A Mobile Accelerator Architecture for Ray Tracing

Josef Spjut, Daniel Kopta, Erik Brunvand, and Al Davis

Abstract—Mobile computing in the form of smart phones and tablets is becoming ubiquitous. As these devices are being used for increasingly sophisticated tasks, the graphics requirements are also increasing. With the growing desire for highly realistic graphics, the use of ray tracing for rendering will become essential. Ray tracing efficiently models complex illumination effects to improve visual realism in a very different way than current graphics accelerators, which use rasterization on SIMD hardware. Ray tracing also has some intriguing advantages in the mobile computing space where screen pixel counts are not likely to grow significantly, but scene size and complexity will continue to grow. We present a novel multi-core MIMD graphics accelerator architecture that is well suited to ray tracing on mobile platforms. Our architecture provides a large number of floating point resources and exploits thread-level parallelism to keep those units active during ray tracing. We show that a small-footprint version of this architecture is suitable for the mobile computing space, and has performance up to 13 times faster than an existing mobile graphics solution for ray tracing.

Index Terms—Ray Tracing, Graphics Accelerator, Computer Architecture

1 Introduction

Recent years have seen a huge increase in the computational power and popularity of mobile devices. Smart phones are dominating the cell phone market and low-power tablet computing devices are becoming increasingly popular. Some of the advantages over more traditional computing platforms are that these devices are always available, are usually connected to the network, and have support for advanced graphics. Graphics support is important not only for graphics-intensive user interfaces, but increasingly because the applications themselves require high-quality graphics. Mobile computing applications are being deployed in situations ranging from medical, to scientific applications where visualizing data quickly and accurately is essential.

Interactive computer graphics architectures are currently dominated by single instruction, multiple data (SIMD) hardware accelerators executing some variant of triangle rasterization [1]. Sometimes this execution model is called SIMT for single instruction, multiple threads. The terms SIMD and SIMT will be used interchangeably in this paper. These highly specialized processors stream the image primitives through parallel SIMD pipelines to increase performance, a process made possible because the scene primitives can be projected to the screen's coordinate space independently. However, this becomes a bottleneck for highly realistic rendering because it limits shading to per-primitive computations and does not allow for efficient computation of global illumination effects without first repeating the transformation operations for each possible light direction.

• *The authors are with the School of Computing at the University of Utah, Salt Lake City, UT, 84112.*
E-mail: sjosef,dkopta,elb,ald AT cs.utah.edu

Ray tracing is the main alternative rendering algorithm to z-buffer based rasterization [2], [3]. The principle is simple: at each pixel a ray is cast from the viewer's eye through the pixel into the virtual scene. That operation returns information about the closest primitive seen by that ray. The pixel is then shaded (colored) based on the material properties of that primitive. All high-performance ray tracers use a hierarchical scene partitioning structure, known generically as an acceleration structure, to prune the ray intersection test [4]. This results in both divergent program execution due to branching as the ray descends through the acceleration structure, and non-coherent memory accesses to the scene database. Neither of these behaviors are efficiently supported in a parallel SIMD architecture [5], [6], [7].

From each point where the ray intersects an object in the scene, an additional "secondary" ray can be recursively cast into the scene to determine optical effects such as shadows, reflections, refraction, caustics (focused light from an indirect source), and other global illumination and optical effects. Ray tracing has distinct advantages over rasterization in terms of its ability to easily render these optical effects, making ray tracing the rendering algorithm of choice for highly realistic images. Ray tracing can also be effectively used for traversing volumetric data and large data sets such as medical images and other scientific data.

Another potential advantage of ray tracing for mobile platforms is that first-order performance scales linearly with the number of screen pixels. The inner loop of a ray tracer iterates over the pixels, which can each be processed independently. The hierarchical acceleration structure allows the search of the scene data to behave roughly logarithmically in the number of primitives, whereas first-order rasterization performance scales linearly with the number of scene primitives. Some culling

based on scene partitioning is possible, but in general rasterization time grows with the number of geometric primitives.

