A Feasibility Study of Ray Tracing on Mobile GPUs

Yunbo Wang*
Institute of Microelectronics
Tsinghua University

Chunfeng Liu
Institute of Microelectronics
Tsinghua University

Yangdong Deng
School of Software
Tsinghua University

Abstract

Ray tracing is considered to be a promising technology for enhancing visual experience of future graphics applications. This work investigates the feasibility of ray tracing on mobile GPUs. A ray tracer was developed by integrating state-of-the-art construction and traversal algorithms and implemented in both CUDA and OpenCL. We then performed a detailed characterization of the ray tracing workload in terms of runtime, memory usage, and power consumption on both NVIDIA Tegra K1 and PowerVR SGX 544-MP3 GPUs. The results are compared against mobile CPU and desktop GPU implementations. It is proved that the Tegra K1 GPU already allows constructing the acceleration structure of 1M-triangle scene in around 120ms and performing traversal at a throughput of 15 to 70 million rays per second.

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

Keywords: ray tracing, mobile GPUs, performance, memory bandwidth

1 Introduction

Mobile platforms exemplified by cell phones and tablets have been playing the most active role in today's information technology industry. On such mobile devices, computer graphics are still the most fundamental building blocks of human-computer interface. It is thus essential to be able to continuously improve the visual user experiences of mobile applications. In fact, the pursuit for better visual impacts has been the main driver leading to the exponential increase of modern GPU industry. In the past, a major way to achieve better user experiences is to increase the display resolution. For instance, Apple's iPad 3 exploits the Retina technology to offer a resolution of 2048x1536 on a 9.7-inch screen. It was claimed that at such a resolution a user would not be able to tell the difference of two adjacent pixels from a normal watching distance. However, this fact also suggests that we need new means to deliver even more entertaining visual effects of mobile devices.

At the present time, ray tracing has been considered as the most promising next-generation rendering technology [Keller et al. 2013]. However, ray tracing has by far typically been deployed on the desktop computers and even supercomputer, as it is considered as too expensive in terms of runtime. Accordingly, a wide spectrum of custom hardware solutions has been proposed to accelerate ray tracing [Schmittler et al. 2004; Spjut et al. 2009; Lee et al. 2013; Nah and Manocha 2014; Nah et al. 2011]. One such hardware accelerator was recently deployed in a PowerVR series 6 GPU [Imagination 2014a].

Though enabling a higher computing throughput at a given silicon estate, custom hardware still suffers from a reduced level of programming flexibility, which is often essential for complex graphics applications. In addition, existing ray tracing software have to be re-written to be executed on custom hardware. On the other hand, now it is close to performing real time ray tracing on the latest GPUs such as GTX780, thanks to the advancement of both GPU hardware, programing technology and algorithms. At the meantime, mobile GPUs are offering fast-growing computing capability. For instance, PowerVR SGX544 GPU [Imagination 2010], which is used in iPad Air and iPad 5s, has 4 multiprocessors with each having 4 Arithmetic Logic Units (ALUs) that run in parallel. Tegra K1 [NVIDIA 2014b], the latest NVIDIA mobile GPU, has as many as 192 CUDA cores, i.e., one quarter of CUDA cores installed on NVIDIA GTX 650Ti. As a matter of fact, mobile GPUs are designed under a tighter budget of area and memory bandwidth. An interesting problem, therefore, has to be raised: how big a performance gap has to be narrowed before real-time ray tracing is feasible on mobile GPUs? It is also appealing to study the unique trade-off of designing efficient ray tracing applications on mobile GPU architectures. Moreover, another essential problem is to identify the bottleneck of mobile GPUs for ray tracing.

As the first step toward addressing the above problems in a systematic manner, in this paper we investigate the feasibility of ray tracing on mobile GPUs. Efficient ray tracing solutions are developed for two different mobile GPUs, NVIDIA Tegra K1 and PowerVR SGX 544-MP3. Using CUDA [NVIDIA 2014a] and OpenCL [Khronos 2014] languages, our ray tracing implementation consists of both construction and traversal processes of the acceleration structure. We then quantitatively analyze the ray tracing performance in terms of construction speed, traversal and intersection throughput, memory bandwidth, and power consumption.

2 Background

To investigate the feasibility of mobile ray tracing, we developed ray tracers on two different GPUs, NVIDIA Tegra K1 GPU and PowerVR SGX 544MP3. We did not use PowerVR series 6 GPU with custom ray tracing hardware because the focus of this work is software-based ray tracing.

