TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing

Josef Spjut, *Student Member, IEEE*, Andrew Kensler, Daniel Kopta, *Student Member, IEEE*, and Erik Brunvand, *Member, IEEE*

***Abstract*—Threaded Ray eXecution (TRaX) is a highly parallel multithreaded multicore processor architecture designed for real-time ray tracing. The TRaX architecture consists of a set of thread processors that include commonly used functional units (FUs) for each thread and that share larger FUs through a programmable interconnect. 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 memory system. One specific motivation behind TRaX is to accelerate single-ray performance instead of relying on ray packets in single-instruction–multiple-data mode to boost throughput, which can fail as packets become incoherent with respect to the objects in the scene database. In this paper, we describe the TRaX architecture and our performance results compared to other architectures used for ray tracing. Simulated results indicate that a multicore version of the TRaX architecture running at a modest speed of 500 MHz provides real-time ray-traced images for scenes of a complexity found in video games. We also measure performance as secondary rays become less coherent and find that TRaX exhibits only minor slowdown in this case while packet-based ray tracers show more significant slowdown.**

***Index Terms*—Computer architecture, computer graphics, ray tracing.**

I. Introduction

At PRESENT, almost every personal computer has a dedicated processor that enables interactive 3-D graphics. These graphics processing units (GPUs) implement the z-*buffer* algorithm introduced in Catmull's landmark University of Utah dissertation [1]. In this algorithm, the inner loop iterates over all triangles in the scene and projects those triangles to the screen. It computes the distance to the screen (the z-value) at each pixel covered by the projected triangle and stores that distance in the z-buffer. Each pixel is updated to the color of the triangle (perhaps through a texture lookup or a procedural texturing technique) unless a smaller distance, and, thus, a triangle nearer to the screen, has already been written to the z-buffer (see Fig. 1). A huge benefit of this approach is that all triangles can be processed independently with no knowledge of other objects in the scene. Current mainstream graphics processors use highly efficient z-buffer rasterization hardware to achieve impressive performance in terms of triangles processed per second. This hardware generally consists of deep nonbranching pipelines of vector floating-point (FP) operations as the triangles are streamed through the GPU and specialized memory systems to support texture lookups. However, the basic principle of z-buffer rasterization, that triangles are independent, becomes a bottleneck for highly realistic images. This assumption limits shading operations to per-triangle or per-pixel computations and does not allow for directly computing global effects such as shadows, transparency, reflections, refractions, or indirect illumination. Tricks are known to approximate each of these effects individually, but combining them is a daunting problem for the z-buffer algorithm.

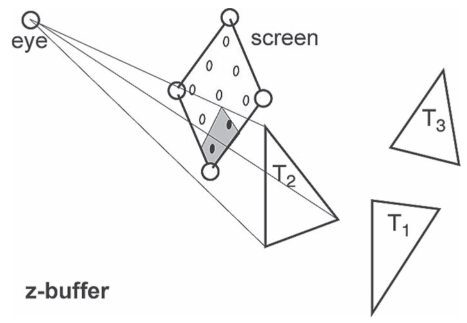

Fig. 1. z-buffer algorithm projects a triangle toward the nine-pixel screen and updates all pixels with the distance to the eye (the "z" value) and the triangle's color unless a smaller distance is already written in the z-buffer.

Modern GPUs can interactively display several million triangles in complex 3-D environments with image-based (lookup) texture and lighting. The wide availability of GPUs has revolutionized how work is done in many disciplines and has been a boon to the hugely successful video game industry. While the 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 benefited significantly from this revolution:

1) those that have data sets much larger than a few million triangles, such as vehicle design, landscape design, manufacturing, and some branches of scientific visualization;
2) those that have nonpolygonal data not easily converted into triangles;
3) those that demand high-quality shadows, reflection, refraction, and indirect illumination effects, such as architectural lighting design, rendering of outdoor scenes, and vehicle lighting design.

Manuscript received December 1, 2008; revised March 17, 2009 and July 14, 2009. Current version published November 18, 2009. This work was supported by the National Science Foundation under Grant CCF0541009. This paper was recommended by Associate Editor L. Pozzi.

