order to address this issue, we present a dedicated kd-tree construction hardware architecture (Section 3.7).

Rendering effects: The main purpose of the proposed architecture is to produce real-time Whitted ray tracing [Whitted 1980] on mobile devices. Therefore, our architecture supports full Whitted effects, including specular reflection, refraction, and hard shadows. Our architecture can be also used to accelerate interactive distribution ray tracing [Cook et al. 1984]. To support this feature, the ray-generation unit in our architecture (Section 3.3) supports a Sudoku-based sampling technique [Boulos et al. 2006]. Additionally, the shading unit (Section 3.6) supports quadtree displacement mapping [Tevs et al. 2008] to provide detailed geometry to mobile 3D applications.

Primitive type: RayCore currently supports only triangles as geometric primitives. This strategy improves the performance and simplifies the system design, because it eliminates branching to support different primitive types [Wald et al. 2001]. Therefore, other types of primitives should be converted into triangles before rendering, as is done in rasterization-based GPUs. To reduce the required operations for a ray-triangle intersection test, we chose Wald's pre-computation-based intersection algorithm [Wald 2004].

3.2 Ray-Tracing Unit

Figure 3 depicts an overall block diagram of an RTU. The data path includes several units to handle setup processing, ray generation, traversal and intersection (T&I), hit point calculation, and shading. The architecture shown in this figure contains four T&I pipelines in each T&I unit. The memory system is composed of caches and buffers. The T&I caches are configured as two-level hierarchies;

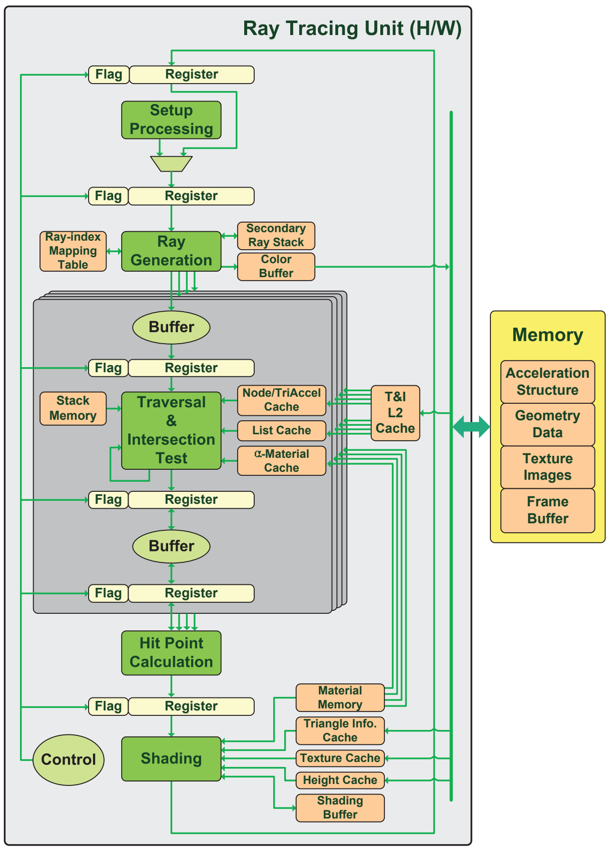

Fig. 3. The overall architecture of the ray-tracing unit.

each level-one (L1) cache (Node/TriAccel and list caches) shares a T&I level-two (L2) cache. The external memory is configured for AS data, geometry data (including triangle information), texture images, and the frame buffer.

Initially, the setup-processing unit (Section 3.3) passes ray information to the ray-generation unit (Section 3.3). The ray information either corresponds to a primary ray generated during the setup, or a secondary ray generated from the shading unit (Section 3.6). After a ray is generated, the T&I unit (Sections 3.4-3.5) performs node traversals and ray-triangle intersection tests to compute the hit point. After the hit point is generated, the position (x, y, and z) of the hit point is calculated in the hit-point calculation unit (Section 3.6). The shading unit then performs Phong illumination and texture mapping. When a ray generates additional secondary rays, the ray information is transferred to the setup-processing unit. When the transferred ray information indicates that there are no more propagated rays for a given pixel, the final color value of the pixel is stored in the color buffer.

3.3 Setup Processing and Ray Generation

The setup-processing unit initializes the information to generate primary rays. This information consists of the ray type (primary ray) and a ray index calculated by the screen coordinates. Because the number of in-flight rays in an RTU (227) is less than total pixel sizes, we use the reduced register bit width for the ray indices (8 bits) instead of the full bit width for the screen coordinates. The setup-processing unit also has a multiplexer for selecting either an initial primary ray or a secondary ray defined by the shading unit. If primary rays are continuously generated without regard to the status of the output buffer in the shading unit, deadlock can occur due to the circular wait between rays in the shading unit and rays in the ray generation unit. To prevent this, we prioritize secondary rays over primary rays.

The ray-generation unit generates a ray with the ray information from the setup-processing unit, stores a secondary ray in the dedicated stack, performs shadow ray culling, and stores the final color value in the color buffer. The generated rays in the ray-generation unit are supplied to the T&I pipelines.

A detailed procedure for generating each type of rays is described as follows. First, primary rays are generated by the Morton order (a.k.a. Z-curve) [Morton 1966] to improve cache efficiency, as shown by [Aila and Laine 2009]. To support this order, we use a very simple 6-bit counter and 4-bit position shifts for 64 pixels. If we define an original 6-bit block number as $i_5i_4i_3i_2i_1i_0$, the Morton-ordered coordinate value (x, y) is $(i_5i_3i_1, i_4i_2i_0)$.

Second, the number of secondary rays for a pixel varies according to the ray depth and the sample size. If a reflection ray and a refraction ray are generated concurrently, one is sent to the following pipeline stage for execution and the other is stored in a secondary ray stack. There are 16 entries in the stack in order to fully support both reflections and refractions. To prevent stack overflow, we restrict the maximum ray recursion depth to 15.

Third, a shadow ray's information for each light source is transferred into the ray-generation unit one by one from the shading unit. For effective shadow-ray tracing, a backface culling method [Suffern 2007], which rejects shadow rays generated on the backside of a triangle, is adopted. For this culling, we compare the hit triangle's surface normal with the position of the light source. Ambient occlusion (AO) rays are exceptional cases for the culling method because they are not affected by the position of light sources. The ray-generation unit also supports textured shadows.

Finally, sample rays are generated to support distribution ray tracing and super-sample anti-aliasing (SSAA). Our current implementation supports AO and diffuse inter-reflection among the various effects produced by distributing the rays. AO rays and diffuse inter-reflection rays are distributed according to the pre-defined Sudoku sequence [Boulos et al. 2006] on the hemisphere. The Sudoku sampling and tiling approach prevents temporal scintillation. For SSAA, RayCore supports regular grid sampling (2×2) and edge-based adaptive sampling.

3.4 Unified Traversal and Intersection (T&I) Pipeline

In this section, we describe our unified T&I pipeline. The goal of this architecture is to solve the load imbalance problem in previous separate T&I pipelines, as described in Section 3.1. In our architecture, a single pipeline performs nested loops for traversal and intersection operations to maintain hardware utilization. Moreover, the unified pipeline is suitable for supporting various acceleration structures. Our current implementation supports kd-trees, but BVHs and bounding interval hierarchies (a.k.a. BKD-trees or skd-trees) [Wächter and Keller 2006; Woop et al. 2006b; Havran et al. 2006] can easily be supported using the ray-box intersection test mode shown in Figure 5.

