TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing

Josef Spjut
Solomon Boulos

Daniel Kopta
School of Computing
University of Utah
Salt Lake City, UT 84112

Erik Brunvand
Spencer Kellis

ABSTRACT

Ray tracing is a technique used for generating highly realistic computer graphics images. In this paper, we explore the design of a simple but extremely parallel, multi-threaded, multi-core processor architecture that performs real-time ray tracing. Our architecture, called TRaX for Threaded Ray eXecution, consists of a set of thread states that include commonly used functional units for each thread and share large functional units through a programmable interconnect to maximize utilization. The memory system takes advantage of the application's read-only access to the scene database and write-only access to the frame buffer output to provide efficient data delivery with a relatively simple structure. Preliminary results indicate that a multi-core version of the architecture running at a modest speed of 500 MHz already provides real-time ray traced images for scenes of a complexity found in video games. We also explore the architectural impact of a ray tracer that uses procedural (computed) textures rather than image-based (look-up) textures to trade computation for reduced memory bandwidth.

1. INTRODUCTION

At present almost every personal computer has a dedicated processor that enables interactive 3D graphics. These graphics processing units (GPUs) implement the *z-buffer* algorithm introduced in Catmull's landmark University of Utah dissertation [6]. In this algorithm the inner loop iterates over all triangles in the scene. Each pixel is updated with the distance to the eye (the "z" value) and the triangle's color unless a smaller distance is already written in the z-buffer (See Figure 1). GPU support for this algorithm involves deep non-branching pipelines of vector floating point operations as the triangles are streamed through the GPU.

These modern GPUs can interactively display several million triangles with image-based (look-up) texture and lighting. The wide availability of GPUs has revolutionized how work is done in many disciplines, and has enabled the hugely successful video game industry. While the commodity hardware implementation of the z-buffer algorithm has allowed excellent interactivity at a low cost, there are (at least) three classes of applications that have not significantly benefited from this revolution: those that have datasets much larger than a few million triangles (Greenberg has argued that typical model sizes are doubling annually [10]) such as vehicle design, landscape design, manufacturing and some branches of scientific visualization, those that have non-polygonal data not easily converted into triangles, and those that demand high quality shadows, reflection, and refraction effects such as architectural lighting design, rendering of outdoor scenes, vehicle lighting design, movies, and perhaps future video games.

These classes of applications typically use Whitted's ray tracing algorithm [31, 9, 28]. The ray tracing algorithm is better suited to huge datasets than the z-buffer algorithm because it more naturally employs hierarchical scene structuring techniques that allow the creation of an image in time sub-linear in the number of objects. The z-buffer algorithm is linear in the number of scene objects that are potentially visible after culling. It is ray tracing's larger time constant and lack of a commodity hardware implementation that makes the z-buffer a faster choice for data sets that are not huge. Ray tracing allows flexibility in the intersection computation for the primitive scene objects. This allows non-polygonal primitives such as splines or curves to be represented directly. Ray tracing is much better suited for creating visual effects such as shadows, reflections, and refractions because it directly simulates the physics of light based on the light transport equation [16, 17]. By directly and accurately computing global visual effects using ray optics ray tracing can create graphics that are problematic (or impossible) for the pure z-buffer algorithm.

While the ray tracing algorithm is not particularly parallel at the instruction level, it is extremely (embarrassingly) parallel at the thread level. The fundamental algorithm for ray tracing is to loop over all pixels on the screen and determine what can be seen by finding the nearest object seen through that pixel. This loop "finds the nearest object" by doing a line query in 3D (Figure 1). This line query, also known as "ray casting" can be repeated recursively to determine shadows, reflections, refractions, and other optical effects. In the extreme, every ray cast in the algorithm can be computed independently. What is required is that every ray have read-only access to the scene database, and write-only access to a pixel in the frame buffer. Importantly, the threads never require

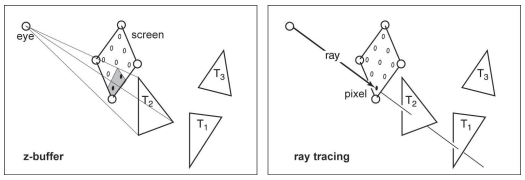

Figure 1: Left: the z-buffer algorithm projects a triangle toward the nine pixel screen and writes all pixels with the distance to the eye (the "z" value) and its color unless a smaller distance is already written in the z-buffer. Each triangle can be processed independently. Right: the ray tracing algorithm sends a 3D half-line (a "ray") into the set of objects and finds the closest one. In this case the triangle T_2 is returned and the color at the intersection point is computed. Each pixel can be processed independently.

1-4244-2334-7/08/$20.00

communication with other threads.