For a mobile device, the number of pixels is not expected to grow dramatically. An iPhone4 Retina display (640x960) is reported to be roughly at the resolution of the human eye already [8]. A tablet such as an iPad (1024x768) [9] or Samsung Galaxy (1280x800) [10] has somewhat higher pixel count because of larger screen size. Scene data can be expected to increase in size and complexity as new applications are explored [11]. Mobile ray tracers will be able to handle larger scenes as the memory capacity of mobile devices increases.

In order for acceleration structures to effectively cull geometry to prevent unnecessary computation, the branching pattern for ray tracing is unpredictable. The majority of existing graphics accelerator architectures depend on keeping clusters of threads coherent in order to achieve high utilization of the SIMD functional units. Since it is hard to predict the branching that will occur among a group of threads, we have designed our architecture to permit threads to diverge gracefully. Allowing threads to execute independently has previously been shown to be useful for accelerating ray tracing [12], [13].

We start with a number of multiple-instruction multiple-data (MIMD) thread processors (TPs) which are very simple in-order integer processors. Each TP has a small register bank and a local memory which is used for thread-local stack operations. Clusters of TPs share multiply and add floating point units (FPUs), which are more area and energy expensive than integer units. We adjust the size of the cluster to provide high utilization of the FPUs. TPs and their shared FPUs are grouped into what we call a thread mutliprocessor (TM). TPs in a TM share a floating point divide and inverse square root unit, which is even more area and energy expensive, yet rarely utilized. Each TM shares a banked instruction cache to allow TPs that do not conflict for a particular bank to proceed in parallel. Initially there is a relatively high bank conflict rate which causes software threads on the TPs to experience delays. However, we find that natural "staggering" of the threads through divergent execution quickly yields higher levels of parallelism. Multiple TM tiles share a banked data cache which contains the global, shared scene data and frame buffer. An example TM with 32 TPs s can be seen in Figure 1. We present the experimental group and cluster sizes in detail in section 3.1.

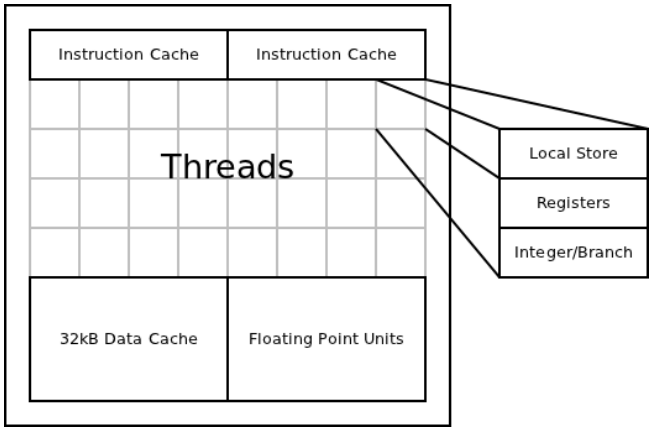

Fig. 1: A 32-thread TM with shared caches and FPUs

2 Related Work

While mobile ray tracing is a relatively new concept, mobile graphics accelerators have been in use for some time and have been growing in popularity. Additionally, ray tracing has been implemented on more traditional, high power GPUs with some success.

2.1 Commercial

Tegra [14] is a commercial System on Chip (SoC) design from NVIDIA targeting mobile computing devices such as cell phones, media players and tablets. An important part of the SoC is the inclusion of a graphics accelerator intended for rasterization. While rasterization and ray tracing share some of the same shading requirements, ray tracing more naturally handles hidden surface removal, indirect lighting, and shadow effects. Ray tracing has been performed on NVIDIA's discrete GPU solutions, however current Tegra chips do not have the same unified compute architecture yet and would likely perform ray tracing poorly. A highly optimized ray tracer written for a GTX 280 only achieves between 61% and 86% SIMD efficiency [7]. Note that this SIMD efficiency does not represent the floating point efficiency directly as many of the executed instructions are integer or control instructions. A comparison of accelerator compute capabilities, including the graphics accelerator from Tegra 2, can be found in Table 1.