Although the unified-pipeline approach can increase the hardware area to support multiple modes, its greatly simplified control logic and interfaces between units more than compensate for the increased hardware area. Our MIMD architecture requires only one input and output buffer per T&I pipeline, which does not greatly

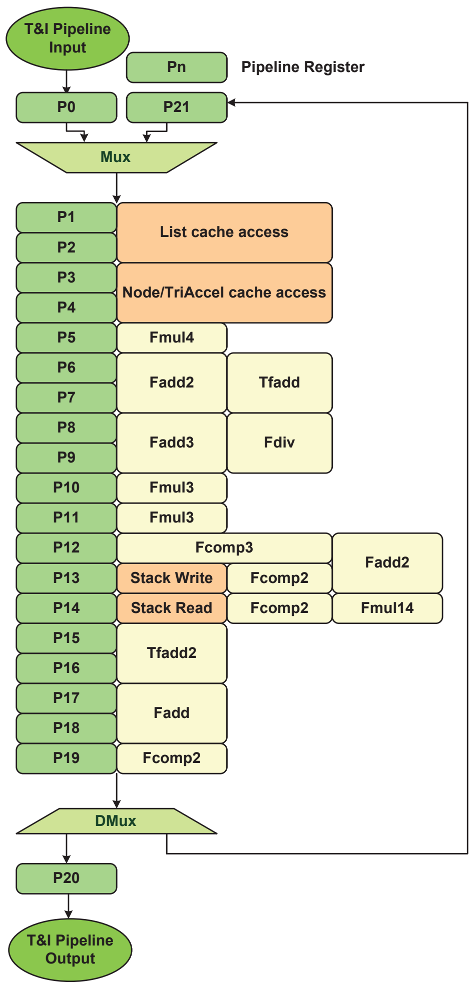

Fig. 4. A T&I pipeline block diagram. The numerical values attached at the end of each functional unit signify the number of implemented functional units.

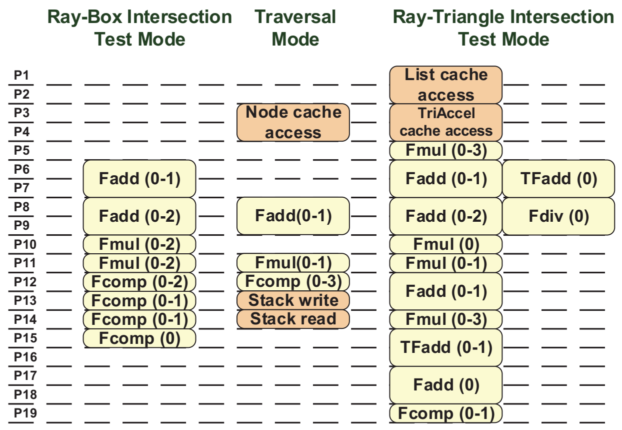

Fig. 5. The three operation modes in the T&I pipeline. The numbers in parentheses signify the number of used functional units shown in Figure 4.

increase the hardware area; this is because no communication between the T&I pipelines is necessary, given that each pipeline is dedicated to the traversal of a single unique ray.

Figure 4 shows a block diagram of a unified T&I pipeline. Two-cycle latency is needed for an L1 cache access, a floating-point addition (Fadd), a floating-point division (Fdiv), and a floating-point addition with three operands (Tfadd). One-cycle latency is required for floating-point multiplication (Fmul) and floating-point comparison (Fcomp). Each pipeline stage is connected with an internal data path, so the shape data (a kd-tree node or a triangle), ray data, and intermediate results can be transferred from a pipeline stage to another pipeline stage without delay. The width of the internal data path varies according to the particular stage of the pipeline. We unify a TriAccel cache for the TriAccel data structure in Wald's intersection test algorithm [Wald 2004] and a node cache to reduce the required SRAM size. Unlike the unified node/TriAccel cache, we use an independent list cache to exploit sequential access patterns.

Figure 5 illustrates three operation modes in the T&I pipeline: a ray-box intersection test, node traversal, and a ray-triangle intersection test. It also illustrates which hardware resources in the pipeline are utilized for each mode.

The first mode is used to calculate the initial *t* distance of the ray; it performs this step by testing the intersection between a ray and the bounding box surrounding the entire scene. The *t* distance is the distance from the ray's origin along the ray's normalized direction.

The second mode is activated in case of inner nodes to determine which of the two child nodes is first intersected by the ray. If an intersection is found, it can be returned as the near-side node. If both child nodes need to be traversed, the child node on the far side of the splitting plane is stored in the stack memory for a front-to-back traversal order. If neither of the child nodes is intersected by the ray, node data is read from the stack for recursive execution on that node.

To increase T&I pipeline utilization, two kd-tree traversal steps are concurrently processed in a pipeline; this is possible when the current and next nodes are located in the same cache block. Because we use the kd-tree layout in [Pharr and Humphreys 2010], the left child node is adjacent to its parent node, and they can be stored in a cache block. In this case, we traverse the nodes in parallel. If both the parent node and the child node are visited, we output the child's traversal result; if only the parent node is visited, we output only the parent's result.

The third mode is activated for leaf nodes and finds the triangle intersected by the ray. In this mode, the triangle list information is first read from the list cache to get the triangle's index. The pre-computed TriAccel data [Wald 2004] transferred from CPUs is retrieved from the node/TriAccel cache, and an intersection test is performed for a given ray to find the nearest hit point (for non-shadow rays) or any hit point (for shadow and AO rays). When the final intersected triangle is computed, the triangle's information is sent to the next stage; otherwise, the flow returns to the traversal stage and the traversal continues.

The shadow ray intersection tests continue until the ray strikes an opaque triangle. To support transparent shadows, we check whether all triangles hit by the shadow ray are transparent. In this case, we perform multiplication and accumulation operations to calculate the transparency rate. This accumulated alpha value is stored in the additional α-material cache (Figure 3) for simpler shading.

3.5 Memory System of the T&I Unit

Because the memory access in the T&I units accounts for the majority of memory access in overall ray tracing, the memory system of the T&I unit needs to be designed for efficiency. To reduce the number of external memory accesses, each T&I pipeline in the T&I unit is equipped with an L1 cache; all four pipelines share a common L2 cache. While the two-level hierarchical model increases the efficiency, the cache miss penalty is still quite high because a cache miss results in a pipeline stall. Thus, we include a latency-hiding technique to address this issue.

RayCore's memory system is designed around simple multi-threading, which offers both ease of hardware implementation and efficient hiding of memory latency. To achieve these goals, we devise an effective "looping for the next chance" scheme for each L1 and L2 cache (Figure 6). The principle of this scheme is simple; a cache miss triggers an idle setting in the ray thread that missed it for the remaining pipeline stages; that ray thread is set to active mode at the next loop to re-access the cache. In other words, a cache miss acts as prefetching data for the next loop. In case of the L1 cache, we bypass an L1 cache miss, and the ray thread is passed into its subsequent pipeline stage without stalling. In the next iteration, the cache request is generated again because the ray thread returns to the top of the pipeline (P21) in Figure 4. If a cache miss occurs again, then the above process is repeated. During the miss-handling process, the cache controller accesses either the L2 cache or the external memory to fetch the data.

The L2 cache operates as follows. If an L1 cache miss occurs, a request for an L2 cache access is outputted from the L1 Addr FIFO to the L2 cache. If the request results in an L2 cache hit, the address and the data for the request return back to the L1 Addr/Data FIFO. If the request results in a cache miss, the request from the L1 Addr FIFO goes to the L2 Addr FIFO and is discarded in the L1 Addr FIFO. By doing so, the next L2 cache access in the L1 Addr FIFO can be immediately performed.

The proposed scheme only uses existing internal memory (input/output buffers and pipeline registers) and, because of its additional memory space, can avoid a certain amount of buffering. Therefore, our technique is actually more area-efficient than the GPU-style multi-threading with large register files that are described in Section 3.1.

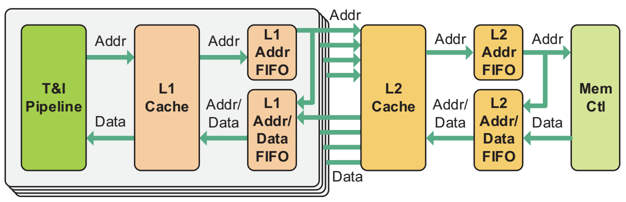

Fig. 6. A memory system for the looping for the next chance scheme.