The authors are with the School of Computing, University of Utah, Salt Lake City, UT 84112 USA (e-mail: elb@cs.utah.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2028981

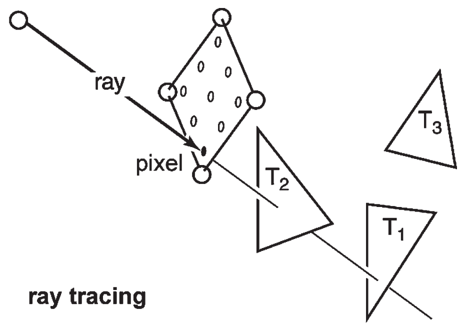

Fig. 2. Ray-tracing algorithm sends a 3-D half line (a "ray") into the set of objects and finds the closest one. In this case, the triangle T_2 is returned.

These classes of applications typically use Whitted's ray-tracing algorithm [2]–[4]. The ray-tracing algorithm is better suited to huge data sets than the z-buffer algorithm because its natural use of hierarchical scene structuring techniques allows image rendering time that is sublinear in the number of objects. While z-buffers can use some hierarchical culling techniques, the basic algorithm is linear with respect to the number of objects in the scene. 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 is better suited for creating shadows, reflections, refractions, and indirect illumination effects because it can directly simulate the physics of light based on the light transport equation [5], [6]. By directly and accurately computing composite global visual effects using ray optics, ray tracing can create graphics that are problematic for the z-buffer algorithm. Ray tracing also allows flexibility in the intersection computation for the primitive objects, which allows nonpolygonal primitives such as splines or curves to be represented directly. Unfortunately, computing these visual effects based on simulating light rays is computationally expensive, particularly on a general-purpose CPU. The ray-tracing algorithm currently requires many high-performance CPUs to be interactive at full-screen resolution.

While the ray-tracing algorithm is not particularly parallel at the instruction level, it is extremely (embarrassingly) parallel at the thread level. Ray tracing's inner loop considers each pixel on the screen. At each pixel, a 3-D half line (a "ray") is sent into the set of objects and returns information about the closest object hit by that ray. The pixel is colored (again, perhaps using texture lookups or a procedurally computed texture) according to this object's properties (Fig. 2). 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 has read-only access to the scene database and write-only access to a pixel in the frame buffer. Importantly, threads never have to communicate with other threads (except to partition work among the threads, which is done using an atomic increment instruction in our implementation). This type of memory utilization means that a relatively simple memory system can keep the multiple threads supplied with data.

To summarize, the parallelization of rasterizing happens by processing triangles in parallel through multiple triangle-processing pipelines that can operate concurrently. Ray tracing processes pixels/rays in parallel. Each pixel corresponds to a primary ray (or a set of primary rays in an oversampled implementation) from the eye into the scene. These primary rays may spawn additional secondary rays, but all those rays can continue to be processed concurrently with every other ray.

This paper is an extended version of a previous conference paper [7] in which we propose a custom processor architecture for ray tracing called Threaded Ray eXecution (TRaX). This paper adds to that paper additional details of the memory system and significant results related to TRaX's ability to handle noncoherent secondary rays.

The TRaX processor exploits the thread-rich nature of ray tracing by supporting multiple-thread contexts [thread processors (TPs)] 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 (FUs) between TPs. We explore tradeoffs between the number of TPs versus the number of FUs 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 (lookup) 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 multicore chip because of the independent nature of the computation threads. We evaluate the performance of our architecture using two different ray-tracing applications: a recursive Whitted-style ray tracer [2]–[4] that allows us to compare directly to other hardware ray-tracing architectures and a path tracer [6], [8] that allows us to explore how the TRaX architecture responds to incoherent secondary rays, arguably the most important types of rays when considering a ray tracer [9].

This paper does not analyze TRaX's ability to handle dynamically changing scenes. We assume that the necessary data structures are updated on the host machine as needed; thus, the performance we measure is for rendering a single frame. We are, however, currently exploring the possibility of dynamic scene updating on the TRaX architecture.