PowerVR [15] is an architecture that does very similar computations to those done by Tegra chips. The main distinction of the PowerVR parts is that they separate the image into a set of screen tiles which can be independently processed. Triangles that overlap each screen tile are placed into corresponding geometry bins prior to hidden surface removal. Visibility is then determined by performing a simple ray cast for each pixel and each primitive in the tile. The professed benefit of the tile-based approach is that with accurate depth information, the renderer can avoid processing fragments for many of the hidden surfaces that would not contribute to the final display color. Ray tracing similarly removes hidden surfaces prior to processing fragments, but is capable of retaining access to global scene data. Similar to the Tegra, PowerVR chips are designed for rasterization and can perform ray tracing with some difficulty, despite the use of ray casting for hidden surface removal, because global scene data is not retained.

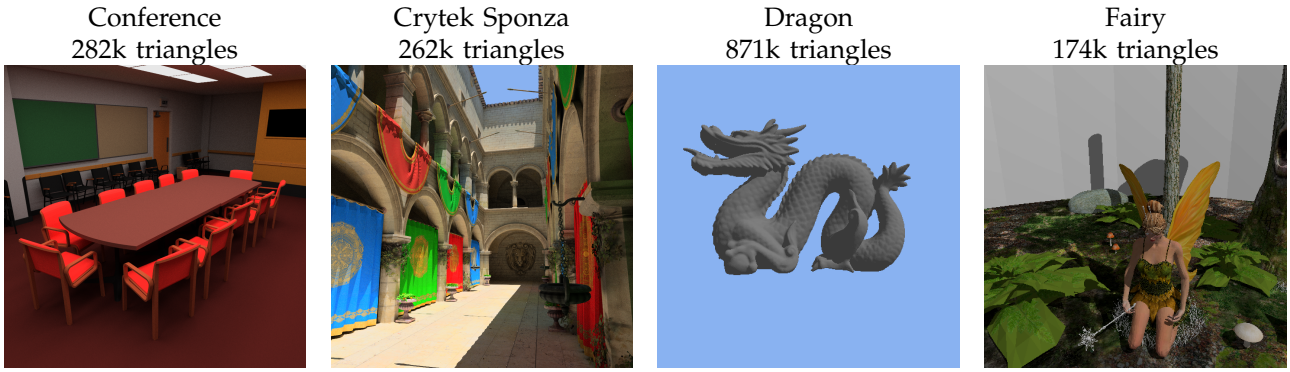

Fig. 2: Test scenes used to evaluate performance

2.2 Research

Lohrmann [16] presents a method for performing ray tracing on more traditional GPU-style architectures. Ray tracing is expressed as vertex and fragment shader programs that execute within the traditional rasterization pipeline and operate on scene data stored within the texture and buffer memories of the GPU. While Lohrmann's approach is a useful way to repurpose existing hardware, our architecture is designed to have the exact hardware resources needed for ray tracing. In addition, Chang et. al. [17] find that bounding volume hierarchies (BVH)s are the most energy efficient acceleration structure on both CPUs and GPUs. We similarly use a BVH for our acceleration structure to reduce power consumption.

Kim et al. [18], [19] demonstrate their Mobile Ray Tracing Processor (MRTP), which is similar to most SIMD targeted ray tracers in that they experience the difficulty of dealing with the SIMT execution model. Their approach is to allow the architecture to dynamically reconfigure a hybrid vector SIMD configuration with fewer dependent threads of execution. However, to ensure high vector utilization, the SIMD threads must be able to find opportunities to issue 3-wide vector operations. While this dynamic reconfigurability is interesting, we employ a MIMD design to allow for more thread flexibility. The MRTP achieves a peak performance of 673K rays/sec using 16 mm^2 in a 0.13 μm process running at 100 MHz on a small scene. The MRTP only executes 103K rays/sec for the much larger dragon model, which is representative of the size of modern scenes. Their work is the best point of comparison for mobile ray tracing accelerators, hence Table 2 provides comparison with their best case performance, naively area and frequency scaled to 65nm and 500 MHz. Anido et al. [20] also synthesize an architecture for interactive ray tracing in a 0.13 μm process that only consumes 0.125 mm^2. However, their work tests only very simple scenes and does not use an acceleration structure, making it a poor point for comparison to the work presented here.