3.6 Hit-Point Calculation and Shading

The hit-point calculation unit computes the intersection point between ray and triangle, expressed parametrically as $p(t) = o + t\vec{d}$, where o is the ray's origin and $\vec{d}$ is the ray's normalized direction.

The shading unit performs computations related to Phong illumination and texture mapping using the information stored in the material memory and the triangle-information cache. The calculated color is added to the previous color in the shading buffer; the final color is determined by accumulating color values from all rays generated from a given pixel. If anti-aliasing is enabled, the color values from multiple subpixels are averaged with a box filter. The final color value, along with other information, is eventually sent through the loop to the setup processing unit, where the color is written to the frame buffer. The shading buffer also stores ray information for secondary ray generation, and this ray information is transferred into the setup-processing unit.

Texture mapping in our system supports mip-mapping with a bilinear filtering scheme. It is similar to the previous rasterization method. It also includes a typical texture cache architecture [Hakura and Gupta 1997]. For bilinear filtering, four texel data are concurrently fetched and interpolated. For effective mip-mapping, we employ a mip-map level selection method using the ray length and the pre-calculated value [Park et al. 2011]. Compared to ray differential-based methods, this selection method reduces the computational resources needed to calculate a mipmap level. By adopting this method, we achieve both less texture aliasing and higher texture cache hit rates: up to 96%. We further increase texture-cache hit rates with a tile representation on texture data. Our texture-mapping unit also supports textured shadows.

In order to address the issue of using simple geometric primitives in mobile 3D graphics, the shading unit supports quadtree displacement mapping [Tevs et al. 2008]. We improve upon Tevs et al.'s method by using start-level decision, multi-level descending, and selective ascending algorithms. The displacement mapping hardware unit consists of a view calculation unit, a fully-pipelined quadtree traversal unit with a height cache, and an address calculation unit. The complete description of the proposed algorithm and architecture is beyond the scope of this paper, and is given in another full paper [Kwon et al. 2014].

3.7 Tree-Building Unit for Dynamic Scenes

In dynamic scenes, the acceleration structure should be updated for each frame [Wald et al. 2009]. Therefore, the performance of these acceleration structure updates is very important. In this section, we present a new tree-building hardware architecture to achieve the following goals: fast kd-tree construction without tree-quality degradation, minimized off-chip memory accesses, and exploitation of a burst memory access.

Figure 7 illustrates the proposed tree-building hardware architecture. Instead of exact SAH calculation for all nodes [Wald and Havran 2006], we divide the tree-construction procedure into two steps to achieve both fast construction and good tree quality: binning [Shevtsov et al. 2007; Djeu et al. 2011] and sorting. This

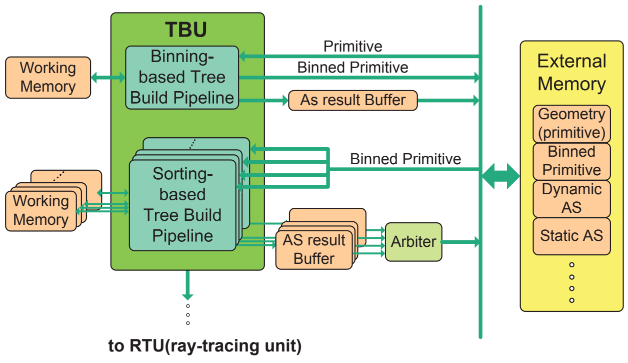

Fig. 7. The proposed tree-building architecture.

strategy is based on the experimental result in Razor [Djeu et al. 2011]; the quality degradation of binning is only a few percent, but binning achieves 3-7× faster tree construction speed as compared to exact SAH kd-tree construction. Our architecture uses binning for high-level tree construction and sorting for low-level tree construction. A tree-building unit (TBU) consists of a binning-based tree-building pipeline (TBP) for high-level nodes and multiple sorting-based TBPs for low-level nodes. For primitive sorting in the sorting-based TBPs, we used merge-sort hardware units [Siliconarts 2013].

To minimize off-chip memory access, the sorting-based TBP uses the internal SRAM for sorting, split plane selection, and geometry classification. This approach is better than using caches, because sorting is based on random access patterns. To maintain tree data in the working memory, the data is transferred from the binning-based TBP to sorting-based TBPs as soon as it is small enough (in other words, when the working set size is less than the size of working memory in a sorting-based TBP (51.3 KB).).

We reorder node data for an efficient burst transfer in DDR memory. Tree data consists of node data and geometry list data. The list data is stored as an array, so a burst-memory access of the list data is simple. In contrast, a burst-memory access has difficulty generating the node data because of its location. For the 8-byte compact kd-tree layout [Pharr and Humphreys 2010], the node data is stored in the depth-first order to remove the pointer to the left child node. This depth-first layout is naturally made in a single-threaded tree construction, but it is difficult in parallel kd-tree construction and the order cannot be predetermined. Therefore, if we write node data into the external memory without reordering, we cannot efficiently use a burst-memory access because the written node order in the internal memory is not the same as the depth-first order. To solve this problem, we reallocate a node construction sequence as the depth-first layout. With this reordering process, nodes are stored in the internal memory as the depth-first layout, as far as possible. After that, the generated node data can be transferred to the external memory via an efficient burst-memory access.

For more effective handling of dynamic scenes, we combine the two-tree approach [Bikker 2007] and the two-level approach [Wald et al. 2003]. We first classify the objects in the scene as static or dynamic. A static tree of all static objects is built only once on a CPU. For high tree quality, the static tree is constructed using the SAH with split clipping [Wald and Havran 2006]. In contrast, each dynamic object has its own tree, and these dynamic subtrees are rebuilt during each frame on the TBUs; a top-level dynamic tree is constructed using the dynamic subtrees. In the T&I unit, the top-level dynamic tree is traversed before the traversal of the static tree. This partial-update strategy is very useful to reduce tree-construction workloads if the greater part of the scene is static [Bikker 2007; Kang et al. 2013].

Currently, the TriAccel data [Wald 2004] is calculated on a CPU and is transferred to RTUs. According to Havel and Herout [2010], the precomptation throughput of Wald's TriAccel data is 31M triangles/s with a single thread on Intel Core 2 Duo E8200 (2.66 GHz). Therefore, this precomputation is not a bottleneck in our benchmarks.

3.8 Ray Tracing API

RayCore provides OpenGL ES 1.1-like API extensions to separate static and dynamic objects. Static objects are retained for subsequent frames and dynamic objects are transferred to the tree builder via vertex arrays to reconstruct dynamic subtrees during each frame. By using our extension functions for ray tracing, programmers can easily add ray-tracing effects to their OpenGL ES-based applications. If programmable shading is required, programmers will be able to use the existing GPU. In this case, an interface between T&I units and programmable shaders, which is similar to SGRT [Lee et al. 2013], can be applied to RayCore. With regard to S/W programming, the OpenGL ES API can be the medium that connects RayCore and the mobile GPU cores in the same AP.

4. HARDWARE IMPLEMENTATION

In this section, we describe a specific hardware implementation of our RayCore architecture. We first introduce our FPGA prototype. We then describe our ASIC evaluation and analyze the power consumption of the RayCore architecture.

4.1 FPGA Prototype

Figure 8 shows a Dynalith Systems iNEXT-V6 board, which contains two Xilinx Virtex-6 LX550 FPGA chips, 2 GB of DDR3 DRAM, and 8 MB of SRAM. A TFT LCD board with 800×480 screen resolution is attached to the iNEXT-V6 board. Its connection with any host computer is controlled via the PCI Express interface.

The implementation of the proposed ray-tracing architecture is built on four FPGA chips with two iNEXT-V6 boards. It operates at a speed of 84 MHz, and the 64-bit bus at the same frequency is used to access the external memory. The host computer (CPU) sends data to DRAM on each FPGA chip through the PCI express interface and through the bus functional model (BFM) integrated in the FPGA.

