SGRT: A mobile GPU architecture for real-time ray tracing

Conference Paper · July 2013

DOI: 10.1145/2492045.2492057

CITATIONS
38

READS
344

9 authors, including:

Won-Jong Lee
Intel
52 PUBLICATIONS **254** CITATIONS
SEE PROFILE

Youngsam Shin
Samsung
29 PUBLICATIONS **195** CITATIONS
SEE PROFILE

Jaedon Lee
Ewha Womans University
20 PUBLICATIONS **127** CITATIONS
SEE PROFILE

Jinwoo Kim
Yonsei University
14 PUBLICATIONS **131** CITATIONS
SEE PROFILE

SGRT: A Mobile GPU Architecture for Real-Time Ray Tracing

Won-Jong Lee[†*] Youngsam Shin[†] Jaedon Lee[†] Jin-Woo Kim[‡] Jae-Ho Nah[§]
Seokyoon Jung[†] Shihwa Lee[†] Hyun-Sang Park[‖] Tack-Don Han[‡]

[†]SAMSUNG Advanced Institute of Technology [‡]Yonsei University
[§]University of North Carolina at Chapel Hill [‖]National Kongju University

Figure 1: *Full ray traced (including shading and bilinear filtered texturing) scenes by our GPU on a FPGA platform: Ferrari (left, 210K triangles) and Fairy (right, 170K triangles).*

Abstract

Recently, with the increasing demand for photorealistic graphics and the rapid advances in desktop CPUs/GPUs, real-time ray tracing has attracted considerable attention. Unfortunately, ray tracing in the current mobile environment is very difficult because of inadequate computing power, memory bandwidth, and flexibility in mobile GPUs. In this paper, we present a novel mobile GPU architecture called SGRT (Samsung reconfigurable GPU based on Ray Tracing) in which a fast compact hardware accelerator and a flexible programmable shader are combined. SGRT has two key features: 1) an area-efficient parallel pipelined traversal unit; and 2) flexible and high-performance kernels for shading and ray generation. Simulation results show that SGRT is potentially a versatile graphics solution for future application processors as it provides a real-time ray tracing performance at full HD resolution that can compete with that of existing desktop GPU ray tracers. Our system is implemented on an FPGA platform, and mobile ray tracing is successfully demonstrated.

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

Keywords: ray tracing, GPU, mobile

1 Introduction

Recently, mobile graphics have become increasingly important. A variety of 3D graphics applications, including UIs, games, and entertainment in embedded devices, such as smart phones, tablet PCs, digital TVs, and consumer electronics, are being highlighted. Increased interest in mobile graphics drives manufacturers of application processors (APs) [Exynos 2013] [Tegra 2013] [Snapdragon 2013] to invest more of their resources in GPUs [iPad4 2013]. These trends in mobile graphics will continue with the explosive growth of the mobile market segment. In the near future, it will be shown that the more realistic and immersive applications, such as AR/MR, digital imaging, and natural user interfaces (NUIs), give a better user experience by combining graphics and other technologies (i.e., vision, image, and camera) [Peddie 2011].

Ray tracing is a potential rendering technique for these future mobile graphics applications. Geometry, material, and illumination information will be captured from the objects in real worlds by the 3D camera of the future mobile devices [Goma 2011]. Captured information can be directly applied in a ray tracer because it is a physically based rendering algorithm. These ray-traced objects will be naturally mixed with real-world objects and make the AR/MR application more immersive [Kan and Jaufmann 2012]. In addition, ray tracing can be suitable for a mobile display whose screen is relatively small, because the first-order ray tracing performance scales linearly with the number of screen pixels, not the number of scene primitives [Spjut et al. 2012]. Furthermore, the ray tracing algorithm has both control flow- and data parallel-intensive operations, which can be appropriately distributed to the heterogeneous platforms because the multi-/many-core CPUs/GPUs architecture will be strengthened in the future mobile SoC environment [HSA 2013].

However, there has been minimal research on real-time ray tracing in the mobile environment [Spjut et al. 2012] [Kim et al. 2012a], in contrast to the many research studies that have addressed the desktop CPUs/GPUs/MIC platform. Though mobile graphics capabilities and performance have advanced considerably in recent years, real-time ray tracing in current mobile GPUs is very difficult for the following reasons. First, the floating-point computational power is inadequate. Real-time ray tracing (at 30 fps) a real-world application at 720p HD resolution (1280×720) requires a performance of at least 300 Mrays/s (about 1∼2 TFLOPS) [Slusallek 2006], but the theoretical peak performance of the current flagship mobile GPU is no more than 378 GFLOPS (ARM Mali T678 [ARM 2013a]). Second, the execution model of the mobile GPU is the SIMD-based multithreading or SIMT, which is not suited for incoherent ray trac-

*e-mail:joe.w.lee@samsung.com

ing, because it causes branch and memory access divergence [Gribble and Ramani 2008] [Kim et al. 2012a]. Third, the memory bandwidth can constitute a bottleneck, because the external DRAM is not dedicated for the GPUs, but has to be shared with CPUs in the mobile SoC. According to recent results of the OpenCL ray tracing benchmarks [KISHONTI 2013], the ray tracing at 2048×1024 resolution on Google Nexus 10 (Exynos 5250, ARM Mali T604 GPU) and SonyXperia Z (Qualcomm Snapdragon S4 Pro, Adreno 320 GPU) recorded the lowest scores.

In this paper, we propose a new mobile GPU architecture, which is called SGRT (Samsung reconfigurable GPU based on Ray Tracing). SGRT can resolve the above problems and realize real-time mobile ray tracing by combining two components. First, it has a fast compact hardware engine (3.32 mm^2 per core, 65 nm) that accelerates the traversal and intersection (T&I) operations, which are computationally dominant in ray tracing. Our hardware, called the T&I unit, is based on a T&I engine investigated in [Nah et al. 2011]; however, the current T&I unit differs in that it consists of newly proposed parallel-pipelined traversal units, which can reduce unnecessary data transfer between pipeline stages, and improve performance. The new T&I unit allows efficient incoherent ray tracing by combining the proposed new feature and the high performance features, such as the MIMD execution model and ray accumulation unit (RAU), proposed in [Nah et al. 2011]. Second, it employs a flexible programmable shader called SRP (Samsung Reconfigurable Processor), which supports software ray generation and shading (RGS), which was developed in [Lee et al. 2011]. Unlike the conventional mobile GPU, the SRP can support both control-flow and highly parallel data processing, which makes the application of high performance RGS kernels possible. Although we limit the range of this paper to the static scene, the design of the SGRT architecture is such that it can be extended to deal with dynamic scenes. The concept of SGRT was previously proposed in an extended abstract [Lee et al. 2012].

The performance of SGRT is evaluated through cycle-accurate simulations, and the validity of the architecture is verified by FPGA prototyping. The experimental results show that SGRT can provide real-time ray tracing at full HD resolution (Fairy scene, 34 fps at 4 SGRT cores (500 MHz T&I unit and 1 GHz SRP)) and compete with not only the mobile ray tracing processor but also the desktop GPU/MIC ray tracer in terms of the performance per area. Moreover, the full SGRT system is implemented at the Verilog/RTL level, and demonstrated on an FPGA platform as a mobile ray tracing GPU.

To summarize, the main contribution of this paper are:

- A new T&I hardware architecture based on an area-efficient parallel pipelined traversal unit.
- High performance RGS kernels efficiently mapped onto a programmable shader.
- Implementation of the full system at the RTL level and demonstration of a mobile ray tracing GPU on an FPGA.

2 Related Work

In this section, we describe the research related to SGRT, which comprises studies on parallel ray tracing, dedicated hardware, and mobile ray tracing.