While our accelerator architecture targets the mobile SoC space, it has a lot in common with more general and higher power architectures. TRaX [12], [13], Rigel [21], Copernicus [22], etc. all take a basic block of threads, similar to the one presented here, and tile them on a chip. The main difference is that these tiled architectures target a die area of around 200 mm^2 and a relatively high power budget, while this work targets a die area of about 10 mm^2 with much less power consumption.

3 Architecture and Methodology

Our TP architecture is based on a simple, in-order integer thread execution model. Several of these thread TPs are then grouped together in a TM to share a number of floating point execution units and banked instruction and data caches. The TM tile can be replicated to increase the total compute power. Since the floating point units are shared within the architecture, we strive to find a design point that is capable of achieving high utilization of these shared units. To a great extent the floating point utilization depends on the particular application executing on the system. In this work, we consider a ray tracer that traces primary visibility as well as shadow rays.

To program our accelerator, we write a basic ray tracer in C/C++ and generate LLVM [23] intermediate code. We use a customized LLVM back end to emit code compatible with our simple, RISC-like ISA. In order to execute architecture specific instructions, we expose a few simple compiler intrinsics to the programmer. The single executable is then run on each thread independently. The primary form of communication among threads is a simple atomic increment instruction that each thread uses to find a unique assignment. Global memory operations are managed by the programmer and the acceleration structure is built by the host CPU and made available in the accelerator's memory space.

We simulate our architecture using a custom cycle-accurate system simulator in which we can instantiate a number of TPs and TMs as well as the associated functional units, including caches and local memories. Conflicts for shared functional units are resolved in a round robin fashion. This prevents starvation of any individual thread. Via cycle-accurate simulation, we can

accurately track functional unit utilization, memory usage and other statistics related to the execution of our accelerator.

3.1 Architectural Details

Each TP consists of a simple, in-order, single-issue integer processor with 32 general purpose registers and a small 512 byte local memory. The local memory acts as an extended register file for local stack operations. We do not employ branch prediction and rely instead on thread parallelism to achieve higher performance and to keep the shared floating point units busy. In every configuration the FPU is shared by 8 TPs. We find empirically that this is sufficient since each TP spends execution on pointer chasing and waiting for memory requests to return which keeps it from issuing FPU instructions on every cycle.

In addition to the FPUs that are shared by each TP cluster, we also have one special purpose floating point divide and inverse square root unit. Since this special purpose functional unit is rarely used, we use only one of them per TM and for TMs comprising up to 64 TPs. It should also be noted that the FPDIV/iSQRT functional unit has a latency of 8 cycles at 500 MHz. This is higher than any of the other functional units in the accelerator, all of which have single cycle execution.

Each TM has a 4 kB, 16-bank instruction cache for every 16 TPs allowing threads to issue in parallel as long as they are fetching instructions from independent banks. In practice our in-order threads have enough execution divergence that sharing this instruction cache does not have a large negative impact on performance. Sharing the cache banks and floating point units largely mitigates the die area overhead that a MIMD architecture would normally have over a SIMD approach.

For each TM, we use a 16kB banked data cache that caches data from the global shared memory. We find that one bank per 8 TP cluster is the appropriate choice. The global memory segment includes all of the scene data, acceleration structure, and frame buffer. Because the thread assignment gives one pixel at a time to each thread, we force all frame buffer writes to go around the data cache thereby preventing pollution of the cache by lines which are write only.

We limit the off-chip bandwidth to 8 GB/s based on the fact that upcoming mobile SoCs, such as the Samsung Exynos 5250 [24], achieve up to 12.8 GB/s of memory bandwidth with a 64 bit memory bus. We believe 8 GB/s is a reasonable assumption for a compute-bound GPU in the near future because SoCs also share that memory bandwidth with other IP blocks. We note that if the GPU and host CPU are both in memory bound computational segments, the shared bandwidth will impose performance restrictions. Section 4.1 considers a future SoC with more available memory bandwidth.

For area and performance estimates, we use Synopsys DesignWare/Design Compiler [25] and a commercial 65nm CMOS cell library to synthesize functional units, and Cacti 6.5 [26] for our cache and memory analysis.