We first implemented four RTUs on the FPGA board to measure ray tracing performance. In this setting, each RTU is implemented on a unique FPGA chip, so there are four RTUs in our FPGA prototype. FPGA #0 acts as a master for load distribution, sending a block of pixel coordinates to any idle RTU on the other four units; we can thus dynamically allocate each block of 8×8 pixels into an RTU. The detailed task scheduling is as follows. After an RTU completes its execution on the allocated 8×8 pixels, another 8×8 pixel address is immediately requested to FPGA #0. If it is available within the current 16×16 pixels, it is returned from FPGA #0. Otherwise, FPGA #0 requests the next 16×16 pixel address into the pixel coordinate generator and an 8×8 pixel address within the next 16×16 pixels are returned. The final color values generated in each RTU are stored in the SRAM frame buffer on the FPGA #1 through the SRAM controller of FPGA #1.

We also implemented the TBU on the FPGA board to measure the kd-tree construction performance. The TBU consists of one binning-based TBP and four sorting-based TBPs. In this implementation, we used only a single FPGA chip (FPGA #0) for the TBU.

Table I shows the list of hardware resources for each unit. We use a 24-bit floating-point format (1 sign bit, 7 exponent bits, and 16 fraction bits) to reduce the register requirements. We use a table-based approach for the square root unit. To reduce the error caused

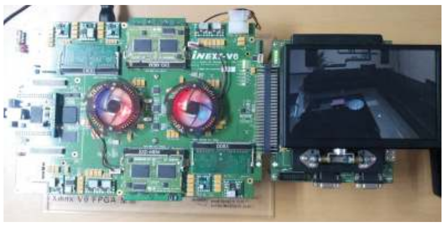

Fig. 8. An iNEXT-V6 board.

Table I. Hardware complexity for each unit in an FPGA chip.

	2-input adder	3-input adder	Comparator	Multiplier	Divider	Square root	Exp. unit
Ray-tracing unit (RTU)							
Ray generation	3	2		6	1	2	1
Traversal and Intersection	41	4	60	69	4		
Hit-point calculation and Shading	17	6	9	27 (18-bit) 8 (8-bit)	6	2	1
Total	61	12	71	110	11	4	2
Tree-building unit (TBU)							
1 binning TBP	14	1	92	12	1	0	0
4 sorting TBPs	132	4	0	84	0	0	0
Total	146	5	92	96	1	0	0

by accessing data in the square root table, the two adjacent values are linearly interpolated. The shading unit uses eight 8×8 multipliers for texture filtering.

Because the 24-bit precision has higher possibility of visual artifacts (e.g., holes) than the 32-bit precision, we applied the following solutions to our hardware architecture. First, when designing the arithmetic units for our hardware architecture, we reduce precision errors as far as possible. Second, we apply geometry transformations to geometry data to reduce the precision errors in ray-triangle intersection tests. Using this combination of approaches, our benchmarks do not show any significant visual artifacts in our prototype.

The total required SRAM size for an RTU is 507.1 KB. The sizes of an L1 Node/TriAccel cache, an L1 list cache, an L2 T&I Cache, a triangle-information cache, a texture cache, and a height cache for displacement mapping are 124.38 KB, 90.38 KB, 124.88 KB, 50.81 KB, 16.44 KB, and 4.23 KB, respectively. The set associativity of the L1 and L2 caches is two and eight, respectively. The total size of the internal buffers is 94.48 KB. The internal buffers consist of a ray-index mapping table, a secondary ray stack, a color buffer, traversal stacks, α-material memory, T&I input/output buffers, and a global scene information buffer for material and light data. The size of a traversal stack is 36 KB and this stack supports the maximum tree depth of 32. Because different rays exist in pipeline registers and input/output buffers, these rays can be concurrently processed. The maximal number of in-flight rays per RTU is 227.

The total required SRAM size for a TBU is 218.62 KB. A binning-based TBP has 13.42 KB of working memory, and a sorting-based TBP has 51.3 KB of working memory.

4.2 ASIC Evaluation

For the ASIC evaluation, we used TSMC's 28 nm high-performance, low-power (HPL) process [TSMC 2012] and Synopsys design compiler [Synopsys 2013]. A RayCore unit was synthesized up to 650 MHz with a voltage of 0.9 V; we set the target frequency to 500 MHz with some margin. The total area per RTU is 3 mm^2. Therefore, six RTUs can be allocated into 18 mm^2; this is similar to the area of current mobile GPUs (e.g., PowerVR SGX543 MP2) and other mobile ray-tracing hardware architectures [Kim et al. 2013; Spjut et al. 2012]. Also, a TBU occupies 1.6 mm^2.

The internal power consumption of a TBU and six RTUs, including controllers, L2 caches and the AXI bus interface, is approximately 1 W. This low power consumption is possible for the following reasons. First, we designed the RayCore architecture using fixed pipelines specifically for high performance and low power consumption. According to Hameed et al. [2010], the instruction fetch-and-decode logic occupies up to 45% of the total power consumption of a processor; RayCore needs neither instruction fetch-and-decode logic nor instruction caches. Second, the pipeline control logic is greatly simplified with the unified T&I architecture. Third, large register files and large caches, which consume a lot of power, are reduced in our architecture; the "looping for the next chance" technique minimizes required registers for multi-threading, and our efficient memory system achieves high cache hit rates with small L1/L2 caches. Fourth, the latest 28 nm HPL process [TSMC 2012] delivers 2× the gate density of the 40nm process, while reducing standby and operation power by more than 40%. To sum up, RayCore's low power consumption comes from both our careful design and our use of recent innovations in fabrication technology.

5. EXPERIMENT AND RESULTS

In this section, we first describe ray tracing performance and kd-tree construction performance of the FPGA prototypes, respectively. We then compare the RayCore ASIC version with other ray-tracing and kd-tree construction approaches.

5.1 Ray-Tracing Performance of the FPGA Prototype

This section includes the performance evaluation results for the ray-tracing unit (RTU). All numerical results in the evaluation were measured directly from the FPGA prototype. The kd-trees were constructed by the SAH [Wald and Havran 2006].

We use two different scene setups for our benchmark testing. First, to effectively describe full Whitted ray-tracing effects, we designed three new scenes, as shown in Figure 9: Kitchen (296 K triangles), Room with moving light (240 K triangles), and Living room (360 K triangles). The Kitchen and Living room scenes each have one moving camera and two static light sources. In contrast, the Room with moving light scene has one static camera, one static light source, and one dynamic light source. To measure ray coherence, we set the ray recursion depths to 0 and 10. In the latter case, coherence between the rays is quite low; rays of different types (primary, shadow, reflection, and refraction rays) and different depths are processed simultaneously in a T&I pipeline. Note that reflection or refraction rays were spawned only if the material on the hit point was reflective or refractive. The screen resolutions on this benchmark are 800×480 and 1600×960.

The second benchmark was structured in order to analyze the performance of the different ray types, similar to [Aila and Laine 2009]. In this benchmark, two scenes were selected (Figure 10): Conference (282 K triangles) and Sibenik (80 K triangles). In this benchmark, we set the ray types to a primary ray (the most coherent type), an AO ray, and a diffuse inter-reflection ray (the least coherent type). One light source used for all. A primary ray and a diffuse inter-reflection ray cast a shadow ray to the light source, but an AO ray does not cast any shadow rays because it is itself treated as a shadow ray. There were 32 samples per pixel for AO and diffuse inter-reflection rays. AO rays were terminated by the cut-off value of 5.0 for the maximum t distance.