II. Background

Because most applications are using larger and larger models (Greenberg has argued that typical model sizes are doubling annually [10]), and because most applications are demanding increasingly more visual realism, we believe the trends favor ray tracing (either alone or in combination with rasterization for some portions of the rendering process). Following the example of GPUs, we also believe that a special-purpose architecture can be made capable of interactive ray tracing for large geometric models. Such special-purpose hardware has the potential to make interactive ray tracing ubiquitous. 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.

A. GPUs

Graphics processing is an example of a type of computation that can be streamlined in a special-purpose architecture and achieve much higher processing rates than on a general-purpose processor. This is the insight that enabled the GPU revolution in the 1980s [11]–[14]. 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 the recent generation of z-buffer GPUs possible [15], [16]. Current GPUs have up to hundreds of FP units on a single GPU and an aggregate memory bandwidth of 20–80 GB/s from their local memories. That impressive local memory bandwidth is largely to support frame-buffer access and image-based (lookup) textures for the primitives. These combine to achieve graphics performance that is orders of magnitude higher than what 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:

1) **Traversal:** traversing the acceleration structure, which is a spatial index that encapsulates the scene objects to identify a set of objects that the ray is likely to intersect with;
2) **Intersection:** intersecting the ray with the primitive objects contained in the element of the bounding structure that is hit;
3) **Shading:** computing the illumination and color of the pixel based on the intersection with the primitive object and the collective contributions from the secondary ray segments. This can also involve texture lookups or procedural texture generation.

The traversal and intersection 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 traditional z-buffer-based architecture.

While it is possible to perform ray tracing on GPUs [17]–[19], until recently, 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 a general-purpose GPU (GPGPU) architecture is toward more and more programmability of the graphics pipeline. Current high-end GPGPUs, such as the G80 architecture from nVidia [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-type GPGPU assumes that every set of 32 threads (a "warp") essentially executes the same instruction and that they can thus be executed in a single-instruction–multiple-data (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 [19], and a complete ray tracer has been demonstrated by nVidia using a collection of four of their highest performance graphics cards, but little has been published about the demo [21].

B. 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 multicore 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 [22], [23]. The biggest difference in our approach is that we do not depend on the coherence of the ray packet to extract thread-level parallelism. Thus, our hardware should perform well even for diverging secondary rays used in advanced shading effects for which grouping the individual rays into coherent packets may not be easy.

In addition to general multicore chips, direct support for multithreading is becoming much more common and appears even in some commercially released processors such as the Intel Netburst architecture [24], the IBM Power5 architecture [25], and the Sun Niagara [26]. The biggest limiting factor for these general architectures is that the individual processors are heavily underutilized while performing ray tracing. This is due largely to the relatively small number of FP resources on a CPU and the highly branch-dependent behavior of ray-tracing threads. We believe that a larger number of simpler cores will perform better than fewer more complex cores of a general CPU due to providing a more targeted set of computation resources for the application.

The IBM Cell processor [27], [28] is an example of an architecture that might be quite interesting for ray tracing. With a 64-b in-order power processor element core (based on the IBM Power architecture) and eight synergistic processing elements (SPEs), the Cell architecture sits somewhere between a general CPU and a GPU-style chip. Each SPE contains a 128 × 128 register file, 256 kb of local memory (not a cache), and four FP units operating in SIMD. When clocked at 3.2 GHz, the Cell has a peak processing rate of 200 GFlops. 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 four to eight times faster than a single-core x86 CPU [29]. In order to get those speedups, the ray tracer required careful mapping into the scratch memories of the SPEs and management 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 an SIMD fashion and while using considerably less programmer effort because we do not rely on a programmer-managed scratch memory.

C. Ray-Tracing Hardware

Other researchers have developed special-purpose hardware for ray tracing [30], [31]. The most complete of these are the SaarCOR [32], [33] and ray processing unit (RPU) [34], [35] architectures from Saarland University. SaarCOR is a custom hard-coded ray trace processor, and RPU has a custom kd-tree traversal unit with a programmable shader. Both are implemented and demonstrated on a field-programmable gate array. All high-performance ray tracers organize the scene being rendered into an "acceleration structure" of some sort, which permits fast traversal of the scene volume to quickly arrive at the primitive geometry. Common structures are kd-trees, bounding volume hierarchies (BVHs), oct-trees, and grids. The traversal of this structure is done in hardware in the Saarland architectures and requires that a kd-tree be used. Only when a primitive is encountered that the programmable shader is called to determine the color of that primitive (and, thus, the color of the pixel).