2.1 Parallel ray tracing

A variety of parallelizing approaches has been tried to accelerate the rendering speed efficiently in several platforms, such as CPUs/GPUs/MIC. Ernst [2012] classified these approaches into the three types in terms of the object to be parallelized when the ray-box intersection is performed. First, the multiple rays to be intersected to a single box can be parallelized in SIMD fashion, which is called *packet tracing*. The packet tracing approach has been widely utilized by employing CPU's SIMD intrinsics [Wald 2007] or GPU's SIMT multithreading [Garanzha and Loop 2010], but it is not suitable for the secondary rays due to the issue of branch divergence. In order to overcome this drawback, a ray reordering, sorting, or scheduling algorithm to improve ray coherency has to be applied [Garanzha and Loop 2010]. Second, a single ray-based SIMD traversal approach, *multi-BVH*, can also be used [Ernst 2008] [Tsakok 2009]. Multi-BVH parallelizes not the rays but the boxes of the nodes in the BVH tree. This method is a better solution for incoherent rays and scales well with larger SIMD widths. However, the complexity of both the tree structures and build algorithm can be increased. Third, the intersection tests of the single ray and box can be parallelized independently. This can be done in an MIMD architecture. Each ray is treated as a separate thread in this architecture, which means that improved hardware utilization can be obtained when the incoherent rays are processed. TraX [Spjut et al. 2009], MIMD threaded multiprocessor [Kopta et al. 2010], T&I engine [Nah et al. 2011], and this paper belong in this category.

2.2 Dedicated hardware

To date, various dedicated hardware research studies have been conducted on high performance ray tracing. SaarCOR [Schmittler et al. 2004] was the first ray tracing dedicated pipeline, which was composed of a ray generation/shading unit, 4-wide SIMD traversal unit, list unit, transformation unit, and intersection test unit. Woop et al. [2005] proposed a programmable RPU architecture, in which the ray generation, shading, and intersection were executed on a programmable shader. Later, adding a node update unit to the RPU architecture, they proposed the D-RPU architecture for dynamic scenes [Woop 2007]. SaarCOR, RPU, and D-RPU are all based on packet tracing, which can lower the performance in the case of incoherent rays due to the poorer SIMD efficiency, as mentioned in the previous section. To resolve this problem and obtain an improved SIMD utilization, Gribble and Ramani [2008] proposed a wide-SIMD ray tracing architecture with a stream filter that can filter the active rays in a ray packet during each step. In contrast, TRaX [Spjut et al. 2009] and MIMD TM [Kopta et al. 2010] used single ray thread execution rather than packet tracing, employing a small and lightweight thread processor and shared functional units, and shared caches. The T&I engine [Nah et al. 2011] is dedicated hardware for accelerating the T&I operation, which is composed of traversal units using an ordered depth-first layout and three-phase intersection units; each unit commonly includes a ray accumulation unit for latency hiding. However, the target of the above architectures is not the mobile but the desktop environment. Finally, the CausticRT [2013] is commercial ray tracing hardware targeted desktop to mobile platforms, which has been released in the market, but whose detailed architecture has not been published.

2.3 Mobile ray tracing

Relatively little research has been carried out on mobile ray tracing. Nah et al. [2010] implemented an OpenGL|ES-based ray tracer on existing mobile CPUs and GPUs. Their approach is a hybrid method to assign the tasks for kdtree construction to CPUs, and the tasks for ray traversal and shading to GPUs. However, the rendering performance was very limited (predicted to be $1 \sim 5$ fps on mobile CPUs and GPUs). In contrast, Spjut et al. [2012] proposed a version of the MIMD TM [Kopta et al. 2010] in which the number of thread

multiprocessors and active threads was reduced, making it suitable for mobile devices. Kim et al. [2012a] also proposed a dedicated multi-core processor for mobile ray tracing, which can be configured to both SIMT and MIMD modes. However, the above architectures could not provide a sufficiently high performance (less than 30 Mrays/s) for real-time ray tracing of real-world applications.

3 Proposed System Architecture

In this section, we describe the overall architecture of the proposed ray tracing system. First, we present the basic design decision and review the SGRT system organization, and then explain the T&I unit and the SRP, the core components of the SGRT, in detail.

3.1 Basic design decision

Mobile Computing Platform: We designed our system using a mobile-friendly strategy. The mobile computing platform [Exynos 2013] [Tegra 2013] [Snapdragon 2013] integrates multi-core CPUs, many-core GPUs, and dedicated hardware circuits into a single system. The system bus and shared caches minimize communication costs. In order to utilize this heterogeneous computing platform, the ray tracing algorithm is carefully analyzed in terms of computing, memory access pattern, and power consumption, and then partitioned into hardware and software.

Dedicated Hardware Units: The T&I operations are the most resource-consuming stages. Hardwired logic is appropriate because T&I are repeating operations and require low-power consumption. In general, fixed function hardware increases computational performance by exploiting hardwired or customized computation units, customized wiring/interconnect for data movement, reduced unused resources, and removed instruction fetching/decoding overhead at some cost in general-purpose processor. In addition, this approach is more advantageous in terms of energy efficiency which is a critical issue in mobile platfom. Due to battery capacity and heat dissipation limits, for many years energy has been the fundamental limiter for computational capability in mobile platform, which might include as many as 10 to 20 accelerators to achieve a superior balance of energy efficiency and performance. According to [Borkar and Chien 2011] [Hameed et al. 2010], units hardwired to a particular data representation or computational algorithm can achieve 50∼500x greater energy efficiency than a general-purpose register organization. Therefore, we designed a T&I unit with a fixed pipeline.

Programmble Shader: We assign the RGS operations to the programmable shader of existing GPUs in the mobile computing platform, because the RGS requires a certain degree of flexibility to support various types of ray and shading algorithms.

Acceleration Structure: We selected the BVH tree as the acceleration structure for various reasons. First, it enabled us to simplify the hardware structures. In contrast to the kdtree, where a primitive might be included in multiple leaf nodes, the BVH is an object hierarchy where a primitive is included in a leaf node. Therefore, the BVH negates the need for LIST units [Schmittler et al. 2004] [Woop et al. 2005] [Nah et al. 2011] to manage primitives. Second, even though the current SGRT was tested using static scenes, we selected the BVH for our future architecture because it can support dynamic scenes.

Static and Dynamic Tree Build: We assigned the tree build operation to multi-core CPUs that fully support control-flow and multi-level caches, because the tree build is an irregular operation that includes sorting and random memory access. Current our GPU is tested using static scenes. However, we designed our architecture with the intention of extending it to handle dynamic scenes, as we mentioned in the introduction. For rendering dynamic scenes, the tree has to be updated in every frame. If the tree construction is performed by CPUs, the bus between the CPUs and the SGRT cores can become a bottleneck. This problem might be alleviated by an asynchronous BVH build [Wald et al. 2008] in combination with a BVH tree refit, because the tree build and transfer can be hidden during the rendering with the refitted trees. The performance of mobile CPUs (multi-core ARM CPUs) is sharply increasing (the number of cores and clock frequency) and SIMD intrinsics, such as NEON [ARM 2013b], are already available. Therefore, we expect that real-time tree build on mobile CPUs will soon be realized in our future research.

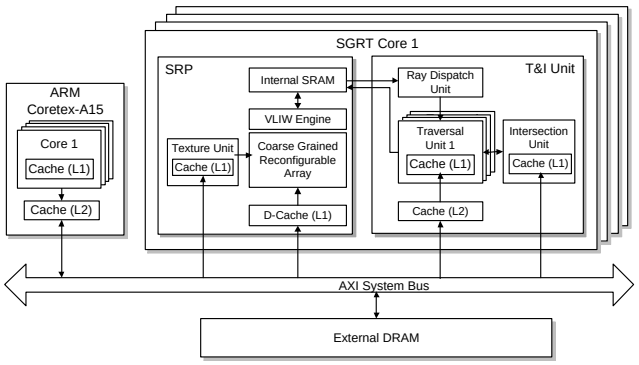

Figure 2: *Overall architecture of the proposed ray tracing system.*

Single Ray-based Architecture: As in [Nah et al. 2011], our T&I unit is based on single-ray tracing because it is more robust for incoherent rays than SIMD packet tracing. According to [Mahesri et al. 2008], SIMD architectures showed a poorer performance per area than MIMD architectures. Therefore, our method processes each ray independently via a different data path, as in MIMD approaches.