We propose a custom processor architecture for ray tracing called TRaX (Threaded Ray eXecution). The TRaX processor exploits the thread rich nature of ray tracing by supporting multiple thread contexts (thread processors) in each core. We use a form of dynamic data-flow style instruction issue to discover parallelism between threads, and share large less frequently used functional units between thread processors. We explore trade-offs between the number of thread processors versus the number of functional units per core. The memory access style in ray tracing means that a relatively simple memory system can keep the multiple threads supplied with data. However, adding detailed image-based (look-up) textures to a scene can dramatically increase the required memory bandwidth (as it does in a GPU). We also explore procedural (computed) textures as an alternative that trades computation for memory bandwidth. The resulting multiple-thread core can be repeated on a multi-core chip because of the independent nature of the computation threads. Other thread-rich applications that could take advantage of the TRaX architecture include chess AI, image and video processing, and cryptography.

2. BACKGROUND AND PREVIOUS WORK

Ray tracing can, of course, be implemented on general purpose CPUs, and on specially programmed GPUs. Both approaches have been studied, along with a few previous studies of custom architectures.

2.1 Graphics Processing Units

A carefully crafted computational pipeline for transforming triangles and doing depth checks along with an equally carefully crafted memory system to feed those pipelines makes our current generation of z-buffer GPUs possible [1, 19]. Current GPUs have up to hundreds floating point units on a single GPU and aggregate memory bandwidth of 20-80 Gbytes per second from their on-chip memories. That impressive on-chip memory bandwidth is largely to support image-based (look-up) textures for the primitives. These combine to achieve graphics performance that is orders of magnitude higher than could be achieved by running the same algorithms on a general purpose processor.

The processing power of a GPU depends, to a large degree, on the independence of each triangle being processed in the z-buffer algorithm. This is what makes it possible to stream triangles through the GPU at rapid rates, and what makes it difficult to map ray tracing to a traditional GPU. There are three fundamental operations that must be supported for ray tracing: intersecting a ray with the acceleration structure that encapsulates the scene objects, intersecting the ray with the primitive objects contained in the element of the bounding structure that is hit, and computing the illumination and color of the pixel based on the intersection with the primitive object and the collection of the contributions from the secondary ray segments. These operations require branching, pointer chasing, and decision making in each thread, and global access to the scene database: operations that are relatively inefficient in a z-buffer-based architecture.

While it is possible to perform ray tracing on GPUs [24, 2, 11], these implementations have not been faster than the best CPU implementations, and they require the entire model to be in graphics card memory. While some research continues on improving such systems, the traditional GPU architecture makes it unlikely that the approach can be used on large geometric models. In particular the inefficiency of branching based on computations performed on the GPU, and the restricted memory model are serious issues for ray tracing on a traditional GPU. The trend, however, in GPU architecture is towards more and more programmability of the graphics pipeline. Current high-end GPUs such as the G80 from nVidia, for example [21, 20], support both arbitrary memory accesses and branching in the instruction set, and can thus, in theory, do both pointer chasing and frequent branching. However, a G80 assumes that every set of 32 threads (a "warp") essentially executes the same instruction, and that they can thus be executed in SIMD manner. Branching is realized by (transparently) masking out threads. Thus, if branching often leads to diverging threads very low utilization and performance will occur (similar arguments apply to pointer chasing). Results for parts of the ray tracing algorithm on a G80 have been reported [11], but to date no complete ray tracing systems have been reported on these new platforms.

2.2 General CPU Architectures

General purpose architectures are also evolving to be perhaps more compatible with ray tracing type applications. Almost all commodity processors are now multi-core and include SIMD extensions in the instruction set. By leveraging these extensions and structuring the ray tracer to trace coherent packets of rays, researchers have demonstrated good frame rates even on single CPU cores [30, 4]. The biggest difference in our approach is that we don't depend on the coherence of the ray packet to extract thread-level parallelism. Thus our hardware should perform well even for secondary rays used in advanced shading effects for which grouping the individual rays into coherent packets may not be easy.

The IBM Cell processor [15, 14] is an example of an architecture that might be quite interesting for ray tracing. With a 64-bit in-order power processor element (PPE) core (based on the IBM Power architecture) and eight synergistic processing elements (SPE), the Cell architecture sits somewhere between a general CPU and a GPU-style chip. Each SPE contains a 128x128 register file, 256kb of local memory (not a cache), and four floating point units operating in SIMD. When clocked at 3.2 GHz the Cell has a peak processing rate of 200GFlops. Researchers have shown that with careful programming, and with using only shadow rays (no reflections or refractions) for secondary rays, a ray tracer running on a Cell can run 4 to 8 times faster than a single-core x86 CPU [3]. In order to get those speedups the ray tracer required careful mapping into the scratch memories of the SPEs and managment of the SIMD branching supported in the SPEs. We believe that our architecture can improve on those performance numbers while not relying on coherent packets of rays executing in a SIMD fashion.