2.1 Tegra K1

As NVIDIA's latest mobile processor, Tegra K1 uses the same Kepler architecture as its desktop equivalent. With a single Streaming Multiprocessor eXtended (SMX), i.e. the core processor of Kepler GPU, Tegra is able to deliver a floating point computing throughput of 384 GFLOPS of 32-bit computations. An SMX features 192 CUDA processors with each consisting of a fully pipelined floating point unit and an integer unit. SMX follows a Single Instruction Multiple Threads (SIMT) execution model. These cores and other computing resources are organized into multiple execution clusters. In a single cycle, the units in one cluster always execute the same instruction from a group of 32 threads, so-called a warp. Different clusters are scheduled by 4 warp schedulers. Tegra is equipped

*e-mail:yunbowang1989@163.com

SIGGRAPH Asia 2014, December 03 – 06, 2014, Shenzhen, China.
2014 Copyright held by the Owner/Author.
ACM 978-1-4503-1891-4/14/12
http://dx.doi.org/10.1145/2669062.2669071

with a 128KB unified L2 cache and a 64-bit memory bus to a 2GB LPDDR3 memory. Tegra GPU is NVIDIA first GPU supporting both graphics (e.g., OpenGL) and general purpose programming (e.g., CUDA).

2.2 PowerVR SGX GPU

Imagination's PowerVR GPUs are widely used in various mobile devices. It has a fewer number of cores than Tegra and supports OpenCL for general purpose computing. The SGX 544MP3 GPU has 3 Universal Scalable Shader Engines (USSEs) as the main computing chassis and sustains a peak throughput of 51.1 GFLOPS. A USSE is the counterpart of SMX in NVIDIA GPUs. It has 16 residency slots that can be occupied by work-items (i.e., threads in CUDA) among which 4 work-items can be executed in parallel. Accordingly, the SGX 544MP3 GPU supports up to 12 active work-items at a given time.

3 Algorithm and Implementation

In this work, we developed ray tracers by integrating state-of-the-art algorithmic modules. We intentionally adopted mature algorithms so as to assess the feasibility of "modern" ray tracing techniques on mobile GPUs. Typically, a ray tracer consists of three steps. First, the graphics primitives (usually triangles) in a 3-D scene are organized into a spatial acceleration structure like bounding volume hierarchy (BVH) and KD-tree. Second, a large number of rays are emitted to traverse the acceleration structure and identify their intersections with the triangles. Third, the hit information derived in the previous step is used to render the scene. With current GPUs, the first two steps dominate the computation time.

3.1 Construction of Acceleration Structure

We chose BVH as the acceleration structure for its superior construction speed. The BVH construction algorithm proposed by Karras [Karras 2012] was adopted in this work. The algorithm can be decomposed into three steps. Initially, each primitive in a scene is assigned with a Morton code according to its 3-D coordinates. The primitives are sorted and assigned to leaf nodes with regard to their Morton codes. Second, the algorithm constructs internal nodes by exploiting the characteristics of Moron codes. In the third step, the Axis-Aligned Bounding-Box (AABB) is computed for each node in the BVH.

We implemented the algorithm in both OpenCL and CUDA. The construction process is realized as 4 kernels because the first aforementioned step is decomposed into two kernels, Morton code generation and sorting. In the first kernel, we segment the bounding box of the whole scene into 10x10x10 small cubes and each cube is associated with a unique 30-bit Morton code. A thread (thread and work-item are used interchangeably in this paper) is allocated to each triangle to determine its Morton code according to the center coordinates. The second kernel sorts all triangles by their Morton codes and assigns them to the leaf nodes of a binary tree in an ascending order. All leaves in a sub-tree should share a common prefix in their Morton codes. We use the radix sort code released in the CUB package [NVIDIA 2013]. In the third kernel, we launch a thread for each internal node. It searches the neighborhood and identifies the range of leaf nodes covered by the current node. Then the thread continues to search the split point in this range.

After running the above three kernels, we already have the layout of the whole BVH. The next kernel calculates the AABB for each node. In this kernel, we assign a thread for each leaf node. Each thread works upwards until hitting the root. As the number of nodes reduces by half at an upper level, we use an atomic flag to terminate the firstly arrived thread and only allow the second thread to continue working at the upper level.

3.2 Ray Traversal and Intersection Test

We used the BVH traversal algorithm proposed by Aila [Aila and Laine 2009] to perform the traversal process. The algorithm is optimized for GPU execution. It integrates the BVH traversal process and ray-triangle intersection into a single thread. We adopted the CUDA source code released with the paper [Aila and Laine 2009] and ported it to OpenCL. Both CUDA and OpenCL codes were intensively optimized for better efficiency on mobile GPUs.