TABLE 1: Comparison of mobile graphics accelerator architectures. All accelerators are scaled to 65nm and 500 MHz naively for better comparison with our configurations. *Tegra 2 die size is estimated from a die photo.

Architecture	Size(mm^2)	GFLOPS	RT GFLOPS
PowerVR SGX543MP1	8.0	18.0	
PowerVR SGX543MP2	16.0	36.0	
NVIDIA Tegra 2	6*	8.0	
MRTP [18] (130nm)	16.0	4.3	≈1.2
MRTP (naively scaled)	4.0	21.5	≈6.0
32 (1x32)	1.9	4.0	2.5
48 (1x48)	2.5	6.0	3.7
64 (2x32)	3.8	8.0	4.9
64 (1x64)	3.2	8.0	4.9
96 (2x48)	5.1	12.0	7.2
128 (4x32)	7.6	16.0	9.3
128 (2x64)	6.3	16.0	9.2
192 (6x32)	11.4	24.0	12.6
192 (4x48)	10.1	24.0	12.7
256 (8x32)	15.2	32.0	15.5
256 (4x64)	12.6	32.0	15.7
288 (6x48)	15.2	36.0	17.1
384 (8x48)	20.2	48.0	20.3
384 (6x64)	18.9	48.0	20.3
512 (8x64)	25.3	64.0	23.1

Although we do not have accurate power consumption data for this architecture, we can make a rough estimate based on estimated energy from Cacti and the activity factor reported by our simulator. Our 4 TM $\times$ 32-thread chip uses an average of 4 Watts rendering our test scenes. It should be noted that the caches and memories generated by Cacti are not optimized for low power, and it is likely that power consumption can be greatly reduced for more custom designed devices.

4 Results

We simulated the execution times for a number of configurations of the proposed architecture on a simple ray tracer application to gather performance and utilization data. We consider TM configurations with 32, 48 and 64 TPs per TM and for 1, 2, 4, 6 or 8 TM tiles. Results in all tables are ordered by the number of total threads across all TMs and are annotated by the number of TMs and the number of TPs per TM in parentheses. The test scenes in Figure 2 were run on each configuration and the results presented are an average across the benchmark scenes unless indicated otherwise. Note that while some of the images shown have textures, the ray tracer used to report results does not perform texturing. Each scene was rendered at a resolution of 1280x720 with primary rays and shadow rays for a single light source. For every eight threads in a TM we provide one floating point multiplier and one floating point adder while the entire TM shares one special functional unit regardless of the number of threads. Thus a 32-thread TM has a maximum 9 FLOP per cycle capability while the 48 and 64 TP TMs have 13 and 17 FLOPs respectively.

TABLE 2: Ray tracing performance, shown in millions of rays per second.

Threads	conference	crytek	dragon	fairy	Average
32 (1x32)	2.48	1.41	1.94	1.81	1.91
48 (1x48)	3.74	2.11	2.81	2.72	2.84
64 (2x32)	4.94	2.80	3.62	3.59	3.74
64 (1x64)	4.96	2.78	3.60	3.60	3.74
96 (2x48)	7.43	4.19	5.17	5.37	5.54
128 (4x32)	9.80	5.55	6.18	7.03	7.14
128 (2x64)	9.86	5.52	6.09	7.08	7.14
192 (6x32)	14.5	8.24	5.88	10.2	9.72
192 (4x48)	14.7	8.26	6.07	10.3	9.84
256 (8x32)	19.1	10.8	5.75	12.3	12.0
256 (4x64)	19.3	10.8	5.90	12.6	12.2
288 (6x48)	21.5	12.2	5.91	13.4	13.2
384 (8x48)	27.0	15.5	5.74	14.7	15.7
384 (6x64)	27.2	15.5	5.86	14.9	15.9
512 (8x64)	32.5	18.2	5.68	15.8	18.1

A comparison of floating point capabilities of our architecture and commercial rasterization architectures can be found in Table 1. The "RT GFLOPS" column is the simulated floating point performance when running our ray tracer and is not reported for the commercial architectures because ray tracers are not readily available for comparison on those architectures. The "RT GFLOPS" entry for MRTP [18], [19] is approximated based on the thread issue data provided in their papers. Only multiplies and adds are considered in the floating point compute capabilities of the various architectures, and do not include the rarely used FPDIV/iSQRT special function unit.