2.3 Ray Tracing Hardware

Other researchers have developed special-purpose hardware for ray tracing [18, 12]. The most complete of these are the SaarCOR [25, 26] and Ray Processing Unit (RPU) [33, 32] architectures from Saarland University. SaarCOR is a custom hard-coded ray trace processor, and RPU has a custom K-D tree traversal unit with a programmable shader. Both are implemented and demonstrated on an FPGA. With an appropriately described scene (using K-D trees and triangle data encoded with unit-triangle transformations) the RPU can achieve very impressive frame rates, especially when extrapolated to a potential CMOS ASIC implementation [32].

Our design is intended to be more flexible than the RPU by having all portions of the ray tracing algorithm be programmable, allowing the programmer to decide the appropriate acceleration structure and primitive encoding, and by accelerating single ray performance rather than using four-ray SIMD packets. There is, of course, a cost in terms of performance for this flexibility, but if adequate frame rates can be achieved it will allow our architecture to be used in a wider variety of situations. There are many other

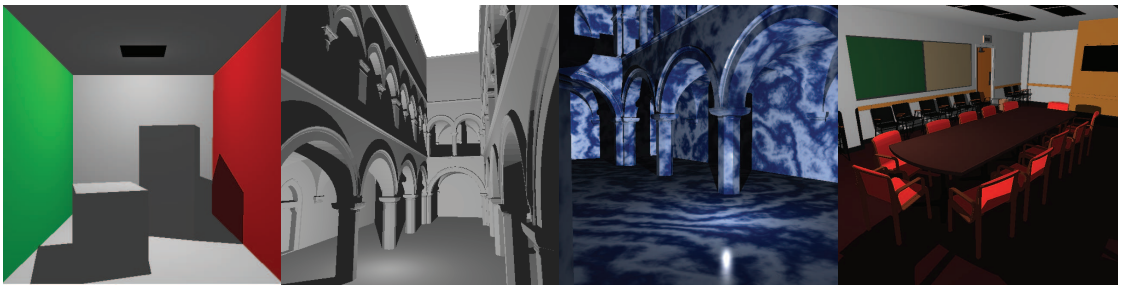

Figure 2: Test scenes rendered on our TRaX architectural simulator. From left to right: Cornell, Sponza, Sponza with procedural textures, Conference. These are standard benchmarking scenes for ray tracing.

applications that share the thread-parallel nature of ray tracing.

3. TRAX ARCHITECTURE DESIGN

Our overall chip design (Figure 3) is a die consisting of a L2 cache with an interface to off-chip memory and a number of repeated identical cores with multiple threads each. Due to the low communication requirements of the threads, each core only needs access to the same read only memory and the ability to write to the frame buffer. The only common memory is provided by an atomic increment instruction that provides a different value each time the instruction is executed

3.1 A Single Thread Processor

Each thread processor (TP) in a TRaX core can execute its own thread code, where a software thread corresponds to a ray. The thread has a fixed-size register file as well as local access to a set of functional units that are contained within the TP. Figure 5 shows these functional units as well as the register file. The type and number of these functional units is variable in our simulator. More complex functional units are shared by the TPs in a core.

Instructions are issued in-order in each Thread Processor to reduce the complexity at the thread level. An instruction may stall waiting for a previous instruction to complete, but correct single thread execution is guaranteed.

3.2 Design of a Single Core

A core consists of a collection of Thread Processors (TPs), shared functional units, shared issue and load/store units, and a shared L1 cache. Each thread executes concurrently and keeps independent program state. Figure 4 shows how the threads and functional units are arranged within a core. The issue unit keeps the instruction memory that is the same for all threads since they execute the same code. It also keeps the program counters for each of the threads in the core. Each thread has access to its exclusive functional units. Through a simple interconnect fabric we allow each thread to issue instructions to the shared functional units as well.

In order to access main memory, each core has a shared L1 cache of a modest size (see Section 6) to exploit locality and screen coherence within the core. All load instructions use the cache to read from memory. While standard cached writes are supported by the simulator, the code we execute does not require this feature. To prevent cache pollution by our frame buffer writes, all of the writes to the output frame buffer in our code write around the cache to the screen buffer output.

Each functional unit is independently pipelined to complete execution in a given number of cycles, with the ability to issue a new instruction each cycle. In this way, each thread is potentially able to issue any instruction on any cycle. With the shared functional units, memory latencies and possible dependence issues, not all threads may be able to issue on every cycle. The issue unit gives threads priority to claim shared functional units in a round robin fashion.