4 Results and Analysis

The algorithms introduced in the previous section were integrated into a complete ray tracer. We developed a CUDA version for Tegra GPU and an OpenCL version for PowerVR GPU. Both include five kernels, generating Morton Code, sorting the triangles, constructing internal nodes, calculating AABBs and traversal. The traversal is performed at a resolution of 1024x768 with 10 primary rays for each pixel. The performance results are collected on a Jetson TK1 development board (Tegra K1) and an ODROID XU development board (PowerVR), respectively. Three popular scenes illustrated in Figure 1 are used to evaluate the performance of the ray tracers.

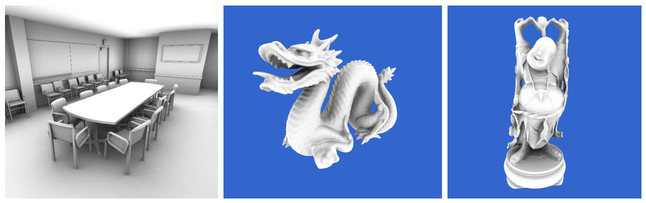
Figure 1: *Test scenes: Conference (283K tris), Dragon (871K tris), Duddha (1.08M tris).*

4.1 Results on NVIDIA Tegra K1

The Tegra K1 GPU integrates 4 Cortex A15 CPUs and a Kepler GPU core. The available memory bandwidth is 13.76GB/s. Our ray tracer is complied with CUDA 6.0 compiler under a Linux for Tegra OS (a derivative version of Ubuntu 12.04). We then evaluate the performance in four aspects, runtime, memory bandwidth, occupancy, and power consumption.

4.1.1 Performance

We compared three implementations, 1 single-threaded version on a Cortex A15 CPU core, a CUDA implementation on Tegra K1 GPU running at 0.852GHz, and a CUDA implementation on GTX 650Ti GPU. The latter GPU is designed for desktop applications. It has 4 SMXs running at 1.033GHz and a memory bandwidth of up to 91 GB/s. The runtimes of 5 kernels as well as the total construction time are listed in Table 1. The construction process consists of four kernels. The sum of their runtimes is the total construction time.

It turns out that the ray tracing engine on the Tegra GPU is quite efficient. The overall implementation is faster than the CPU-based implementation by over one order of magnitude. In addition, the performance of the mobile ray tracer is around 1/4 of that of GTX 650Ti. It can be found that Tegra K1 is more efficient in the calculating AABB kernel than down-scaling GTX 650Ti to a single-SMX GPU. In spite of the lower available memory bandwidth, the

Scene	Platform	Generating Morton Code (ms)	Sorting (ms)	Constructing Internal Node (ms)	Calculating AABBs (ms)	Total Construction Time (ms)	Traversal (MRays/s)
Conference (283K tris)	Tegra K1 GPU	1.56	4.71	3.01	16.08	25.36	69.84
	2.3GHz Cortex A15	103.63	97.45	43.72	124.27	369.07	1.14
	GTX-650Ti (CUDA)	0.21	1.33	0.49	4.55	6.58	341.93
Dragon (871K tris)	Tegra K1 GPU	6.02	10.92	38.52	45.04	100.5	17.71
	2.3GHz Cortex A15	375.48	168.53	555.62	409.01	1508.64	0.43
	GTX-650Ti (CUDA)	1.09	3.42	5.84	14.37	24.72	92.85
Buddha (1.08M tris)	Tegra K1 GPU	7.45	13.14	45.89	54.75	121.23	15.09
	2.3GHz Cortex A15	467.58	338.14	661.76	497.07	1964.55	0.33
	GTX-650Ti (CUDA)	1.34	4.29	6.98	17.81	30.42	73.98

Table 1: *Runtimes of Ray Tracing Workloads (The total construction time of the first four kernels).*

performance of ray tracing seems to be scalable with number of SMXs. Among the five kernels, constructing internal node, calculating ABBs and traversal add up together to consume more than 80% of the GPU computing time.

4.1.2 Memory Bandwidth

We performed detailed profiling to identify the main performance bottleneck of ray tracing on Tegra K1. Table 2 lists the memory characteristics of the three most time-consuming kernels. Tegra is actually a System-on-Chip (SoC) integrating CPU cores, a GPU and other logics. All hardware on the SoC share a dual-channel LPDDR3 memory with a 64-bit bus and a 924-Mhz clock. The peak bandwidth equals 13.76GB/s. On the other hand, the lower bandwidth is compensated by the 128KB L2 cache exclusively used by Tegra K1 GPU, while the competition for L2 cache is more intensive on desktop GPUs.