Tables II and III show the performance results for the first and second benchmarks, respectively. These tables include the number of rays, cache hit rates, memory traffic, frames per second (FPS), and Mrays/s. We believe that 'Mrays/s' is the most important metric to estimate the overall performance of a ray-tracing accelerator because the number of rays in a scene varies according to the scene setup. The current RayCore FPGA implementation achieves 18–26

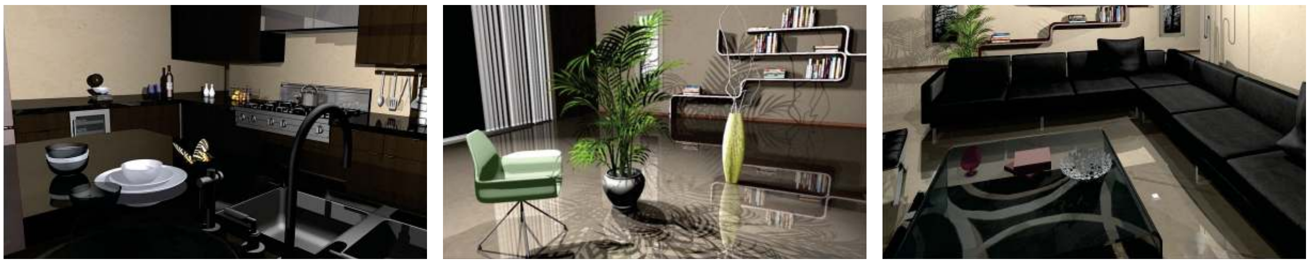
Fig. 9. Three static test scenes for Whitted ray tracing : Kitchen, Room with moving light, and Living room. These scenes can be rendered at interactive frame rates on our FPGA prototype.

Table II. FPGA performance results for Whitted ray tracing. We used four RTUs at 84 MHz for this experiment. Abbreviations: NT - node and TriAccel, TI - triangle information, and tex. - texture.

Scene (depth)	# of rays (M)	Cache hit rate (%) (L1 NT, L1 list, L2 T&I, TI, tex.)	Memory traffic (MB/s) (non-tex. / tex.)	FPS	Mrays/s
800×480 resolution					
Kitchen (0)	1.1	99 / 99 / 57 / 98 / 96	50 / 25	21.3	24.5
Kitchen (10)	1.8	99 / 99 / 54 / 96 / 95	88 / 31	14.5	26.2
Moving light (0)	1.1	99 / 99 / 62 / 96 / 96	85 / 26	20.8	23.7
Moving light (10)	1.8	99 / 99 / 61 / 96 / 95	89 / 33	13.6	25.5
Living room (0)	1.1	99 / 99 / 63 / 98 / 92	45 / 56	20.8	24.0
Living room (10)	1.6	99 / 99 / 62 / 98 / 93	47 / 48	15.4	25.2
1600×960 resolution					
Kitchen (0)	4.6	99 / 99 / 58 / 99 / 96	16 / 30	5.4	24.9
Kitchen (10)	7.3	99 / 99 / 56 / 98 / 95	34 / 36	3.7	26.8
Moving light (0)	4.5	99 / 99 / 64 / 98 / 96	25 / 33	5.3	23.9
Moving light (10)	7.4	99 / 99 / 63 / 98 / 94	26 / 43	3.4	25.9
Living room (0)	4.6	99 / 99 / 64 / 99 / 91	14 / 64	5.2	24.3
Living room (10)	6.6	99 / 99 / 62 / 99 / 92	15 / 58	3.8	25.5

Table III. FPGA performance results for distribution ray tracing. We used the same FPGA implementation as Table II. Abbreviations: pri - primary rays, AO - ambient occlusion, and dif - diffuse inter-reflection.

Scene (ray type)	# of rays (M)	Cache hit rate (%) (L1 NT, L1 list, L2 T&I, TI, tex.)	Memory traffic (MB/s) (non-tex. / tex.)	FPS	Mrays/s
800×480 resolution					
Conference (pri)	0.7	99 / 99 / 65 / 99 / -	62 / 0	27.5	21.1
Conference (AO)	13.0	98 / 99 / 78 / 99 / -	43 / 0	1.7	23.3
Conference (dif)	24.2	91 / 98 / 66 / 96 / -	468 / 0	0.7	18.9
Sibenik (pri)	0.7	99 / 99 / 73 / 97 / -	30 / 0	27.9	21.4
Sibenik (AO)	13.0	99 / 99 / 85 / 97 / -	26 / 0	1.8	23.6
Sibenik (dif)	25.2	89 / 99 / 75 / 87 / -	605 / 0	0.7	18.0
1600×960 resolution					
Conference (pri)	3.0	99 / 99 / 64 / 99 / -	20 / 0	7.0	21.7
Conference (AO)	52.2	99 / 99 / 82 / 99 / -	24 / 0	0.4	23.4
Conference (dif)	97.9	92 / 99 / 73 / 97 / -	420 / 0	0.2	20.4
Sibenik (pri)	3.0	99 / 99 / 73 / 99 / -	8 / 0	7.0	21.7
Sibenik (AO)	52.2	99 / 99 / 90 / 99 / -	8 / 0	0.4	23.6
Sibenik (dif)	100.9	90 / 99 / 78 / 90 / -	527 / 0	0.2	20.4

Mrays/s. We also measure the memory traffic bandwidth and observe 8–605 MB/s. In the scenes including textures, memory traffic increases due to texture fetching. In the case of diffuse inter-reflection rays, memory traffic increases due to lower ray coherence than other ray types. However, the performance degradation caused by lower ray coherence is not large, so we believe that our MIMD-style unified T&I architecture has robust performance regardless of ray types.

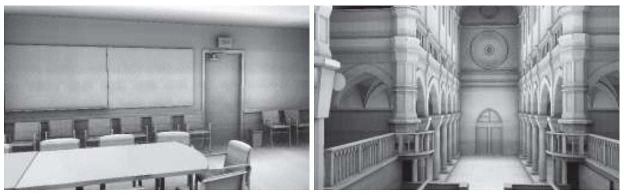
Fig. 10. Sample images from two static test scenes: Conference (courtesy of Anat Grynberg and Greg Ward) and Sibenik (courtesy of Marko Dabrovic). These images were rendered with ambient occlusion.

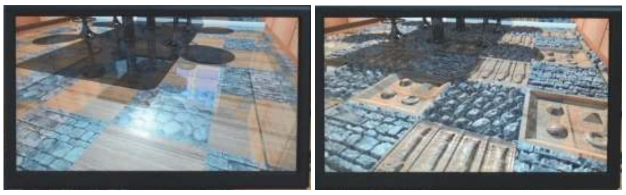
Fig. 11. Two captured images rendered without displacement mapping (left) and with displacement mapping (right). Images from [Kwon et al. 2014].

The frame rate of the high resolution (1600×960) is approximately 4× lower than that of the low resolution (800×480) because of the 4× increased number of rays. However, the ray-tracing performance (Mrays/s) is slightly higher in the higher screen resolution due to increased cache hit rates. The high resolution decreases the memory traffic from non-texture data due to increased ray coherence [Wald et al. 2001], but it slightly increases the memory traffic from texture data because higher resolutions require finer texture levels [Park et al. 2011]. Taken together, though, high resolutions decrease the memory traffic, since the lower memory traffic from non-texture data is more than the increase in memory traffic from texture data at the high resolution (Table II). Due to these reasons, the higher screen resolutions slightly increase ray-tracing performance, while considerably decreasing the overall memory traffic. This result also suggests that additional power consumption for off-chip memory accesses can also decrease when the screen resolution increases.

We also measured the result of quadtree displacement mapping in the BART Kitchen scene [Lext et al. 2001] (Figure 11). We modified the floor texture map to a 512×512 height map. When the displacement mapping mode is enabled, the RayCore FPGA version achieves a performance of 18–20 Mrays/s, with approximately 30% increase of memory traffic to access the height map. These results indicate that our architecture can render quadtree displacement mapping at a real-time rate with a small overhead.

Finally, we compare the performance of our FPGA prototype to a state-of-the-art mobile ray-tracing hardware architecture [Lee et al. 2013]. For Whitted ray tracing, SGRT [Lee et al. 2013] demonstrated 1.3-2.1 FPS at 45 MHz with two Virtex-6 LX760 chips, and RayCore can achieve 13-15 FPS at 84 MHz with four Virtex-6 LX550 chips. If we assume the same clock frequency and number of FPGA chips, SGRT will achieve 4-7 FPS. Therefore, we believe that our FPGA prototype is more efficient than the current SGRT prototype.