Each thread state controls the execution of one ray-thread. Because the parallelism we intend to exploit is at the thread level, and not at the instruction level inside a thread, many features commonly found in modern microprocessors, such as out-of-order execution, complex multi-level branch predictors, and speculation, are eliminated from our architecture. This allows available transistors, silicon area, and power to be devoted to parallelism. In general, complexity is sacrificed for expanded parallel execution. This will succeed in offering high-performance ray tracing if we can keep a large number of threads issuing on each cycle (see Section 6.1).

4. SIMULATION METHODOLOGY

We have developed a flexible, detailed simulator to analyze performance across many hardware configurations. Given the unique nature of our architecture, it was not reasonable to adapt available simulators to our needs. In the style of Simplescalar [5], our cycle-accurate simulator allows for very quick prototyping, customization and extension. Chip-level results are extrapolated from the results of these core simulations. Functional units are added to the simulator in a modular fashion, allowing us to support arbitrary combinations and types of functional units and instructions. This allows very general architectural exploration starting from our basic thread-parallel execution model. Unlike a Verilog or VHDL description, however, we cannot directly map this description in to a hardware realization and thus cannot guarantee that our proposed hardware model will fit in the area we would have available on a die, or draw reasonable electrical current. We can, however, make reasonably accurate area estimates of the major components of the architecture such as register files, caches, and arithmetic units by synthesizing these units separately. We perform no power simulations as our main concern is to achieve high utilization of our functional units and expect power to be on par with modern GPUs.

The simulator accepts a configuration file describing a set of Functional Units including memory units and command-line options specifying the number of hardware threads desired and an input scene. Statistics provided by the simulator include total cycles used to generate a scene, functional unit utilization, thread utilization, thread stall behavior, memory and cache bandwidth, memory and cache usage patterns, and total parallel speedup.

Currently, the actual code run is a simple single-ray (i.e., pack-

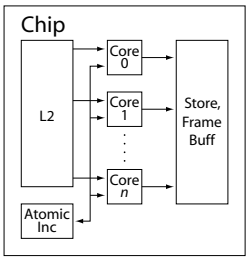

Figure 3: Multi-Core Chip Layout

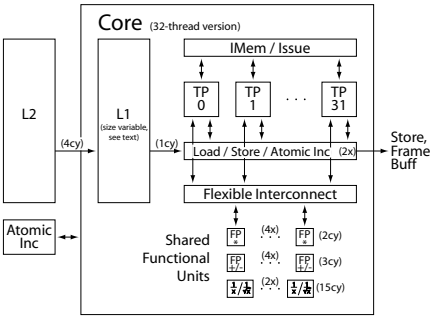

Figure 4: Core Block Diagram

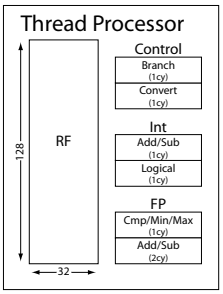

Figure 5: Thread Processor State

etless) ray caster that uses a bounding volume hierarchy (BVH) as its acceleration structure. We use the surface area heuristic to build our hierarchy so that our simulated ray casting has memory behavior similar to that experienced by more full featured ray tracers. More details of this code are found in the next section.

Our ray tracing code is executed on simulated processors having between 1 and 256 threads, with the number of all function units varying between 1 and 64. The number of read and write ports to the L1 cache was also varied. Images may be generated for any desired screen size (see Figure 2 for examples of test scenes). Our primary goal for the current design phase is to determine the optimal allocation of transistors to thread-level resources, including functional units and thread state, in a single core to maximize utilization and overall parallel speedup. We are also looking carefully at memory models and memory and cache usage to feed the parallel threads (and parallel cores at the chip level).

For each simulation we render one frame in one core from scratch with cold caches. The instructions are assumed to be already in the instruction cache since they don't change from frame to frame. The results we show are therefore an accurate representation of changing the scene memory on every frame and requiring invalidating the caches. The results are conservative because even in a dynamic scene, much of the scene might stay the same from frame to frame and thus remain in the cache. We determine the time required to render a single frame and extrapolate the number of frames per second from these results.

4.1 Functional Units

For a simple ray casting application, large, complex instruction sets such as those seen in modern x86 processors are unnecessary. Our architecture implements a basic set of functional units with a simple but powerful ISA. We include bitwise instructions, branching, floating point/integer conversion, memory operations, floating point and integer add, subtract, multiply, reciprocal, and floating point compare. We also include reciprocal square root because that operation occurs with some frequency in graphics code for normalizing vectors.