All kernels except calculating AABBs exhibit low bandwidth utilization (under 50%) relative to the peak bandwidth. Further experiments showed that the 68.68% memory usage of calculating AABBs is acceptable. Micro-benchmarks showed that the memory bus was not saturated at this level of usage and it could sustain an even higher level of memory demand.

metric	Build	AABBs	Traversal
Dram read throughput	0.22 GB/s	6.98 GB/s	4.68G/s
Dram write throughput	0.78 GB/s	2.67 GB/s	0.84G/s
Total Dram Throughput	1.00 GB/s	9.45 GB/s	5.52G/s
Bandwidth Usage Ratio	7.26%	68.68%	40.12%

Table 2: *Memory Bandwidth Profiling. Scene: Buddha.*

4.1.3 Occupancy

Occupancy measures the utilization of GPU resources. It is calculated as the ratio of the number of active threads over the maximum available number of threads on a GPU. Table 3 lists the occupancy values derived on a scene with NVIDIA's profiling tool. Among the three most time-consuming kernels, calculating AABBs has a relatively low occupancy, as the number of nodes is divided by two when moving to an upper level and the parallelism is inherently insufficient in the top 10 levels.

Scene	Build	AABBs	Traversal
Conference	0.78	0.60	0.73
Dragon	0.78	0.47	0.62
Buddha	0.78	0.47	0.61

Table 3: *Profiling of Occupancy.*

4.1.4 Power Consumption

We measured the power performance of ray tracing. We incrementally test the power of the whole Jetson TK1 development board. The average current after booting the system is 0.35A. The peak current during ray tracing test scenes using Tegra GPU is up to 1.09A, while that for the CPU-based ray tracer is 0.85A. That is to say, firstly, with a 12V input voltage, the peak power for the GPU-based ray tracer is 13.08W. Secondly, even though its instantaneous power value is higher than the CPU-based ray tracer, it is still much more energy efficient for the extremely short runtime.

4.2 Results on PowerVR SGX 544MP3

We developed another ray tracer using the same algorithm as described in Section 3 in OpenCL and OpenGL ES on the ODROID XU development board running the Android OS. The development board is equipped with four 1.6GHz Cortex A15 CPUs and a PowerVR SGX 544MP3 GPU core. The memory bus has a bandwidth of 5GB/s. Experiments showed that radix sort is extremely slow on the PowerVR GPU and so we use CPU to perform this kernel for better overall performance.

4.2.1 Performance

The runtime results are listed in Table 4. Ray tracer is much slower on the PowerVR GPU, with no advantage over a CPU implementation. Its performance is mainly limited by the available number of the SIMD units and lower memory bandwidth. In addition, PowerVR GPU is less efficient for handling program divergence in a given group of parallel threads. However, the ray tracing workload is highly divergent.

4.2.2 Performance Profiling

We used the PVRTune profiling tool [Imagination 2014b] to investigate the detail characteristics of the ray tracing workload on the PoworVR GPU. Figure 2 illustrates the profiling results of BVH construction. The data were collected from the Buddha scene. The x-axis is the time, while the y-axis represents the intensity of workload. The curve labeled as "TA (i.e., Tile Acceleration) Load" has a value of 1 if USSE units have work to do and output data to the TA module. It has a value of 0 during sorting kernel because the kernel is conducted by CPU. The curve named "USSE load: Vertex" report the percentage of USSE cores having job to do. The first kernel has a relatively high occupancy, while the calculating AABB kernel suffers from a very low occupancy. It seems that the reduction alike computing pattern of this kernel does not map well to USSEs.

Scene	Platform	Generating Morton Code (ms)	Sorting (ms)	Constructing Internal Node (ms)	Calculating AABBs (ms)	Total Construction Time (ms)	Traversal (MRays/s)
Conference (283K tris)	SGX 544MP3	138	85	36	249	508	2.21
	1.6GHz Cortex A15	223	85	36	105	449	0.56
	GTX-650Ti (OpenCL)	0.19	2.71	0.53	4.16	7.59	341.93
Dragon (871K tris)	SGX 544MP3	339	217	484	1440	2480	0.62
	1.6GHz Cortex A15	763	217	546	551	2077	0.23
	GTX-650Ti (OpenCL)	1.00	7.53	5.93	11.76	26.22	90.15
Buddha (1.08M tris)	SGX 544MP3	376	252	513	1791	2932	0.49
	1.6GHz Cortex A15	1043	252	680	771	2746	0.18
	GTX-650Ti (OpenCL)	1.21	10.94	7.10	15.27	34.52	74.83