3.2 SGRT system organization

Figure 2 illustrates the overall architecture of the proposed system, which is composed of multi-core CPUs and the SGRT cores. Each SGRT core has a T&I unit and SRP. The goal of the proposed system is to utilize the resources of mobile computing platforms efficiently and achieve a higher performance per area, and finally provide real-time ray tracing in the mobile environment.

Ray traversal and intersection tests are performed by the T&I unit. The T&I unit consists of a ray dispatch unit (RD), multiple traversal unit (TRV), and an intersection unit (IST). We limit the primitive type to triangle to allow a simple hardware structure, and employ Wald's intersection method [2004], which is the most cost-effective algorithm [Nah et al. 2011].

Unlike previous traversal architectures [Nah et al. 2011], the new TRV is a parallel pipelined architecture that splits a single deep pipeline into three different sub-pipelines to reduce the unnecessary transferal of the ray between pipeline stages. Through this architecture, the ray can traverse a tree with minimized latencies and higher cache locality. The TRV architecture is described in detail in section 3.3.1. The abstract of this parallel pipelined TRV study was previously announced in [Kim et al. 2012b].

RGS are performed by the programmable shader. We employed the SRP, which is the shader core of the GPUs developed in [Lee et al. 2011]. The SRP consists of a VLIW (Very Long Instruction Word)

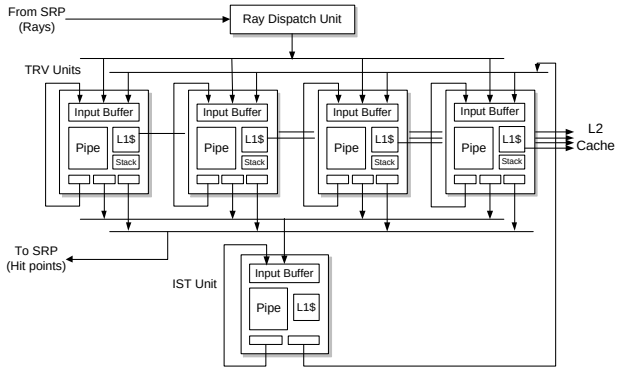

Figure 3: *Internal architecture of the T&I unit.*

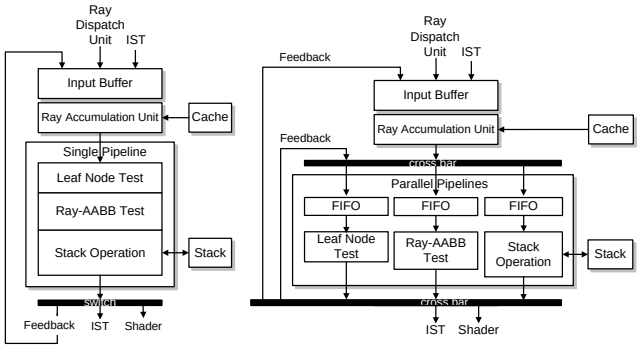

Figure 4: *The conventional single- (left) and our parallel- (right) pipelined traversal unit.*

engine, which supports control-flow, and a CGRA (Coarse Grained Reconfigurable Array), which is responsible for highly parallel data processing. RGS kernels are efficiently mapped onto this CGRA by using high performance features, such as software pipelining and SIMD, and multiple rays are executed in parallel. The structure is described in detail in section 3.4.

TRVs and IST require data fetching. For efficient memory access, we assigned a level-one (L1) cache memory and a ray accumulation unit (RAU) [Nah et al. 2011] to each pipeline. The RAU is used to prevent pipeline stall by storing rays that induce a cache miss, and therefore efficient concurrent processing of multiple rays in pipeline stages can be achieved with this unit. The level-two (L2) cache for multiple TRVs between off-chip memory and the L1 cache reduces memory traffic and contributes to scalable performance, as in [Nah et al. 2011]. The CPU, T&I Unit, and SRP are connected by an AXI system bus.

The T&I unit and the SRP can communicate through not an AXI-system bus but a FIFO-based dedicated interface, because using an external DRAM as the interface would be a bandwidth burden for the memory system. Hence, we configure on-chip FIFO buffers in the SRP to store 64 rays and hit points for ray buffers (ray generation to T&I) and hit buffers (T&I to shading). To allow concurrent execution of the shading kernel and T&I unit, the hit buffers are configured as double buffers. Similarly, the ray buffers are configured as triple buffers for concurrent execution of the primary/secondary ray generation kernels and T&I unit. The size of a ray and a hit point is 64 bytes and 24 bytes, respectively. Therefore, a total of 12 KB ($64\times(2\times64$ bytes$+3\times24$ bytes)) of SRAM is required.

3.3 Dedicated hardware for T&I

Figure 3 shows the detailed T&I architecture for performing the traversal and intersection operation. The T&I unit consists of a ray dispatch unit (RD), four traversal units (TRV), and an intersection unit (IST). Each unit is connected by an internal buffer that passes a ray from one unit to the others. The RD fetches a ray from the ray buffer and first dispatches it to an idle TRV. Because only one ray can be supplied for each cycle, rays are supplied to the TRVs in a FIFO order. The ratio of the TRV to the IST is dependent on the ray and scene characteristic. We found that a ratio of 4:1 was appropriate after testing with various scenes, as proposed in [Schmittler et al. 2004].

BVH tree traversal is performed by TRVs, each of which consists of a memory processing module (input buffer, RAU, L1 cache, short stack) and computation pipeline. The output of the TRV pipeline branches into three paths: a feedback loop path to the input buffer for iterative visits to inner nodes; an output path to send a ray to the IST when a ray reaches a leaf node; and an output path to send a ray to the SRP when a ray finishes BVH tree traversal. The computation pipeline of the TRV consists of 4 floating-point adders, 4 floating-point multipliers, and 11 floating-point comparators.

The ray-triangle intersection test is performed by the IST, which, similarly to TRVs, consists of a memory processing module (input buffer, RAU, L1 cache) and computation pipeline. The output of the IST pipeline branches into two paths: a feedback loop path to the input buffer for iterative testing of triangles in the same leaf node; and an output path to send a ray to the TRV for visiting the next BVH nodes. According to Wald's intersection algorithm [2004], the computation pipeline of the IST consists of 12 floating-point adders, 10 floating-point multipliers, 1 reciprocal unit, and 5 floating-point comparators.

3.3.1 Parallel pipelined traversal unit

We propose a new traversal unit based on parallel pipelines. Figure 4 compares the conventional single pipeline [Nah et al. 2011] and proposed parallel pipelines. The traversal operation consists of three sub-pipelines: 1) Fetching a tree node and testing whether the node is a leaf or inner node (TRV_PRE); 2) testing the ray-AABB intersection (TRV_AABB); and 3) the stack operation (TRV_POST). A conventional single deep pipeline is designed to connect these operations serially to increase the throughput of ray processing per unit time. However, traversal operations involve non-deterministic changes in the states of a ray. For example, if the result of TRV_PRE is inner node, then the next sub-pipeline TRV_AABB is executed. If the result of TRV_PRE is leaf node, the corresponding ray should be transferred to the IST without executing TRV_AABB and TRV_POST. To handle these branching situations, a single deep pipeline has to bypass the rays to inactive the sub-pipelines without executing any operations, thereby increasing the overall latency.