We first chose a set of functional units to include in our machine-level language, shown in Table 1. This mix was chosen by separating different instruction "classes" into separate dedicated functional units. We implemented a ray casting benchmark using these available resources, then ran numerous simulations varying the number of threads and the width of each functional unit. All execution units are assumed to be pipelined including the memory unit.

Each thread receives its own private FP Add/Sub execution unit. FP multiply is a crucial operation for ray-tracing. Cross and dot products, both of which require multiple FP multiplies, are common in ray tracing applications. Other common operations such as blending also use FP multiplies. The FP multiplier is a shared unit because of its size, but due to its importance, it is only shared among a few threads. The FP Inv functional unit handles divides and reciprocal square roots. The majority of these instructions come from our box test algorithm, which issues three total FP Inv instructions. This unit is very large and less frequently used hence, it is shared among a greater number of threads.

Table 1: Default Functional Unit Mix (500MHz cycles)

Unit Name	Number of units	Latency (cycles)
IntAddSub	1 / thread	1
IntMul	1 / 8 threads	2
FPAddSub	1 / thread	2
FPMul	1 / 8 threads	3
FPComp	1 / thread	1
FPInvSqrt	1 / 16 threads	15
Conversion	1 / thread	1
Branch	1 / thread	1
Cache	1 (mult. banks)	varies

Table 2: Area Estimates (pre-layout) for functional units using Artisan CMOS libraries and Synopsys. IBM8RF is a 130nm high performance cell library. IBM10LP is a 65nm low power cell library. Speed is similar for each circuit.

	Area (μm^2)	
Resource Name	IBM8RF	IBM10LP
2kX16byte SRAM	1,761,578	503,000(est.)
128X32 RF	77,533	22,00(est.)
Integer Add/Sub	1,967	577
Integer Multiply	30,710	12,690
FP Add/Sub	14,385	2,596
FP Multiply	27,194	8,980
FP Compare	1,987	690
FP InvSqrt	135,040	44,465
Int to FP Conv	5,752	1,210

5. RAY TRACING APPLICATION

The ray tracing application used in all of our tests is written by hand in assembly language to take advantage of the functional units and memory model supported in our architecture. The application

Table 3: Scene Data with Results for a single 32-Thread TRaX Core with Phong Shading estimated at 500MHz

Scene	Triangles	BVH Nodes	FPS
conference	282664	266089	1.4282
sponza	66454	58807	1.1193
cornell	32	33	4.6258

provides various shading methods, shadows from a single point light source and BVH traversal. We have rendered three scenes in our simulator, well known in the ray tracing literature as benchmarks: The Cornell Box, Sponza, and the Conference Room scene (Figure 2).

The test scenes we are using (Table 3) exhibit some important properties. The Cornell Box is important because it represents the simplest type of scene that would be rendered. It gives us an idea of the maximum performance possible by our hardware. Sponza on the other hand has over 65000 triangles and uses a BVH with over 50000 nodes. This is a more realistic example of a scene similar in complexity to simple video game scenes. The Conference Room scene is an example of a very large and complex scene with around 300k triangles. This is similar to a typical modern video game scene. Even more complicated scenes including dynamic components will be included in testing as more progress is made.

5.1 Shading Methods

Our ray tracer implements two of the most commonly used shading methods in ray tracing: simple diffuse scattering, and Phong lighting for specular highlights [27, 8]. We also include simple hard shadows from a single point light source. Shadow rays are generated and cast from each intersected primitive to determine if the hit location is in shadow (so that it is illuminated only with an ambient term) or lit (so that it is shaded with ambient, diffuse and Phong lighting).

Diffuse shading assumes that light scatters in every direction equally, and Phong lighting adds specular highlights to simulate shiny surfaces by increasing the intensity of the light if the view ray reflects straight into a light source. These two shading methods increase the complexity of the computation per pixel, increasing the demand on our FUs. Phong highlights especially increase complexity, as they involve taking an integer power, as can be seen in the standard lighting model:

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s$$

The I_p term is the shade value at each point which uses constant terms for the ambient k_a, diffuse k_d, and specular k_s components of the shading. The α term is the Phong exponent that controls the shininess of the object by adjusting the specular highlights. The i terms are the intensities of the ambient, diffuse, and specular components of the light sources.