5.2 Kd-tree Construction Performance of the FPGA Prototype

In this section, we describe the performance of our kd-tree building unit in various scenes shown in Figure 1. The scenes were designed for mobile 3D graphics, and the triangle counts of the scenes are 0.6 K–64 K. All kd-trees were constructed from scratch. A leaf node was made if either the best split of the node was more costly than no split [Wald and Havran 2006] or the number of triangles in the leaf node was four and less.

According to the results in Table IV, the kd-tree build time on the FPGA prototype grow approximately linearly with the primitive count. The longest time is 117.9 ms for the Transparent Shadow scene with 64 K triangles (the last image in Figure 1). Furthermore, the memory traffic for these scenes is rather low: up to 36 MB/frame (the Transparent Shadow scene). This result means that the ASIC version of a TBU at 500 MHz can update all the benchmarks shown in Figure 1 at a frame rate of 50 FPS or more, without high memory bandwidth requirements. Therefore, real-time ray tracing dynamic scenes on mobile devices can be facilitated with our tree building hardware.

Table IV. Kd-tree construction performance of the TBU for the scenes in Figure 1. We used a single TBU at 84 MHz for this experiment.

Scene	Number of triangles	Kd-tree build time (ms)	Memory traffic (MB/frame)
Glass Room	608	1.6	0.1
Gloss	3552	3.6	0.9
Light Attenuation	6128	10.1	2.8
Self Illumination	6140	9.0	2.3
Chess	6604	9.2	2.9
Vegetables	6912	11.2	2.8
Jewel	8770	11.7	2.9
Orgel	9103	20.3	5.8
Landscape	12023	17.7	4.7
Watch	12552	19.5	5.4
Cup	12916	25.6	7.7
Shading Normals	15142	24.2	6.6
Opacity Material	15142	26.5	6.9
Bulb	16056	27.5	8.1
Desk Lamp	17154	27.5	8.6
Water UI	17833	28.6	10.0
Christmas Tree	19174	37.1	10.7
Chair	24014	40.9	13.0
Fractal Flowers	42651	77.6	23.6
Transparent Shadows	63842	117.9	36.1

5.3 Performance Evaluation of the RayCore ASIC Version and Comparison with Other Approaches

We use the same metric as D-RPU [Woop et al. 2006a] to evaluate the performance of our architecture. The expected performance of six RTUs at 500 MHz is nine times faster than the FPGA implementation due to the difference in the clock frequency (84 MHz $\rightarrow$ 500 MHz) and the number of RTUs (4 $\rightarrow$ 6). Additionally, the expected performance of the TBU at 500 MHz is six times faster than the FPGA implementation due to the clock frequency. Because ray tracing is "embarrassingly parallel," it is scalable with sufficient memory bandwidth. Six RTUs and one TBU (with 30 FPS) require only up to 1.1 GB/s of memory bandwidth for Whitted ray tracing for the static scenes shown in Figure 9 and kd-tree construction for the benchmarks shown in Figure 1, respectively. Thus, we anticipate scalable performance with six RTUs and a TBU.

Ray-tracing performance on RTUs: With the above metric, we expect that the RayCore ASIC version will achieve up to 239 Mrays/s (the Kitchen scene) with six RTUs. This performance means that the scenes in Figure 9 can be rendered at 56 FPS at HD 720p resolution.

As aforementioned, we predict that memory traffic will not degrade the overall performance in our benchmark scenes with Whitted ray tracing and AO. In these cases, we expect that the RayCore ASIC version will require up to 1.1 GB/s for ray tracing; this value is obtained by multiplying the highest value in Section 5.1 by nine. Dual LPDDR3-1333 for modern APs provides a memory bandwidth of 12.8 GB/s, which is sufficient for six RTUs. In the case of diffuse inter-reflection, we forecast that faster memory systems are required to prevent memory bottlenecks.

Table V compares our hardware implementation with other approaches in terms of ray-tracing performance. Our architecture achieves far better performance than state-of-the-art mobile ray-tracing hardware architectures; in fact, RayCore's performance is on par with state-of-the-art ray tracers on desktop platforms, despite requiring a much smaller die area and lower power consumption. Note that the documented power consumption of desktop platforms is thermal design power (TDP), and the TDP is similar to actual maximum power consumption of the graphics card [Hagedoorn 2012]. However, RayCore's internal power consumption listed in Table V does not include the power consumed by off-chip memory.

To estimate RayCore's power consumption for off-chip memory accesses, we begin with the maximum power consumption of dual-channel LPDDR3, which is 320 mW [Wagner 2013]. As previously mentioned, six RTUs only require up to 8% of the maximum bandwidth of dual LPDDR3 (12.8 GB/s) for Whitted ray tracing. This off-chip memory bandwidth requirement is much lower than that of previous desktop-based hardware architectures; off-chip memory bandwidth requirements per ray of RayCore, D-RPU [Woop et al. 2006a], and RPU [Woop et al. 2005] in the Conference scene for node/triangle data fetching are 3 bytes, 64 bytes, and 92 bytes, respectively. Therefore, we estimate that the off-chip memory accesses of RayCore will not significantly increase the total power consumption.

If RayCore is implemented on desktop platforms, we can use the larger area, higher clock frequency, higher memory bandwidth, and additional power resources that desktops offer to achieve even higher performance. NVIDIA's state-of-the-art GTX 680 GPU (a single chip version of GTX 690) has an area of 294 mm^2, a clock frequency of 1GHz, a memory bandwidth of 192 GB/s, and a TDP

Table V. Comparison with other ray-tracing platforms in the Conference scene with primary rays.

	Desktop platforms		Desktop ray-tracing H/W		Mobile ray-tracing H/W	
	NVIDIA GTX690	Intel MIC	D-RPU ASIC	Caustic R2100	Reconf. SIMT	RayCore ASIC
	[Gribble and Naveros 2013]	[Benthin et al. 2012]	[Woop et al. 2006a]	[ImgTec 2013]	[Kim et al. 2013]	(ours)
Performance (Mrays/s)	500	210	128	50*	4**	193
Process (nm)	28	45	90	90	90	28
Number of cores	1536 × 2	32	8	-	4	6
Clock (MHz)	915	1200	400	-	50-400	500
Area (mm^2)	294 × 2	-	186	-	16	18 (RTU) + 1.6 (TBU)
Power consumption (W)	300 (TDP)	-	-	30 (max)	0.2@100MHz	1

* Incoherent-ray performance in common scenes ** Bunny model (69 K triangles)

Table VI. Comparison with other SAH-based kd-tree construction approaches.

	CPU approaches		GPU approaches		Dedicated H/W
	[Shevtsov et al. 2007]	[Choi et al. 2010]	[Hou et al. 2011]	[Wu et al. 2011]	RayCore ASIC (ours)
Scene (triangle count)	Bunny (69K)	Bunny (69K)	Robots (71K)	Bunny (69K)	Transparent Shadows (64K)
Time to construct a kd-tree (ms)	27	50	38	59	20
Kd-tree build method	object median + binned/exact SAH	exact SAH	spatial median + exact SAH	exact SAH	binned/exact SAH
Platform	Intel Core2 Duo ×2	Intel Xeon X7550 ×4	NVIDIA Geforce GTX280		Tree-building unit (TBU)
Number of cores	4	32 (8 × 4)	240		1 binning + 4 sorting
Clock (MHz)	3000	2000	648 (core) / 1476 (shader)		500
Process (nm)	65	45	55		28
Area (mm^2)	143 ×2	684 ×4	576		1.6 (TBU only)
Power consumption (W)	65 (TDP) ×2	130 (TDP) ×4	236 (TDP)		1 (total)

of 195W, with 28 nm process technology. In the same environment, the RayCore desktop version can use a 17× larger area, a 2× faster clock frequency, and a 22× higher memory bandwidth. Thus, we predict that the RayCore desktop version would exhibit much higher performance than the current mobile version.