The right-hand side of Figure 4 shows the parallel traversal unit proposed to overcome the abovementioned problems. The sub-pipelines for the three stages are independent, and the outputs constitute a crossbar, through which the rays are fed back to the input stage. The crossbar does need to be connected fully, but only to the paths for each branching between the sub-pipelines. Therefore, only minimal hardware needs to be added to construct the parallel pipelines. After each operation, the processed ray is fed back through the output crossbar or passed on for shading or the intersection test, according to the next operation to be processed. Unlike the existing single-pipeline structure, the proposed structure is able

to reduce the transfer of inessential data, because the ray is immediately fed back to its sub-pipelines when its state is changed. Furthermore, the congurability of this structure enables us to improve the performance in a cost-effective manner via selective sub-pipeline plurality under high usage.

The proposed TRV has a purpose similar to that of the shader architecture of modern GPUs [ARM 2013a]. Both our approach and GPUs aim to reduce the thread (ray in our case) latency and improve throughput. The GPU shader architecture takes advantage of multithreading, which issues each thread to the appropriate functional unit according to the instruction to be executed. Dedicated hardware logic is equipped to schedule the instructions and threads for latency hiding and dependency resolving. One way in which our approach differs from others is that our TRV does not need scheduling logic, because the ray can be routed to the destination immediately after the execution of the sub-pipelines.

3.4 Reconfigurable processor for RGS

In order to support the flexible RGS operation, we utilize a programmable shader called SRP (Samsung Reconfigurable Processor). The SRP is variously used in OpenGL|ES-based mobile GPUs [Lee et al. 2011] and a multimedia DSP [Song et al. 2012]. The multimedia SRP has already been verified by its inclusion in commercial application processors [Exynos 2013]. The mobile GPU target SRP core shows a theoretical peak performance of 72.5 GFLOPS per core (1 GHz clock frequency, 2.9 mm^2 at 65 nm), which is already comparable with commercial mobile GPUs [ARM 2013a]. The SRP is a flexible architecture template that allows a designer to generate different instances easily by specifying different configurations in the target architecture. The SRP also supports full programmability (standard C language) to help application developers develop their target application easily. In this section, we briefly describe the SRP architecture and its programming model, and then explain the RGS kernels that are efficiently mapped onto the SRP.

3.4.1 CGA/VLIW model

Figure 5 depicts the SRP architecture, which includes a tightly coupled very long instruction word (VLIW) engine and coarse-grained reconfigurable array (CGRA). Integrating the VLIW engine and the CGRA in a single architecture has an advantage as compared to the state-of-the art DSP and shader of mobile GPUs. The VLIW engine is designed for general-purpose computations, such as function invocation and branch selection. In contrast, the CGRA is intended only for efficiently executing the computation-intensive kernels, and therefore it can accelerate the rest of the application like a coprocessor. Communication between the processor and the configurable accelerator results in programming difficulties and communication overhead, which can greatly reduce overall performance. Combining the host processor and the accelerator leads to simplified programming, and removes the communication bottleneck. In addition, the CGRA includes components similar to those used in VLIW engines.

The components inside in FU, Figure 5, include: the local buffer, the ALU, input and output multiplexers and a RF. The local data storage is beneficial for code mapping as several iterations of local data processing can be executed in a single FU without the need of transporting intermediate data through the interconnect. Beside of power aspects this virtually increases the feasible depth of data paths.

The kernels can be optimally scheduled for the VLIW/CGRA at compile time. The compilers core technology is a modulo scheduling algorithm [Mei et al. 2003], which can map loops onto the SRP in a highly parallel way. The objective of modulo scheduling, a widely used software pipelining technique, is to execute multiple iterations of the same loop in parallel. To achieve this, the compiler constructs a schedule for one loop iteration such that this same schedule repeats at regular intervals with respect to intra- and inter-iteration dependences and resource constraints. The initiation interval shows the number of cycles that elapse before the next iteration of the loop starts executing.

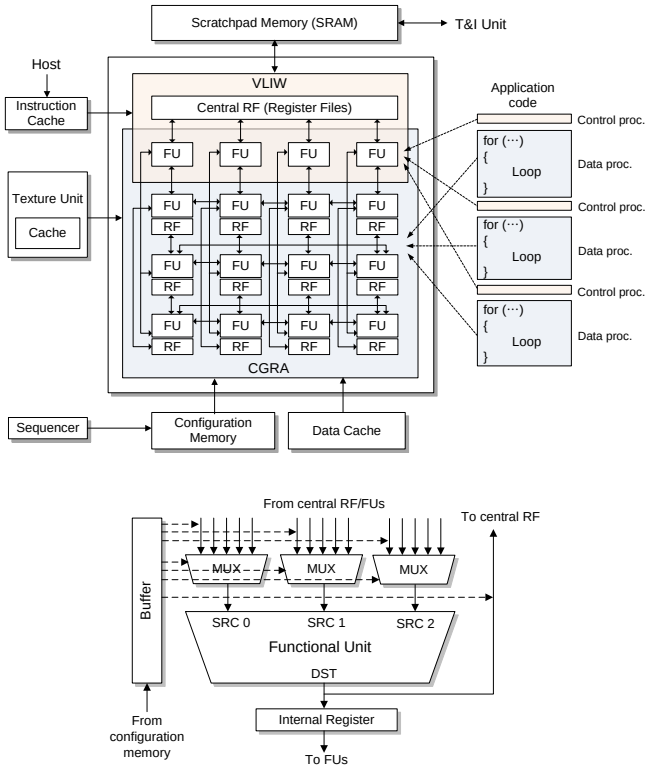

Figure 5: *The SRP architecture including VLIW/CGRA and the mapping of application codes (up), and a functional unit in CGRA (down).*

Instruction sets for executing kernels, such as arithmetic (SIMD and scalar), special function, and texture operations, are properly implemented in each functional unit in the VLIW/CGRA. A special interface to invoke external hardware can be used for driving the texture mapping unit. The SRP has an internal SRAM (240 KB) for stream data and a data cache (4 KB) for random accessible data.

3.4.2 Compilation and simulation

Figure 6 shows our complier and simulation framework. A design starts from a C-langurage description of the application. We write a ray tracing application composed of the non-kernel and the kernel codes. Our compiler front-end parses these sources and generates IR (Intermediate Representations) codes suited for VLIW processor. The compiler outputs also profiling information on all function calls in the source. It uses symbolic instructions and flattened function calls. However, loop descriptions are preserved. These IR codes are then converted into the second XML-based intermediate codes, DRE. The resulting DRE files are split up in two parts: 1) Non-kernel codes to be optimized for the VLIW engine, 2) kernel codes to be optimized for the CGA datapath.

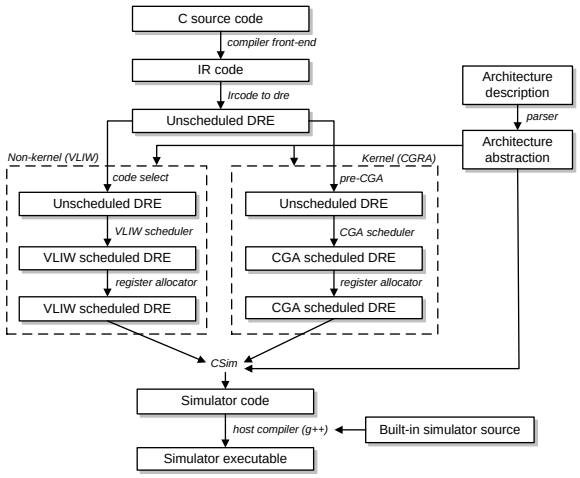

Figure 6: *Overall compiler/simulator framework for SRP.*

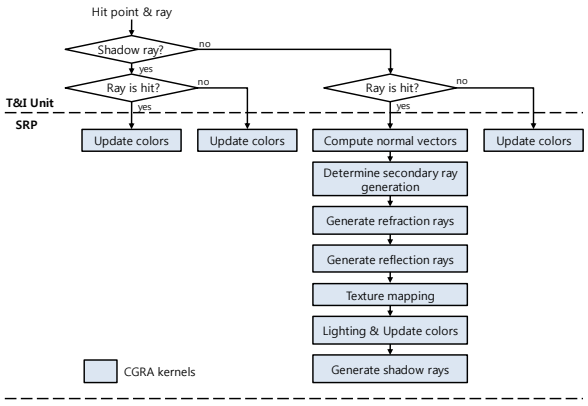

Figure 7: *Execution flow of RGS kernels.*

Non-kernels and kernels are manipulated by several mudules of our SRP compiler. The SRP compiler outputs the scheduled DRE files through a few compilation steps such as code selection, ILP scheduling, register allocations and modulo schduling. For further processing of the code, the target architecture is described in an XML-based language. The parser and abstraction step transform the architecture into an internal graph representation. This is also taken as an input in each compilation step.