5.2 Texturing

We also implement procedural textures, that is, textures which are computed based on the geometry in the scene, rather than an image texture which is simply loaded from memory. Specifically, we use Perlin noise with turbulence [22, 13]. These textures are computed using pseudo-random mathematical computations to simulate natural materials which adds a great deal of visual realism and interest to a scene without the need to store and load complex textures from memory. The process of generating noise is quite computationally complex. First, the texture coordinate on the geometry where the ray hit is used to determine a unit lattice cube that encloses the point. The vertices of the cube are hashed and used to look up eight pre-computed pseudo-random vectors from a small table. For each of these vectors, the dot product with the offset from the texture coordinate to the vector's corresponding lattice point is found. Then, the values of the dot products are blended using either Hermite interpolation (for classic Perlin noise [22]) or a quintic interpolant (for improved Perlin noise [23]) to produce the final value. More complex pattern functions such as turbulence produced through spectral synthesis sum multiple evaluations of Perlin noise for each point shaded. There are 672 floating point operations in our code to generate the texture at each pixel. We ran several simulations comparing the instruction count of an image with and without noise textures. We found that there are on average 50 percent more instructions required to generate an image where every surface is given a procedural texture than an image with no textures.

Perlin noise increases visual richness at the expense of computational complexity, while not significantly affecting memory traffic. The advantage of this is that we can add more FUs at a much lower cost than adding a bigger cache or more bandwidth. Conventional GPUs require an extremely fast memory bus and a very large amount of RAM for storing textures [1, 19]. Some researchers believe that if noise-based procedural textures were well supported and efficient, that many applications, specifically video games, would choose those textures over the memory-intensive image-based textures that are used today [29]. An example of a view of the Sponza scene rendered with our Perlin noise-based textures can be seen in Figure 2.

6. RESULTS

Many millions of cycles of simulation were run to characterize our proposed architecture for the ray-tracing application. We used frames per second as our principle metric extrapolated from single-core results to multi-core estimates. This evaluation is conservative in many respects since much of the scene data required to render the scene would likely remain cached between consecutive renderings in a true 30-fps environment. However, it does not account for repositioning of objects, light sources, and viewpoints. The results shown here describe a preliminary analysis based on simulation.

We target $200mm^2$ as a reasonable die size for a high-performance graphics processor. We used a low power 65nm library to conservatively estimate the amount of performance achievable in a high density, highly utilized graphics architecture. We also gathered data for high performance 130nm libraries as they provide a good comparison to the RPU and achieve roughly the same clock frequency as the low power 65nm libraries.

Basic functional units, including register files and caches, were synthesized and placed-and-routed using Synopsys tools to generate estimated sizes. These estimates are conservative, since hand-designed execution units will likely be much smaller. We use these figures with simple extrapolation to estimate the area required for a certain number of cores per chip given replicated functional units and necessary memory blocks for thread state. Since our area estimates do not include an L2 cache or any off-chip I/O logic, our estimates in Table 4 and Table 5 are limited to $150mm^2$ in order to allow room for the components that are currently unaccounted for.

6.1 Performance

For a ray tracer to be considered to achieve real-time performance, it much have a frame rate of about 30 fps. The TRaX ar-

Table 4: Core Area Estimates to Achieve 30 FPS on Conference. These estimates include the multiple cores as seen in Figures 3 and 4, but do not include the chip-wide L2 cache, memory management, or other chip-wide units.

Thrds /Core	CoreArea mm^2 130 nm	65 nm	Core FPS	Cores	DieArea mm^2 130 nm	65 nm
16	4.73	1.35	0.71	43	203	58
32	6.68	1.90	1.42	22	147	42
64	10.60	2.99	2.46	15	138	39
128	18.42	5.17	3.46	9	166	47

Table 5: Performance comparison for Conference and Sponza assuming a fixed chip area of 150mm^2. This fixed chip area does not include the L2 cache, memory management, and other chip-wide units. It is assumed that those units would increase the chip area by a fixed amount.

Threads /Core	# of Cores 130 nm	65 nm	Conference 130 nm	65 nm	Sponza 130 nm	65 nm
16	32	111	22.7	79.3	17.7	61.7
32	22	79	31.9	112.3	24.1	85.1
64	14	50	34.8	123.6	24.0	85.4
128	8	29	28.2	100.5	17.5	62.4

chitecture is able to render the conference scene at 31.9 fps with 22 cores on a single chip at 130nm. At 65nm with 79 cores on a single chip performance jumps to 112.3 fps.

The number of threads able to issue in any cycle is a valuable measure of how well we are able to sustain parallel execution by feeding threads enough data from the memory hierarchy and offering ample issue availability for all execution units. Figure 7 shows, for a variable number of threads in a single core, the average percentage of threads issued in each cycle. For 32 threads and below, we issue nearly 50% of all threads in every cycle on average. For 64 threads and above, we see that the issue rate drops slightly ending up below 40% for the 128 threads rendering the Sponza scene, and below 30% for the Conference scene.