Kd-tree construction performance on a TBU: Table VI compares our tree-building hardware and other kd-tree construction approaches. We selected benchmark scenes with similar triangle counts (64 K–71 K). The tree construction times of CPU/GPU approaches are 27–59 ms with high computational power (e.g., 4–32 core CPUs or 240-core GPUs); our approach achieves 20 ms with fewer computational resources (one binning unit and four sorting units with 1.6mm^2 die size). Because binned SAH kd-tree construction generally provides higher tree quality than object/spatial median kd-tree construction, our hybrid approach (binned SAH + exact SAH) can provide higher tree quality than other hybrid approaches that use object/spatial median [Shevtsov et al. 2007; Zhou et al. 2008; Hou et al. 2011]. Additionally, our architecture is particularly advantageous in terms of memory traffic (up to 36 MB/frame). This result indicates that the TBU can be very useful for ray tracing dynamic scenes on resource- and energy-limited mobile hardware.

Next, we compare our kd-tree-based TBU to other BVH-based approaches. Recently, various BVH construction algorithms have been introduced for use on CPUs [Gu et al. 2013], GPUs [Lauterbach et al. 2009; Karras 2012; Karras and Aila 2013], and dedicated hardware [Doyle et al. 2013] and these algorithms achieve very fast BVH build time (a few milliseconds). However, it is difficult to directly compare kd-tree build methods with BVH build methods because a kd-tree has more nodes than a BVH, as described in Section 3.1. In preliminary comparisons of our hardware architecture and BVH-based methods, our approach has advantages over CPU/GPU-based methods in performance per unit area and unit energy; those CPU/GPU methods use up to 32 CPU cores [Gu et al. 2013] or a many-core GPU with 2688 cores [Karras and Aila 2013] to achieve their high tree-building performance. Additionally, our TBU is comparable to the BVH build hardware [Doyle et al. 2013]; four BVH build units in [Doyle et al. 2013] require 1 ms to build a BVH for the Toaster scene (11 K triangles), while our ASIC vesion of a TBU requires 3 ms to build a kd-tree for the Landscape scene (12 K triangles). The estimated area of four BVH build units in [Doyle et al. 2013] is 31.88 mm^2 with 65 nm process technology; the area of our TBU is 1.6 mm^2 with the 28 nm process technology. Even considering that the process technologies are different (65 nm vs 28 nm), our kd-tree build architecture is at least competitive with the BVH build architecture [Doyle et al. 2013] in performance per unit area.

6. CONCLUSIONS, LIMITATIONS, AND FUTURE WORK

Limitations and Future Work: The current version of RayCore has several limitations that we would like to resolve in the future. First, we have mainly focused on mobile ray tracing in this paper. However, we think that RayCore can also be used for high-quality off-line rendering. In future studies, we would like to investigate possible ways to accelerate other rendering techniques, such as bidirectional path tracing [Veach and Guibas 1994] and micropolygon ray tracing [Djeu et al. 2011].

Second, the current version of RayCore consists of fixed pipelines. To support various shading effects, such as motion blur, defocus blur, and procedural texturing, we would like to directly combine RayCore with the commodity programmable shaders on a GPU.

Third, even though RayCore includes an efficient memory system, incoherent ray tracing requires high memory traffic. To reduce the required off-chip memory bandwidth, we may consider additional ray-sorting units using other approaches [Moon et al. 2010; Nah et al. 2012] to increase ray coherence.

Fourth, our current architecture can only use kd-trees for its acceleration structure. Because kd-trees and BVHs use almost the same tree traversal and SAH tree construction procedures, we would like to extend our RTU and TBU architectures to also support BVHs in the future. Additionally, we are interested in accelerating shadow ray traversal using some specific traversal orders [Ize and Hansen 2011; Nah and Manocha 2014] instead of the current front-to-back order in our RTU architecture.

Fifth, the precomputation of the TriAccel structure could become a bottleneck in large dynamic scenes. Thus, in the next hardware implementation of RayCore, we would like to consider adding another TriAccel calculation unit to the TBU to improve its handling of large dynamic scenes.

Finally, we used reduced 24-bit precision and geometric transformations to reduce the precision errors. We would like to further analyze its impact on the accuracy and results.

Conclusions: We have proposed a new hardware ray tracer for mobile devices. To achieve high performance per unit area and unit energy, we implemented various novel techniques, including the unified T&I pipeline and a kd-tree building hardware architecture. We verified our architecture using both FPGA and ASIC evaluations. Our results show that RayCore achieved a performance improvement of up to two orders of magnitude over previous mobile ray-tracing hardware [Kim et al. 2013]. We are integrating the RayCore unit into a mobile AP, which will be released in the near future.

RayCore can be used for various mobile applications that demand high-quality 3D graphics. Generating complex lighting effects using current rasterization-based methods requires a great deal of programmer effort; ray tracing naturally supports these effects [Shirley et al. 2008], and makes programming for high-quality images far simpler than current methods. Therefore, ray tracing makes the programming for high-quality images simpler than rasterization methods. RayCore provides both sufficient performance for real-time ray tracing and an OpenGL ES-like API, so RayCore can benefit many types of applications. In the future, we would like to explore applications of our RayCore architecture to mobile applications, such as games, user interfaces, and augmented reality.

Acknowledgments

This research was supported by Siliconarts and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2012R1A1A2004624). Dinesh Manocha was supported by ARO Contract W911NF-10-1-0506, and NSF awards 0917040 and 1320644. We would like to thank the reviewers for their helpful comments.

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of the Conference on High-Performance Graphics 2010*.

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 145–149.

Carsten Benthin, Ingo Wald, Sven Woop, Manfred Ernst, and William R. Mark. 2012. Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics* 18, 9 (2012), 1438–1448.

Jacco Bikker. 2007. Real-time Ray Tracing through the Eyes of a Game Developer. In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing 2007*. 1–10.

Shekhar Borkar and Andrew A Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67–77.

Solomon Boulos, David Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Ingo Wald, and Peter Shirley. 2006. *Interactive Distribution Ray Tracing*. Technical Report No UUSCI-2006-022. SCI Institute, University of Utah.

Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. 2010. Parallel SAH k-D Tree Construction. In *Proceedings of the Conference on High Performance Graphics 2010*. 77–86.

Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. ACM, 137–145.

Peter Djeu, Warren A. Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. 2011. Razor: An architecture for dynamic multiresolution ray tracing. *ACM Transactions on Graphics* 30, 5, Article 115 (2011), 115:1–115:26 pages.

Michael J. Doyle, Colin Fowler, and Michael Manzke. 2013. A hardware unit for fast SAH-optimised BVH construction. *ACM Transactions on Graphics (SIGGRAPH '13)* 32, 4, Article 139 (July 2013), 10 pages.

Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 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.

Christiaan Gribble and Alexis Naveros. 2013. GPU ray tracing with rayforce. In *ACM SIGGRAPH 2013 Posters*. 98:1–98:1.

Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference*. 81–88.

Hilbert Hagedoorn. 2012. *Geforce GTX 680 review*. Technical Report. The guru of 3D. http://www.guru3d.com/articles_pages/geforce_gtx_680_review.

Ziyad S. Hakura and Anoop Gupta. 1997. The design and analysis of a cache architecture for texture mapping. *SIGARCH Computer Architecture News (Proceedings of ISCA '97)* 25, 2 (1997), 108–120.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Computer Architecture News (Proceedings of ISCA 2010)* 38, 3 (June 2010), 37–47.

Jiri Havel and Adam Herout. 2010. Yet Faster Ray-Triangle Intersection (Using SSE4). *IEEE Transactions on Visualization and Computer Graphics* 16, 3 (2010), 434–438.

Vlastimil Havran, Robert Herzog, and Hans.-Peter. Seidel. 2006. On the Fast Construction of Spatial Hierarchies for Ray Tracing. In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing 2006*. 71 –80.

Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. 2011. Memory-Scalable GPU Spatial Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics* 17, 3 (2011), 466–474.

Warren Hunt, William R Mark, and Gordon Stoll. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing 2006*. 81–88.

ImgTec. 2013. *Imagination Technologies ships Caustic Series2 R2500 and R2100 ray tracing acceleration boards*. Technical Report. http://www.imgtec.com/news/release/index.asp?NewsID=722.