We develop a source generator, called *csim*, which is a meta-program to generate C++ cycle accurate simulator using the architecture abstraction and the scheduled DRE files as inputs. After the host compiler, g++, compiles these generated simulator sources with built-in codes, the executable can be obtained. Finally, designers can use this simulator to get quality metrics for the architecture instance under test.

3.4.3 High performance kernels for RGS

There are certain key points for achieving high performance in the SRP parallel computing model, similarly to CUDA [2013] and OpenCL [2013]. 1) *Launch a kernel with a loop having a massive number of iteration counts*, which can involve enough steady states of software pipeline to hide the overhead of the prolog/epilog. However, the iteration counts of a loop (i.e., the number of stream data to be processed in a loop) can be limited by the available size of the internal SRAM of the SRP. Therefore, the appropriate loop size has to be selected. 2) *Avoid a branch in the kernel loop* that is to be scheduled by the modulo scheduler. When the control flow diverges in a kernel loop due to a conditional branch, the compiler serializes each branch path taken, which can lead to performance loss. From the perspective of kernel programming, designing an algorithm with a simple control structure that minimizes unpredictable, data-dependent branches can enhance the performance markedly. 3) *Aggressively utilize the SIMD intrinsics* that are supported by the SRP instruction set architecture.

According to point 1), we choose the size of the kernel loop (i.e., the size of a ray packet) to be 64, according to the results of various experiments and speculative consideration of both CGRA efficiency and SRAM cost. According to point 2), we simplify the overall algorithm to control-flow-free, and therefore ray tracing recursion is converted to iteration loops based on the job-queue controlled by a simple scheduler. We implement the optimized RGS kernels as follows.

Shading kernels: Several branches can occur in the shading operation according to the types of hit points and rays. According to point 2), these should be appropriately classified and then partitioned into control-flow statements and kernel blocks. Control flow statements can be handled by the VLIW engine, and the kernels are executed in a CGRA. The type of ray determines whether a secondary ray is generated. The type of secondary ray to be generated also depends on the types of hit object material. Therefore, we split the original loop in the shading function into small loops for each branch, which became kernel loops, as shown in Figure 7. To reduce classification overhead in the SRP, the T&I classifies the hit points according to the types of ray and hit results. The SRP executes different kernels for these four cases (shadow ray & hit, shadow ray & no hit, other ray & hit, and other ray & no hit). The case of other ray & hit is the most computation-intensive. This case can be classified into the several sub-cases: 1) Compute normal vector in barycentric coordinates; 2) determine whether the secondary ray has to be generated for each primary ray; 3) texturing; and 4) lighting computation. For these sub-cases, kernels also are implemented and invoked in order by the CGRA.

Ray generation kernels: These kernels are intuitively implemented because ray generation does not involve any branches. According to the type of ray to be generated (primary, shadow, reflection, and refraction), different kernels are executed accompanied by shading kernels.

According to point 3), every kernel utilizes the SIMD parallelization. A 4-way SIMD intrinsic for arithmetic, logical, and relative operation is supported by the SRP, like the commercial GPUs. The RGS is an SIMD-friendly operation (having vector components for position (xyzw) and color (rgba)), and therefore the operations in RGS kernels are vectorized with these intrinsics. This is very similar to the conventional vertex and pixel shader in a modern GPU program.

4 Results and Analysis

In this section, we describe the functional validation and performance evaluation of SGRT and the implementation of our system on an FPGA. First, we estimate the hardware complexity and area consumption and analyze the simulation results, and then, finally demonstrate the FPGA prototype.

Table 1: *Complexity of a T&I unit by the number of floating-point units and the required on-chip memory (ADD : adder, MUL : multiplier, RCP : reciprocal unit, CMP : comparator, RF : register file, L1 : L1 cache, L2 : L2 cache).*

	ADD	MUL	RCP	CMP	RF (KB)	L1 (KB)	L2 (KB)
4 TRV	24	24		44	49	64	128
1 IST	12	10	1	5	8	128	
Total	36	34	1	49	57	192	128

Table 2: *Area estimation of a T&I unit (65 nm process, FP : floating point, INT : integer, OH : overhead).*

Funtional units	Area (mm^2)	#	Total area (mm^2)	Memory units	Area (mm^2)	#	Total area (mm^2)
FP ADD	0.003	36	0.11	TRV L1	0.06	4	0.24
FP Mul	0.01	34	0.34	TRV L2	0.64	1	0.64
FP RCP	0.11	1	0.11	IST L1	0.64	1	0.64
FP CMP	0.00072	49	0.04	4K RF	0.019	15	0.29
INT ADD	0.00066	20	0.01				
Control/Etc.			0.14				
Wiring O/H							0.77
Total							3.32

4.1 Hardware complexity and area estimation

Tables 1 and 2 show the hardware complexity and estimated area for the T&I unit, respectively. The hardware setup was structured as follows. TRVs have a 16 KB 2-way L1 cache, and a 128 KB 2-way L2 cache divided into eight banks. The IST has a 4-way 128 KB L1 cache. The block size of all caches is 64 bytes. Resister files (RF) are assigned to in-out buffers, RAUs, traversal stacks, and pipeline latches.

We carefully estimated the area of the T&I unit, in a way similar to that used in [Nah et al. 2011]. We assumed 65 nm technology and obtained the area estimates for arithmetic units and caches from [Kopta et al. 2010] and CACTI 6.5 [Muralimanohar et al. 2007]. Stack operations, pipeline controls, RAU 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.

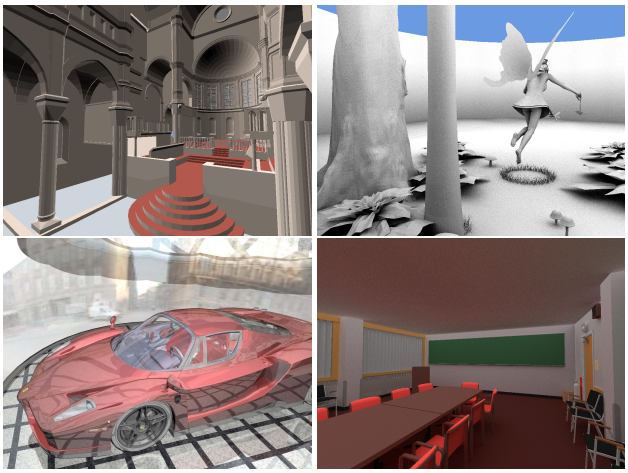

Figure 8: *Test scenes: Sibenik with primal rays, Fairy with ambient occlusion rays, Ferrari with forced specular rays (2-bounce), and Conference with diffuse inter-reflection rays.*

Table 3: *Simulation results of RGS kernels executed on SRP (4 cores at 1 Ghz clock, cache size : texture (32KB), data (4KB), FSR : forced specular ray (2 bounce)).*