Considering a 32 thread core with 50% of the threads issuing each cycle, we have 16 instructions issued per cycle per core. In the 130nm process, we fit 16 to 22 cores on a chip. Even at the low end, the number of instructions issued each cycle can reach up to 256. With a die shrink to 65 nm we can fit more than 64 cores on a chip allowing the number of instructions issued to increase to 1024 or more. Since we never have to flush the pipeline due to incorrect branch prediction or speculation, we potentially achieve an average IPC of more than 1024. Even modern GPUs with many concurrent threads, issue a theoretical maximum of around 256 (128 threads issuing 2 floating point ops per cycle).

Another indicator of sustained performance is the average utilization of the shared functional units. The FP Inv unit shows utilization at 70% to 75% for the test scenes. The FP Multiply unit has 50% utilization and Integer Multiply has utilization in the 25% range. The functional unit configuration is discussed later.

A good cache should exhibit high hit rates to reduce memory latency and external bandwidth requirements. We find that even with a relatively small 2K x 16-byte L1 data cache, hit rates remain in the 95% range. We attribute this performance to low cache pollution because all stores go around the cache.

6.2 Cache Performance

We varied cache size and issue width to determine an appropriate

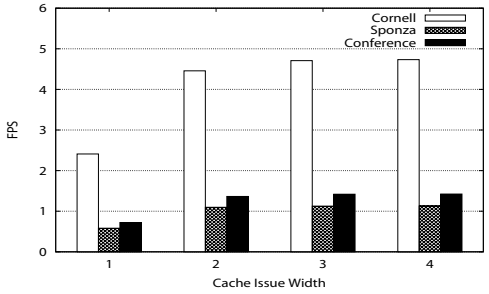

Figure 6: Single core performance as Cache Issue Width is varied.

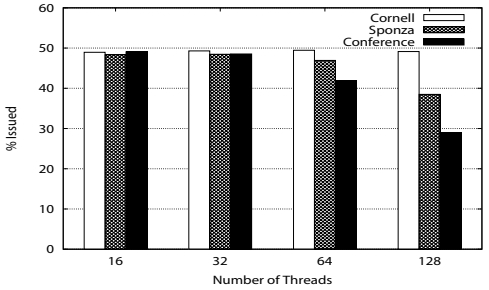

Figure 7: Thread Performance (% Issued)

configuration offering high performance balanced with reasonable area and complexity. For scenes with high complexity a cache with at least 2K lines (16-bytes each) satisfied the data needs of all 32 threads executing in parallel. Although performance continued to increase slightly with larger cache sizes, the extra area required to implement the larger cache meant that total silicon needed to achieve 30fps actually increased beyond a 2K L1 data cache size. To evaluate the number of read ports needed, we simulated a large (64K) cache with between 1 and 32 read ports. Three read ports provided sufficient parallelism for 32 threads.

6.3 Comparison

Comparing against the Saarland RPU [33, 32], our frame rates are higher in the same technology, and our flexibility is enhanced by allowing all parts of the ray tracing algorithm to be programmable instead of just the shading computations. This allows our application to use (for example) any acceleration structure and primitive encoding, and allows the hardware to be used for other applications that share the thread-rich nature of ray tracing.

A ray tracing application implemented on the cell processor [3] shows moderate performance as well as the limitations of an architecture not specifically designed for ray tracing. In particular our hardware allows for many more threads executing in parallel and trades off strict limitations on the memory hierarchy. The effect can be seen in the TRaX performance at 500MHz compared to Cell performance at 3.2GHz. Table 6 shows these comparisons.

7. CONCLUSION

We have shown that a simple, yet powerful, multi-threaded architecture can perform real-time ray tracing running at modest clock speeds on achievable technology. By exploiting the coherence among

Table 6: Performance comparison for Conference against Cell and RPU. Comparison in frames per second and million rays per second(mrps). All numbers are for shading with shadows. TRaX and RPU numbers are for 1024x768 images. Cell numbers are for 1024x1024 images and so the Cell is best compared using the mrps metric which factors out image size.

	TRaX		IBM Cell[3]		RPU[32]	
	130nm	65nm	1 Cell	2 Cells	DRPU4	DRPU8
fps	31.9	112.3	20.0	37.7	27.0	81.2
mrps	50.2	177	41.9	79.1	42.4	128
process	130nm	65nm	90nm	90nm	130nm	90nm
area (mm^2)	≈ 200	≈ 200	≈ 220	≈ 440	≈ 200	≈ 190
Clock	500MHz	500MHz	3.2GHz	3.2GHz	266MHz	400MHz

primary rays with similar direction vectors, the cache hit rate is very high, even for small caches. There is still potential to gain even more benefit from primary ray coherence by assigning nearby threads regions of the screen according to a space filling curve.