Thiago Ize and Charles D. Hansen. 2011. RTSAH Traversal Order for Occlusion Rays. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2011)* 30, 2 (2011), 297–305.

Yoon-Sig Kang, Jae-Ho Nah, Woo-Chan Park, and Sung-Bong Yang. 2013. gkDtree: A group-based parallel update kd-tree for interactive ray tracing. *Journal of Systems Architecture* 59, 3 (2013), 166–175.

Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*. 33–37.

Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*. 89–99.

Alexander Keller, Tero Karras, Ingo Wald, Timo Aila, Samuli Laine, Jacco Bikker, Christiaan Gribble, Won-Jong Lee, and James McCombe. 2013. Ray tracing is the future and ever will be.... In *ACM SIGGRAPH 2013 Courses (SIGGRAPH '13)*. Article 9, 7 pages.

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

Hong-Yoon Kim, Young-Jun Kim, Jiehwan Oh, and Lee-Sup Kim. 2013. A Reconfigurable SIMT Processor for Mobile Ray Tracing With Contention Reduction in Shared Memory. *IEEE Transactions on Circuits and Systems I: Regular Papers* 60, 4 (2013), 938–950.

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2013. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13)*. 121–128.

Daniel Kopta, Joseph Spjut, Erik Brunvand, and Al Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *Proceedings of IEEE International Conference on Computer Design 2010*.

Hyuck-Joo Kwon, Jae-Ho Nah, Dinesh Manocha, and Woo-Chan Park. 2014. Effective traversal algorithms and hardware architecture for pyramidal inverse displacement mapping. *Computers & Graphics* 38 (2014), 140–149.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

Won-Jong Lee, Shi-Hwa Lee, Jae-Ho Nah, Jin-Woo Kim, Youngsam Shin, Jaedon Lee, and Seok-Yoon Jung. 2012. SGRT: a scalable mobile GPU architecture based on ray tracing. In *ACM SIGGRAPH 2012 Talks (SIGGRAPH '12)*. Article 2, 1 pages.

Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seok-Yoon Jung, Shi-Hwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: A Mobile GPU Architecture for Real-Time Ray Tracing. In *Proceedings of the 5th High-Performance Graphics Conference*. 109–119.

Jonas Lext, Ulf Assarsson, and Tomas Möller. 2001. BART : A Benchmark for Animated Ray Tracing. *IEEE Computer Graphics and Applications* 21, 2 (2001), 22–31.

Aqeel Mahesri, Daniel Johnson, Neal Crago, and Sanjay J. Patel. 2008. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. 164–175.

Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. 2010. Cache-oblivious ray reordering. *ACM Transactions Graphics* 29, 3 (2010), 28:1–28:10.

G. M Morton. 1966. *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing,*. International Business Machines Company.

Jae-Ho Nah, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2012. Efficient ray sorting for the tracing of incoherent rays. *IEICE Electronics Express* 9, 9 (2012), 849–854.

Jae-Ho Nah, Yoon-Sig Kang, Kwang-Jo Lee, Shin-Jun Lee, Tack-Don Han, and Sung-Bong Yang. 2010. MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices. In *ACM SIGGRAPH ASIA 2010 Sketches*. Article 50, 50:1–50:2 pages.

Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, Dinesh Manocha, and Tack-Don Han. 2013. HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics* (2013). conditionally accepted with major revision.

Jae-Ho Nah and Dinesh Manocha. 2014. SATO: Surface-Area Traversal Order for Shadow Ray Tracing. *Computer Graphics Forum* (2014). preprint.

Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2011. T&I engine: traversal and intersection engine for hardware accelerated ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2011)* 30, 6, Article 160 (2011), 160:1–160:10 pages.

NotebookCheck. 2013. *Apple A7 Smartphone SOC*. Technical Report. NotebookCheck. http://www.notebookcheck.net/Apple-A7-Smartphone-SoC.103280.0.html.

NVIDIA. 2013. *Whitepaper: NVIDIA Tegra 4 Family GPU Architecture*. Technical Report.

Woo-Chan Park, Dong-Seok Kim, Jeong-Soo Park, Sang-Duk Kim, Hong-Sik Kim, and Tack-Don Han. 2011. The design of a texture mapping unit with effective MIP-map level selection for real-time ray tracing. *IEICE Electronics Express* 8, 13 (2011), 1064–1070.

Woo-Chan Park, Jae-Ho Nah, Jeong-Soo Park, Kyung-Ho Lee, Dong-Seok Kim, Sang-Duk Kim, Jin-Hong Park, Cheong-Ghil Kim, Yoon-Sig Kang, Sung-Bong Yang, and Tack-Don Han. 2008. An FPGA implementation of whitted-style ray tracing accelerator. In *IEEE Symposium on Interactive Ray Tracing, 2008*. 187–187.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)* 29, 4 (2010), 1–13.

Matt Pharr and Greg Humphreys. 2010. *Physically Based Rendering* (second ed.). Morgan Kaufmann.

Karthik Ramani, Christiaan P. Gribble, and Al Davis. 2009. StreamRay: a stream filtering architecture for coherent ray tracing. In *ASPLOS '09: Proceeding of the Architectural support for programming languages and operating systems*. ACM, 325–336.

Alexander Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24, 3 (2005), 1176–1185.

Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. 2004. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 95–106.

Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2007)* 26, 3 (2007), 395–404.

Peter Shirley, Kelvin Sung, Erik Brunvand, Alan Davis, Steven Parker, and Solomon Boulos. 2008. Fast ray tracing and the potential effects on graphics and gaming courses. *Computers & Graphics* 32, 2 (2008), 260–267.

Siliconarts. 2013. *RaySort*. Technical Report. http://www.siliconarts.co.kr.

Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: a multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1802–1815.

Joseph Spjut, Daniel Kopta, Erik Brunvand, and Al Davis. 2012. A Mobile Accelerator Architecture for Ray Tracing. In *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*.

Kevin Suffern. 2007. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd.

Synopsys. 2013. *Power Optimization in Design Compiler*. Technical Report. Synopsys.

Tony Tamasi. 2008. Evolution of Computer Graphics. In *NVISION 08*.

Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. 2008. Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering. In *I3D '07: Proceedings of the 2008 symposium on Interactive 3D graphics and games*. ACM, 183–190.

TSMC. 2012. *28nm Technology*. Technical Report. http://www.tsmc.com/english/dedicatedFoundry/technology/28nm.htm.

Eric Veach and Leonidas Guibas. 1994. Bidirectional estimators for light transport. In *Proceedings of Eurographics Rendering Workshop 1994*. 147–162.

Carsten Wächter and Alexander Keller. 2006. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of the 17th Eurographics Workshop on Rendering*. 139–149.

Barry Wagner. 2013. The Evolving Mobile Platform. In *JEDEC Mobile Forum 2013*.

Ingo Wald. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. Ph.D. Dissertation. Sarrland University.

Ingo Wald, Carsten Benthin, and Philipp Slusallek. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003*. 77–86.

Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* 26, 1, Article 6 (2007), 6:1–6:18 pages.

Ingo Wald and Vlastimil Havran. 2006. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing 2006*. 61–69.

Ingo Wald, Thiago Ize, and Steven G. Parker. 2008. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics* 32, 1 (2008), 3–13.

Ingo Wald, William R Mark, Johannes Gunther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. 2009. State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum* 28, 6 (2009), 1691–1722.

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)* 20, 3 (2001), 153–164.

Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (1980), 343–349.

Sven Woop, Erik Brunvand, and Philipp Slusallek. 2006a. Estimating Performance of a Ray-Tracing ASIC Design. In *Proceedings of IEEE/EG Symposium on Interactive Ray Tracing 2006*. 7–14.

Sven Woop, Gerd Marmitt, and Philipp Slusallek. 2006b. 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*. ACM, 67–77.

Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24, 3 (2005), 434–444.

Zhefeng Wu, Fukai Zhao, and Xinguo Liu. 2011. SAH KD-tree construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG '11)*. 71–78.

Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics* 27, 5 (2008), 1–11.

Received December 2013; accepted February 2014