Test scene	Ray type	Cache hit rate (%) Texture	Data	Bandwidth (GB/s)	Performance (Mrays/sec)
Sibenik	Primary	-	96.76	0.5	182.11
(80K tri.)	FSR	-	91.24	1.9	172.25
Fairy	Primary	93.25	96.87	0.8	175.66
(179K tri.)	FSR	81.49	94.91	1.9	147.45
Ferrari	Primary	86.12	98.09	0.6	183.28
(210K tri.)	FSR	75.95	95.71	2.0	163.67
Conference	Primary	-	98.44	0.2	198.32
(282K tri.)	FSR	-	95.72	0.8	158.79

Table 4: *Performance comparison of single pipelined (SPTRV) and parallel pipelined traversal unit (PPTRV).*

Test scene	Ray type	Average steps SPTRV	PPTRV	Mrays/sec SPTRV	PPTRV	Ratio to SPTRV
Sibenik	Primary	61.30	23.12	27	33	1.24
(80K tri.)	AO	36.55	14.87	48	56	1.15
	Diffuse	81.62	29.93	11	15	1.40
Fairy	Primary	70.86	28.02	22	28	1.27
(179K tri.)	AO	31.53	12.43	52	62	1.20
	Diffuse	51.72	18.99	19	24	1.31
Ferrari	Primary	68.86	25.52	23	29	1.26
(210K tri.)	AO	30.64	11.32	54	64	1.18
	Diffuse	92.24	59.20	20	25	1.25
Conference	Primary	44.66	15.54	36	46	1.30
(282K tri.)	AO	17.23	5.88	99	121	1.23
	Diffuse	43.06	14.59	33	44	1.35

We obtained this percentage from the ratio of the front-end area to that of the execution area (0.233:1) in [Mahesri et al. 2008]. In addition, wiring overhead due to place and route, choice of logic gates, and other optimizations, should be taken into account. Woop [2007] and Mahesri [2008] used 29-33% wiring overhead, and therefore, we added a 30% one-level overhead. From this assumption, we finally obtained 3.32 mm^2 at 65 nm for the estimated area of the T&I unit.

4.2 Simulation results

We conducted a cycle-accurate simulation for functional verification and performance evaluation. We created a cycle-accurate simulator for the T&I unit to simulate its execution. This simulator provides the rendered images, total execution cycles, hardware utilization, cache statistics, and expected performance. We adopted a GDDR3 memory simulator from GPGPUsim [Bakhoda et al. 2009] to execute accurate memory access. To evaluate the performance of the SRP, we utilized an in-house compiled simulator, csim, which simulates the IR (intermediate representation) codes generated by the SRP compiler tool-chain, and then provides total execution cycles, IPC, FU utilization, and cache statistics.

SGRT uses four cores (four T&I units and four SRP cores). In a T&I unit, the number of TRVs and ISTs is four and one, respectively. A TRV has 16 stacks. Therefore, the maximum number of rays that can be simultaneously processed is 256 (4 SGRT cores $\times$ 4 TRV units per core $\times$ 16 stacks per TRV). We configured the clock frequency for the T&I unit and the SRP at 500 MHz and 1 GHz, respectively. We configured a 1 GHz clock and 32 bit 2-channel which is close to LPDDR3 memory, similar to those of the mod-

Table 5: *Simulation results of T&I unit (Four units at 500 MHz clock).*

Test scene	Ray type	Utilization (%)		Average steps		Cache hit rate (%)			Bandwidth (GB/s)	Performance (Mrays/sec)	Ratio to Tesla	Ratio to Fermi
		TRV	IST	TRV	IST	TRV L1	TRV L2	IST L1				
Sibenik	Primary	89	52	23.12	3.10	99	41	99	1.1	132	1.13	0.54
(80K tri.)	AO	92	55	14.87	1.17	99	68	99	0.1	222	1.86	0.91
	Diffuse	49	38	29.93	4.50	72	65	88	2.6	61	1.30	0.65
Fairy	Primary	67	67	28.02	4.40	97	49	99	1.5	112	1.50	0.73
(179K tri.)	AO	61	79	12.43	1.98	94	84	99	0.4	249	2.69	1.52
	Diffuse	49	67	18.99	4.21	73	63	91	3.7	97	2.38	1.33
Conference	Primary	78	70	15.54	3.35	99	57	99	0.3	184	1.30	0.68
(282K tri.)	AO	86	55	11.76	0.12	99	63	99	0.1	485	3.61	1.71
	Diffuse	62	64	14.59	3.82	90	70	96	3.3	178	2.92	1.41

Table 6: *Performance comparison for the Fairy scene.*

	Desktop		Mobile		
	GPU (Optix) [OptiX 2013]	MIC [Wald 2012]	Multiprocessor [Spjut et al. 2012]	Multiprocessor [Kim et al. 2012a]	Ours
Parallelism	SIMT	Wide SIMD	MIMD	SIMT/MIMD	MIMD + loop parallelism
Resolution	1920×1200	1920×1200	1280×720	1280×720	1920×1080 (Full HD)
Platform	NVIDIA GeForce GTX 680	Intel MIC	Thead Multiprocessor	Reconfigurable SIMT	SGRT (H/W + shader)
Clock (MHz)	1006	1000	500	400	500 (H/W), 1000 (shader)
Process (nm)	28	45	65	90	65
Area (mm^2)	294	-	25	16	25
BVH type	SAH	SAH	BVH	-	SAH
FPS	37	29	9	2	34

ern mobile SoC [Exynos 2013]. We set the latency of the L1 and the L2 caches to 1 and 20 cycles, respectively. A BVH tree is built with a full SAH-based algorithm [Wald 2007] and supplied to the front-end of the simulator before the simulation. We chose four test scenes (Figure 8): Sibenik (80 K triangles), Fairy (174 K triangles), Ferrari (210 K triangles), and Conference (282 K triangles).

First, we tested the RGS performance. Table 3 shows the results. The shading process includes Phong lighting and bi-linear filtered texturing. We tested the RGS with two kinds of rays, primary and forced reflection (2-bounce), in order to see the influence of the ray coherency. All the scenes were rendered at a 1024×768 resolution. Our RGS kernels were shown to have a performance of 147∼198 Mrays/s due to the acceleration with the software pipelining and SIMD parallelization in the SRP. The data cache of the SRP to access the primitive data shows a relatively higher hit rate (91∼98%) with a smaller size (4 KB). In contrast, the hit rate of the texture cache (32 KB) was not high (76∼93%) as compared to the data cache, because the texture memory locality decreases after shooting secondary rays with lower coherency, which can be alleviated by texture compression or mipmap texturing for ray tracing [Park et al. 2011].

Table 4 tabulates the simulation results of the proposed parallel pipelined TRV (PPTRV) as compared to those of the conventional single pipelined TRV (SPTRV). To examine the influence of the ray data from Ailas ray tracer [2012], we used 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. All the scenes were rendered at a 1024×768 resolution. The performance values are averages from five representative viewpoints per scene. This setup is the same as that in [Aila et al. 2012] (Sibenik, Fairy, and Conference), except for the type of acceleration structure.

The numbers in the table are the average steps per ray and Mrays/s after the process is executed on a single T&I unit for each case. As can be seen in the table, PPTRV outperforms SPTRV by up to 40% (Sibenik, diffuse ray) and by 26% on average. This is because the PPTRV removes the bypassing in the SPTRV and increases the latency. This improvement in performance can be also seen in the average number of traversal steps: PPTRV reduces the number of steps by up to 2.6-fold. The additional hardware required for the PPTRV as compared to the SPTRV is only two 16-entry FIFO buffers and extra wires. The percentage of the increased area is only 3.6% (3.28 mm^2 → 3.32 mm^2 at 65 nm). Consequently, we found that the PPTRV is an efficient cost-effective traversal unit.