Table 4: *Runtimes of Each Kernel on PowerVR SGX 544MP3.*

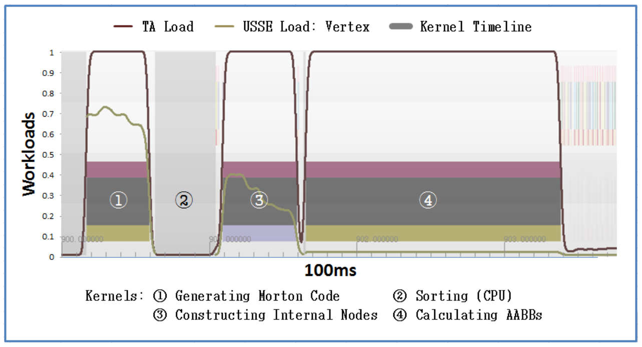

Figure 2: *Profiling ray tracing the Buddha scene.*

4.2.3 Power Consumption

We measured the power of the whole ODROID development board with the SmartPower power meter [ODROID 2013]. The instantaneous power consumption values are shown in Figure 3. The input voltage is 5V and the average power after booting the system is 3.78W. After launching the ray tracer, the peak power consumption is close to 7W. We measured the power consumption of a few other Android applications for comparison. It is 6.13W for playing the game Asphalt 8, and 4.62W for playing a Blu-ray movie. Such an observation suggests that ray tracing on this GPU is relatively inefficient in terms of power consumption.

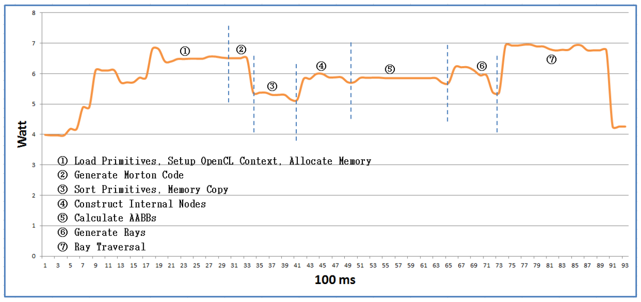

Figure 3: *Instantaneous power consumption of the ODROID XU DEV board during ray tracing the Buddha scene.*

5 Conclusion

As ray tracing is promising to enhance visual experience, this work provides a characterization of the ray tracing workload on two mobile GPUs with different microarchitectures. By integrating state-of-the-art techniques, our ray tracer running on the Tegra K1 GPU allows constructing the BVH of 1M-triangle scene in around 120ms and performing traversal at a throughput of 15 to 70 million rays per second.

Acknowledgements

This work was partially supported by China National Science Foundation under grant number 20121302065. We would like to thank Tong Wang, Zonghui Li, Jinfa Wang and Ruo Jia for the valuable discussions.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics*, 145–149.

IMAGINATION. 2010. *PowerVR Series5XT GPU*. http://www.imgtec.com/powervr/series5xt.asp.

IMAGINATION. 2014. *PowerVR Series6 GPU*. http://www.imgtec.com/powervr/series6.asp.

IMAGINATION. 2014. *PVRTune User Manual Developer*.

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

KELLER, A., KARRAS, T., WALD, I., AILA, T., LAINE, S., BIKKER, J., GRIBBLE, C., LEE, W.-J., AND MCCOMBE, J., 2013. Ray tracing is the future and ever will be.... ACM SIGGRAPH 2013 Courses (SIGGRAPH '13). Article 9, 7 pages.

KHRONOS. 2014. *The open standard for parallel programming of heterogeneous systems*. https://www.khronos.org/opencl/.

LEE, W.-J., SHIN, Y., LEE, J., KIM, J.-W., NAH, J.-H., JUNG, S.-Y., LEE, S.-H., PARK, H.-S., AND HAN, T.-D. 2013. Sgrt: A mobile gpu architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference*, 109–119.

NAH, J.-H., AND MANOCHA, D. 2014. Sato: Surface-area traversal order for shadow ray tracing. *Computer Graphics Forum*, preprint.

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 (Proceedings of SIGGRAPH Asia 2011) 30*, 6, 160:1–160:10.

NVIDIA. 2013. *CUB v1.3.1*. http://nvlabs.github.io/cub/.

NVIDIA. 2014. *CUDA C Programming Guide*.

NVIDIA. 2014. *Whitepaper: NVIDIA Tegra K1 A New Era in Mobile Computing*.

ODROID. 2013. *Smart Power*. http://odroid.com/dokuwiki/doku.php?id=en:odroidsmartpower.

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

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.