With the help of our cycle-accurate simulator we expect to improve the performance of our system along many dimensions. In particular, there may be potential for greater performance by using a streaming memory model for an intelligently selected subset of memory accesses in parallel with the existing cache memory. Ray/BVH intersection in particular will likely benefit dramatically from such a memory system. We will also improve the memory system in the simulator to more accurately simulate L2 cache performance.

We are in the process of improving the ray tracing application used to drive the architectural exploration to include more features. The goal is to allow for Cook style ray tracing [7] with support for multisampling. This will allow for features like soft shadows, motion blur, fuzzy reflections and depth of field. Additionally, we will add support for image based textures as a comparison against procedural textures. Some researchers anticipate that a strong niche for real time ray tracing will involve shallow ray trees (i.e. few reflections), and mostly computed textures [29]. Computed textures using, for example, Perlin noise techniques [22, 13] increase FP ops by about 50% in the worst case (all primitives use these procedural textures), but have a negligible impact on memory bandwidth. This can reserve the memory bandwidth for the portion of the scene with mapped textures and have a positive impact on total system performance by trading computation for memory bandwidth.

We have described an architecture which achieves physically realistic, real-time ray tracing with realistic size constraints. Our evaluation has shown that TRaX performs competitively or outperforms other ray tracing architectures, and does so with greater flexibility at the programming level.

Acknowledgments

The authors thank the other members of the HWRT group: Al Davis, Andrew Kensler, Steve Parker, and Pete Shirley. This material is based upon work supported by the National Science Foundation under Grant No. CCF0541009.

8. REFERENCES

[1] ATI. Ati products from AMD. http://ati.amd.com/products/index.html.

[2] D. Balciunas, L. Dulley, and M. Zuffo. Gpu-assisted ray casting acceleration for visualization of large scene data sets. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing RT06*, sep 2006.

[3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing RT06*, Sept. 2006.

[4] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*, May 2007.

[5] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[6] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, December 1974.

[7] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *Proceedings of SIGGRAPH*, pages 165–174, 1984.

[8] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[9] A. Glassner, editor. *An introduction to ray tracing*. Academic Press, London, 1989.

[10] 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*, pages 477–494, 1997.

[11] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing RT07*, pages 113–118, Sept. 2007.

[12] D. Hall. The AR350: Today's ray trace rendering processor. In *Proceedings of the EUROGRAPHICS/SIGGRAPH workshop on Graphics Ha rdware - Hot 3D Session*, 2001.

[13] J. C. Hart. Perlin noise pixel shaders. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–94, New York, NY, USA, 2001. ACM Press.

[14] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[15] IBM. The Cell project at IBM research. http://www.research.ibm.com/cell.

[16] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffuse environments. In *Proceedings of SIGGRAPH*, pages 133–142, 1986.

[17] J. T. Kajiya. The rendering equation. In *Proceedings of SIGGRAPH*, pages 143–150, 1986.

[18] H. Kobayashi, K. Suzuki, K. Sano, and N. O. ba. Interactive Ray-Tracing on the 3DCGiRAM Architecture. In *Proceedings of ACM/IEEE MICRO-35*, 2002.

[19] nVidia Corporation. www.nvidia.com.

[20] nVidia CUDA Documentation. http://developer.nvidia.com/object/cuda.html.

[21] nVidia G80 Documentation. http://www.nvidia.com/page/geforce8.html.

[22] K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, 1985.

[23] K. Perlin. Improving noise. *ACM Transactions on Graphics*, 21(3):681–682, 2002.

[24] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[25] J. Schmittler, I. Wald, and P. Slusallek. SaarCOR – A Hardware Architecture for Realtime Ray-Tracing. In *Proceedings of EUROGRAPHICS Workshop on Graphics Hardware*, 2002. available at http://graphics.cs.uni-sb.de/Publications.

[26] J. Schmittler, S. Woop, D. Wagner, P. Slusallek, and W. J. Paul. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of Graphics Hardware*, pages 95–106, 2004.

[27] P. Shirley. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[28] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Natick, MA, 2003.

[29] P. Shirley, K. Sung, E. Brunvand, A. Davis, S. Parker, and S. Boulos. Rethinking graphics and gaming courses because of fast ray tracing. In *SIGGRAPH '07: ACM SIGGRAPH 2007 educators program*, 2007.

[30] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[31] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[32] S. Woop, E. Brunvand, and P. Slusallak. Estimating performance of an ray tracing ASIC design. In *IEEE Symposium on Interactive Ray Tracing (RT06)*, September 2006.

[33] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. In *Proceedings of International Conference on Computer Graphics and Interactive Techniques*, pages 434–444, 2005.