For AO rays, the performance improvement was lower than for other rays. This can be explained by the early termination of their traversal. The AO rays are treated as shadow rays, and they therefore terminate their traversal as soon as they find a hit primitive. This any hit termination process reduces the number of average steps and the number of bypassed rays in SPTRV.

Table 5 shows the performance results of the T&I units in the 4-SGRT cores. The test condition is the same as that shown in Table 4. As shown in the table, the T&I units obtained a performance of 61∼485 Mrays/s. As compared to the results of [Aila et al. 2012], our SGRT can compete with the ray tracer on the previous desktop GPU, Tesla, and Fermi architecture. In this comparison, our case does not include RGS, but this would not result in an unfair comparison because the RGS outperforms the T&I (e.g., RSG 182 Mrays/s, T&I 132 Mrays/s, Sibenik, primary ray) and would not constitute a bottleneck according to the results of the RGS performance (Table 3). The performance gap between the primary ray and the diffuse ray was narrow (except for Sibenik) because MIMD with an appropriately sized cache architecture is advantageous for incoherent rays, as shown in [Kopta et al. 2010]. The reason for the overall improved performance was that the current T&I unit combines the enhanced traversal unit and the advantages of previous work (fixed pipeline, RAU, and MIMD architecture).

The diffuse inter-reflection ray performance can be limited by memory bandwidth. We assumed a 2-channel LPDDR3 that theoretically provides a peak bandwidth of 12.8 GB/s [JEDEC 2012a], but we can use about 3∼4 GB/s because the access pattern of the T&I unit is random [Rixner et al. 2000]. The diffuse rays showed lower cache hit rates than other types of rays. These cache misses increase the memory traffic (up to 3.7 GB/s), and the T&I utilization can be decreased due to the memory latency. If the traffic for shading (up to ∼2 GB/s) is also considered, this problem may be aggravated. We believe that this problem can be resolved with future mobile memory, such as LPDDR4 and Wide-IO [JEDEC 2012b].

Table 6 compares our SGRT system and the other approaches in different platforms, i.e., desktop GPU, MIC, and mobile processors. We rendered our test scene at full HD resolution (1920×1080). The area of the 4-SGRT cores includes all the T&I units and the SRPs. The test scene was Fairy.

First, for comparison with a state-of-the-art desktop GPU, we implemented a ray tracer with NVIDIA OptiX 2.6 [2013] and measured NVIDIA GeForce GTX 680. Although this OptiX version supports various BVH trees, we chose a standard SAH BVH for fair comparison. The result is 37 fps, which is similar to our results of 34 fps. We believe that the SGRT is more advantageous in term of the performance per area. It should be noted that we used a much older fab process (65 nm) than GTX 680 (28 nm). In addition, we compared our results with those of Wald's research study on Intel MIC architecture [Wald 2012]. We used the performance number of pure rendering provided in his paper, because his work focused on fast tree build. We believe that the comparison is fair because his tree is also based on the best quality full SAH. The SGRT also outperforms to the MIC (29 fps). Again, we find that our GPU already achieves the performance of ray tracers in the legacy desktop environment.

Second, we also compared our approach to the previous mobile processors for ray tracing [Spjut et al. 2012] [Kim et al. 2012a]. In [Kim et al. 2012a], the authors did not provide the values of rendering performance for Fairy scene, and therefore we computed the rendering performance with the throughput number (Mrays/s) they presented. These two processors occupied small areas (25 mm^2 at 65 nm, 16 mm^2 at 90 nm), which are similar to ours (25 mm^2 at 65 nm). However, the performance is much worse than that of our architecture (9 fps, 2 fps). The performance gap would be wider, because their resolution was 720p HD.

4.3 FPGA prototype

To achieve a more accurate functional verification and demonstration of our SGRT architecture, we also implemented our full GPU system at RTL level with Verilog HDL (Hardware Description Language). The implemented RTL codes are all executed on FPGA platforms, as shown in Figure 9. Our prototype used Xilinx Virtex-6 LX760 FPGA chips [2013], which are hosted by the Synopsys HAPS-64 board [2013b]. We customized 32-bit dual-channel DDR2 memory and dual-port SRAM to the target board. All the memory data (BVH tree, primitive scene, texture, SRP VLIW/CGRA codes) can be uploaded via a USB interface when the application is driven. The rendered scene is stored to a frame buffer and displayed on the in-house fabricated LCD display (800×480) board. A single SGRT core, including a T&I unit, SRP core, AXI-bus, and memory controller, is partitioned and mapped to two Virtex-6 chips due to the size limitation of an FPGA chip. Therefore, the redundant resources can exist in the two chips (e.g., AXI-buses and memory controller). This design uses 80%/60% logic slices, and 31%/41% block memories of two FPGA chips, respectively. We implemented a dedicated AXI-AHB bridge as an interface between the FPGA chips due to the limitation of the number of FPGA pins. The overall system was synthesized and implemented with Xilinx Synplify Premier and ISE PAR at a 45 MHz clock frequency. For fast prototyping, we used the floating-point unit of the Synopsys DesignWare Library [2013a].

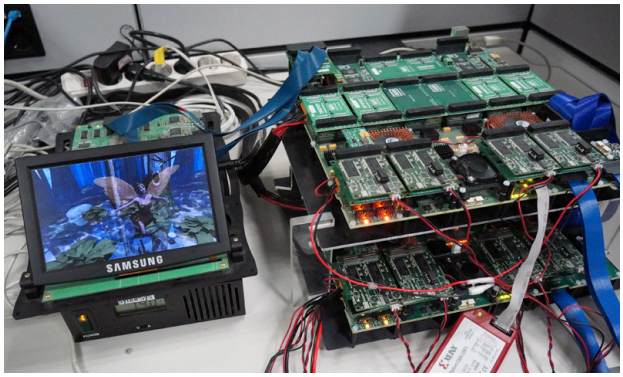

Figure 9: *Our FPGA prototype system.*

Several features of the SGRT could not be implemented due to the size limitation of the FPGA chip. The current RTL version of the T&I unit does not include RAU, and the only three TRVs are integrated in a single T&I unit. In addition, the SIMD RGS kernels could not be used in the FPGA prototype because SIMD intrinsics were not included in the current RTL version of the SRP.

Figure 1 shows the images (Ferrari and Fairy scene) rendered by the SGRT on our FPGA prototype system. Full ray tracing (primary and secondary ray (reflection/refraction/shadow)), including shading and bilinear-filtered texturing, is performed. To add the effect of reflection/refraction to the Fairy scene, we modified the material information of the objects and increased the number of light sources to two. The rendering performance on the FPGA was 2.1 fps and 1.3 fps for Fairy and Ferrari, respectively. As compared with the results presented in Table 6, these values are reasonable, if we consider the difference between the platform environments (FPGA vs. SoC, and core frequency) and non-implemented features in RTL codes. We believe that we will demonstrate an interactive frame rate through latency hiding by adding RAU and increasing the clock frequency in the near future.

5 Conclusion and Future Work

In this paper, we proposed a new mobile GPU architecture called SGRT. SGRT has a fast compact hardware engine that accelerates the T&I operation and employs a flexible programmable shader to support the RGS. SGRT was evaluated through a cycle-accurate simulation, and the validity of the architecture was verified by FPGA prototyping. The experimental results showed that SGRT can provide real-time ray tracing at full HD resolution and compete with not only mobile ray tracing processors but also desktop CPU/GPU ray tracers in terms of the performance per area. The full SGRT system was implemented at the Verilog/RTL level, and then demonstrated on an FPGA platform. This constitutes a significant attempt to apply ray tracing to the mobile terminal that is used worldwide. We believe that SRGT will contribute to the various future applications and enhance the user experience.