Table 2 gives a comparison of the ray processing capabilities of the various configurations that were simulated. As the number of threads increases, so does the raw performance of the configuration. In the case of the dragon scene, the memory access pattern is such that even with only 128 threads, the computation is memory bandwidth limited, preventing further increases in ray tracing performance. Section 4.1 goes into more depth on the bandwidth concern. In order to provide a reasonable comparison to the MRTP, we consider the only scene we share in common with them, viz. the dragon. We choose a 128-thread configuration because the area is similar to what the MRTP would use when scaled to a 65nm process. We also scale their performance up to 500 MHz assuming the change to the 65nm process would allow for a faster clock rate, although a 5x increase is likely optimistic. Our 128 thread configuration is able to perform 6.18 million rays per second while the MRTP achieves only 0.515 million rays per second, giving our architecture a 13x speedup for the same circuit area.

The million rays per second (MRPS) metric is a standard measure of performance in ray tracing systems. It is preferred due to the separation from any details of the image being rendered, such as resolution and rays per pixel, which can vary widely depending on which shading techniques are used. As rays per second increases, either a higher quality image can be rendered in the same amount of time, or the same image can be rendered faster. For an HD resolution of 1280x720 pixels, we can ray trace images with full shadows at 3.4 frames per second. While this is not a real-time frame rate, it is still interactive enough for most medical imaging and visualization applications.

TABLE 3: Performance in millions of rays per second with the baseline and increased memory bandwidth for the dragon scene as well as an average across all scenes tested.

Architecture	8GB/s dragon	16GB/s dragon	8GB/s Average	16GB/s Average
256 (8x32)	5.75	10.17	12.0	12.7
384 (8x48)	5.75	10.16	15.8	16.3
512 (8x64)	5.69	10.14	18.1	18.5

4.1 Memory Bandwidth Concerns

Our architecture performs ray tracing well and is capable of utilizing the available floating point units effectively until the memory bandwidth limit is reached. In particular, the performance of the dragon scene stops scaling because it reaches the bandwidth limit with only 128 total threads for any TM count. However, the bandwidth available to mobile SoCs is likely to grow in the future due to increasing memory clock rates as well as larger memory buses. Table 3 shows the increases in performance that can be achieved when bandwidth is raised to 16GB/s. The dragon scene achieves almost a 2x performance increase since it is primarily memory bandwidth constrained. It is likely that increasing the size of the cache would also decrease the pressure on the memory bus.

5 Conclusions

Pure SIMD is not the most efficient ray tracing architecture due to the divergent execution and memory patterns induced by traversing the acceleration structure and the intrinsic nature of secondary rays [7]. The MRTP architecture [18] addresses this limitation by allowing their architecture to dynamically reconfigure to accommodate smaller SIMT blocks. The MRTP relies on single-thread vector operations to maintain performance while avoiding the extra overhead of moving to a full MIMD architecture. Our alternative approach embraces this divergent behavior and allows threads to execute in MIMD fashion and recovers efficiency through resource sharing. Instead of giving each thread its own floating point multiplier and adder, we decouple those units, sharing them among a group of threads. This type of sharing is not possible in a typical SIMD architecture. Rarely will all threads need the same unit at the same time. Furthermore, we share banked instruction and data caches to enable parallel access when threads are not strictly synchronized. The normal MIMD overhead is greatly reduced, and we are able to find a 13x speedup over the reconfigurable SIMT architecture.

While it may seem counter-intuitive that MIMD can work efficiently, most other MIMD systems do not share expensive computational units. In addition, the programming effort required to exploit parallelism in a SIMT system is quite high. Parallelism can be further increased by allowing the threads to occasionally execute vector instructions, increasing the burden on the programmer. While automated tools can be developed to ease the required programming effort, these tools are not yet widely available. In contrast, the programming effort required to write a ray tracer for a MIMD architecture, such as the one proposed in this work, is minimal. The programmer can depend on the hardware to exploit parallelism through the proper use of shared resources.

As process scaling continues to provide more compute capabilities per square millimeter and watt, we expect designs similar to the one presented in this paper to become a viable alternative. While we have only shown an architecture capable of interactive frame rates at HD resolutions, a few process generations should enable real-time frame rates while supporting more realistic lighting effects.

ACKNOWLEDGMENTS

The Crytek Sponza scene is available from Crytek at http://www.crytek.com/cryengine/cryengine3/downloads
This research was supported in part by NSF grant CNS10174757.

REFERENCES

[1] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," Ph.D. dissertation, University of Utah, December 1974.
[2] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.
[3] A. Glassner, Ed., *An introduction to ray tracing*. London: Academic Press, 1989.
[4] P. Shirley and R. K. Morley, *Realistic Ray Tracing*. Natick, MA: A. K. Peters, 2003.
[5] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Transactions on Graphics (SIGGRAPH '05)*, vol. 24, no. 3, pp. 1176–1185, July 2005.
[6] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *SIGGRAPH '06*. New York, NY, USA: ACM, 2006, pp. 485–493.
[7] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *HPG 2009*, 2009, pp. 145–149.
[8] C. Brandrick, "iPhone 4's retina display explained," June 2010, PC World, http://www.pcworld.com/article/198201/iphone_4s_retina_display_explained.html.
[9] Apple Computer, "iPad technical specifications," http://www.apple.com/ipad/specs/.
[10] Samsung, "Galaxy tablet technical specifications," http://www.samsung.com/global/microsite/galaxytab/10.1/spec.html.
[11] D. P. Greenberg, K. E. Torrance, P. Shirley, J. Arvo, E. Lafortune, J. A. Ferwerda, B. Walter, B. Trumbore, S. Pattanaik, and S.-C. Foo, "A framework for realistic image synthesis," in *Proceedings of SIGGRAPH*, 1997, pp. 477–494.
[12] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A multicore hardware architecture for real-time ray tracing," *IEEE Transactions on Computer-Aided Design*, 2009.
[13] D. Kopta, J. Spjut, E. Brunvand, and A. Davis, "Efficient mimd architectures for high-performance ray tracing," in *Computer Design (ICCD), 2010 IEEE International Conference on*, Oct. 2010.
[14] NVIDIA, "Bringing High-End Graphics to Handheld Devices," NVIDIA Corporation, Tech. Rep., 2011.
[15] POWERVR, "POWERVR MBX Technology Overview," Imagination Technologies Ltd., Tech. Rep., May 2009.
[16] P. J. Lohrmann, "Energy-Efficient Interactive Ray Tracing of Static Scenes on Programmable Mobile GPUs," Ph.D. dissertation, WORCESTER POLYTECHNIC INSTITUTE, February 2007.
[17] C.-H. Chang, P. J. Lohrmann, E. O. Agu, and R. W. Lindeman, "ENCORE: Energy-Conscious Rendering for Mobile Device," in *GPGPU*, October 2007.
[18] H.-Y. Kim, Y.-J. Kim, and L.-S. Kim, "MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 2, 2012.
[19] ——, "Reconfigurable mobile stream processor for ray tracing," in *Custom Integrated Circuits Conference (CICC), 2010 IEEE*, September 2010.
[20] M. Anido, N. Tabrizi, H. Du, M. Sanchez-Elez M, and N. Bagherzadeh, "Interactive ray tracing using a simd reconfigurable architecture," in *Computer Architecture and High Performance Computing, 2002. Proceedings. 14th Symposium on*, 2002, pp. 20 – 28.
[21] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09, 2009.
[22] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a multicore architecture for real-time ray-tracing," in *IEEE/ACM International Conference on Microarchitecture*, October 2008.
[23] Chris Lattner and Vikram Adve, "The LLVM Instruction Set and Compilation Strategy," CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS-R-2002-2292, Aug 2002.
[24] S. Bhagwat, "Samsung exynos 5250 begins sampling - mass production in q2 2012," http://www.anandtech.com/show/5467/samsun-exynos-5250-begins-sampling-mass-production-in-q2-2012.
[25] "Synopsys inc." http://www.synopsys.com.
[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO '07*, 2007, pp. 3–14.