Our GPU has certain limitations. First, complex shading can be a major cost. To date, we have implemented a simple shading effect, and therefore the RGS has not constituted a bottleneck. However, more complex computation for various illuminations and material

shading will cause a bottleneck in the SRP. Currently, we are adding advanced features to the SRP to improve the abstract performance, and then we will provide support for more complex RGS kernels. Although the SGRT is currently using the SRP as a programmable logic device, the shader core of the GPUs already available in mobile SoCs can be an alternative solution. This will be a more cost-effective solution for hybrid rasterization and ray tracing.

Second, our SGRT supports only triangle primitives for simple hardware. However, we may need to employ a programmable solution to support various primitives and various intersection algorithms. The potential solution is to employ a programmable shader, as with RPU [Woop et al. 2005]. A small SRP core including only a VLIW engine may be a solution.

Third, our SGRT at RTL level has not been completely implemented due to the FPGA size limitation and for several other reasons. Therefore, we could not demonstrate the best performance shown in the cycle-accurate simulation. Furthermore, we cannot yet verify SGRT in the ASIC environment, which is close to that of the mobile SoC. We will synthesize the full system with an ASIC design compiler, and conduct a more detailed analysis of the cost and power consumption.

Acknowledgements

The authors thanks Timo Aila and Tero Karras for sharing their GPU ray tracers. Models used are courtesy of Dabrovic (Sibenik), Ingo Wald (Fairy Forest), Anat Grynberg and Greg Ward (Conference Room).

References

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs - kepler and fermi addendum. In *Proceedings of ACM High Performance Graphics 2012, Posters*.

ARM, 2013. ARM flagship mobile GPU, Mali-T678. http://www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute/mali-t678.php.

ARM, 2013. The ARM NEON general-purpose SIMD engine. http://www.arm.com/products/processors/technologies/neon.php.

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

BORKAR, S., AND CHIEN, A. A. 2011. The future of microprocessors. *Communications of the ACM 54*, 5 (May), 67–77.

CAUSTIC, 2013. Caustic series2 raytracing acceleration boards. https://caustic.com/series2.

CUDA, 2013. NVIDIA CUDA 5. http://www.nvidia.com/object/cuda_home_new.html.

ERNST, M. 2008. Multi bounding volume hierarchies. In *Proceedings of IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*, 35–40.

ERNST, M. 2012. Embree: Photo-realistic ray tracing kernels. In *ACM SIGGRAPH 2012,Exhibitor Technical Talk*.

EXYNOS, 2013. Samsung application processor. http://www.samsung.com/exynos.

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.

GOMA, S. R. 2011. A 3D camera solution for mobile platform. In *Workshop on 3D Imaging*.

GRIBBLE, C., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *Proceedings of IEEE/Eurographics Symposium on Interactive Ray Tracing*, 59–68.

HAMEED, R., QADEER, W., WACHS, M., AZIZI, O., SOLOMATNIKOV, A., LEE, B. C., STEPHENRICHARDSON, KOZYRAKIS, C., AND HOROWITZ, M. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer architecture (ISCA)*, 37–47.

HSA, 2013. Heterogeneous system architecture foundation. http://www.hsafoundation.com.

IPAD4, 2013. Inside the apple iPad4, A6X a very new beast! http://www.chipworks.com/blog/recentteardowns/2012/11/01/inside-the-apple-ipad-4-a6x-a-very-new-beast/.

JEDEC, 2012. Low power double data rate 3 SDRAM (LPDDR3). http://www.jedec.org/sites/default/files/docs/JESD209-3.pdf.

JEDEC, 2012. Wide I/O single data rate (Wide I/O SDR). http://www.jedec.org/sites/default/files/docs/JESD229.pdf.

KAN, P., AND JAUFMANN, H. 2012. High-quality reflection, refraction, and caustics in augmented reality and their contribution to visual coherence. In *Proceedings of International Symposium on Mixed and Augmented Reality (ISMAR)*, 99–108.

KIM, H.-Y., KIM, Y.-J., OH, J., AND KIM, L.-S. 2012. A reconfigurable SIMT processor for mobile ray tracing with contention reduction in shared memory. *IEEE Transactions on Circuits and Systems 1*, 99, 1–13.

KIM, J.-W., LEE, W.-J., LEE, M.-W., AND HAN, T.-D. 2012. Parallel-pipeline-based traversal unit for hardware-accelerated ray tracing. In *Proceedings of ACM SIGGRAPH Asia 2012, Posters*.

KISHONTI, 2013. CLbenchmark 1.1. http://clbenchmark.com.

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

LEE, W.-J., WOO, S.-O., KWON, K.-T., SON, S.-J., MIN, K.-J., LEE, C.-H., JANG, K.-J., PARK, C.-M., JUNG, S.-Y., , AND LEE, S.-H. 2011. A scalable GPU architecture based on dynamically embedded reconfigurable processor. In *Proceedings of ACM High Performance Graphics 2011, Posters*.

LEE, W.-J., LEE, S., NAH, J.-H., KIM, J.-W., SHIN, Y., LEE, J., AND JUNG, S. 2012. SGRT: A scalable mobile GPU architecture based on ray tracing. In *Proceedings of ACM SIGGRAPH 2012, Talks*.

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

MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo

scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE) 2003*, 10296.

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

NAH, J.-H., KANG, Y.-S., LEE, K.-J., LEE, S.-J., HAN, T.-D., AND YANG, S.-B. 2010. MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices. In *Proceedings of ACM SIGGRAPH ASIA 2010 Sketches*.

NAH, J.-H., PARK, J.-S., PARK, C., KIM, J.-W., JUNG, Y.-H., PARK, W.-C., AND HAN, T.-D. 2011. T&I Engine: traversal and intersection engine for hardware accelerated ray tracing. *ACM Transactions on Graphics 30*, 6 (Dec).

OPENCL, 2013. Khronos OpenCL. http://www.khronos.org/opencl/.

OPTIX, 2013. NVIDIA OptiX. http://www.nvidia.com/object/optix.html.

PARK, W.-C., KIM, D.-S., PARK, J.-S., KIM, S.-D., KIM, H.-S., AND HAN, T.-D. 2011. The design of a texture mapping unit with effective mip-map level selection for real-time ray tracing. *IEICE Electron. Express 8*, 13 (July), 1064–1070.

PEDDIE, J. 2011. OpenGL ES and mobile trends - the next-generation processing units. In *ACM SIGGRAPH 2011 Khronos OpenGL ES and Mobile BOF Meeting*.

RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. 2000. Memory access scheduling. In *Proceedings of the 27th annual 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 ACM SIGGRAPH/EUROGRAPHICS Graphics Hardware*, 95–106.

SLUSALLEK, P. 2006. Hardware architectures for ray tracing. In *ACM SIGGRAPH 2006 Course Notes*.

SNAPDRAGON, 2013. Qualcomm application processor. http://www.qualcomm.com/snapdragon.

SONG, J. H., LEE, W. C., KIM, D. H., KIM, D.-H., AND LEE, S. 2012. Low-power video decoding system using a reconfigurable processor. In *Proceedings of IEEE International Conference on Consumer Electronics (ICCE) 2012*, 532–533.

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.

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

SYNOPSYS, 2013. Designware library. http://www.synopsys.com/IP/SoCInfrastructureIP/DesignWare/Pages/default.aspx.

SYNOPSYS, 2013. HAPS-60 series of FPGA systems. http://www.synopsys.com/Systems/FPGABasedPrototyping/Pages/HAPS-60-series.aspx.

TEGRA, 2013. NVIDIA application processor. http://www.nvidia.com/object/tegra-4-processor.html.

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

WALD, I., IZE, T., AND PARKER, S. 2008. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics 32*, 1, 3–13.

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

WALD, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, 33–40.

WALD, I. 2012. Fast construction of SAH BVHs on the intel many integrated core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1, 47–57.

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

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

XILINX, 2013. Virtex-6 FPGA family. http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/index.htm.