The programmable portion of the RPU is known as the shading processor and is used to determine the shading (color) of each pixel once the intersected triangle primitive is determined. This portion consists of four four-way vector cores running in SIMD mode with 32 hardware threads supported on each of the cores. Three caches are used for shader data, kd-tree data, and primitive (triangle) data. Cache coherence is quite good for primary rays (initial rays from the eye to the scene) and adequate for secondary rays (shadows, reflections, etc.). With an appropriately described scene (using kd-trees and triangle data encoded with unit-triangle transformations), the RPU can achieve very impressive frame rates, particularly when extrapolated to a potential CMOS application-specific integrated circuit implementation [35].

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

Most recently, there have been proposals for multicore systems based on simplified versions of existing instruction set architectures that may be useful for ray tracing. These approaches are closest in spirit to our architecture and represent work that is concurrent with ours; thus, detailed comparisons are not yet possible. Both of these projects involve multiple simplified in-order cores with small-way multithreading, and both explicitly evaluate ray tracing as workload. The Copernicus approach [36] attempts to leverage existing general-purpose cores in a multicore organization rather than developing a specialized core specifically for ray tracing. As a result, it requires more hardware to achieve the same performance and will not exceed 100 million rays per second unless scaling to 115 cores at a 22-nm process. A commercial approach, called Larrabee [37], is clearly intended for general-purpose computing and rasterizing graphics, as well as ray tracing, and makes heavy use of SIMD in order to gain performance. Because it is intended as a more general-purpose processor, Larrabee also includes coherency between levels of its caches, something which TRaX avoids because of its more specialized target. This coherency is accomplished using a ring network that communicates between local caches, which adds complexity to the architecture.

D. Target Applications

There are several applications, such as movies, architecture, and manufacturing, that rely on image quality and need shadows and reflections. These applications already use batch ray tracing but would benefit greatly from interactive ray tracing.

Other applications are not currently close to being interactive on GPUs regardless of image quality because their number of primitive objects N is very large. These include many scientific simulations [38], the display of scanned data [39], and terrain rendering [40]. While level-of-detail techniques can sometimes make display of geometrically simplified data possible, such procedures typically require costly preprocessing and can create visual errors [41].

Simulation and games demand interactivity and currently use z-buffer hardware almost exclusively. However, they spend a great deal of computational effort and programmer time creating complicated procedures for simulating lighting effects and reducing N by model simplification. In the end, they have imagery of inferior quality to that generated by ray tracing. We believe that those industries would use ray tracing if it was fast enough.

We have customized the hardware in our simulator to perform well for ray tracing, which is our primary motivation. While TRaX is programmable and could be used for other applications, we have not explored TRaX's performance for a more robust range of applications. There are certainly other multithreaded applications that might perform very well. However, one major restriction on other applications running on TRaX is the (intentional) lack of coherence between the caches on the chip, which would hinder applications with substantial communication between threads.

III. TRaX Architecture

Threads represent the smallest division of work in a ray-traced scene; thus, the performance of the entire system depends on the ability of the architecture to flexibly and efficiently allocate functional resources to the executing threads. As such, our architecture consists of a set of TPs that include some FUs in each processor and that share other larger FUs between TPs. A collection of these TPs, their shared FUs, issue logic, and shared L2 cache are collected into a "core."

A full chip consists of many cores, each containing many TPs, sharing an on-chip L2 cache and off-chip memory and I/O bandwidth. Because of the parallel nature of ray tracing, threads (and, thus, cores) have no need to communicate with each other except to atomically divide the scene. Therefore, a full on-chip network is neither provided nor needed. In order to support multichip configurations, off-chip bandwidth is organized into lanes, which can be flexibly allocated between external memory and other I/O needs.

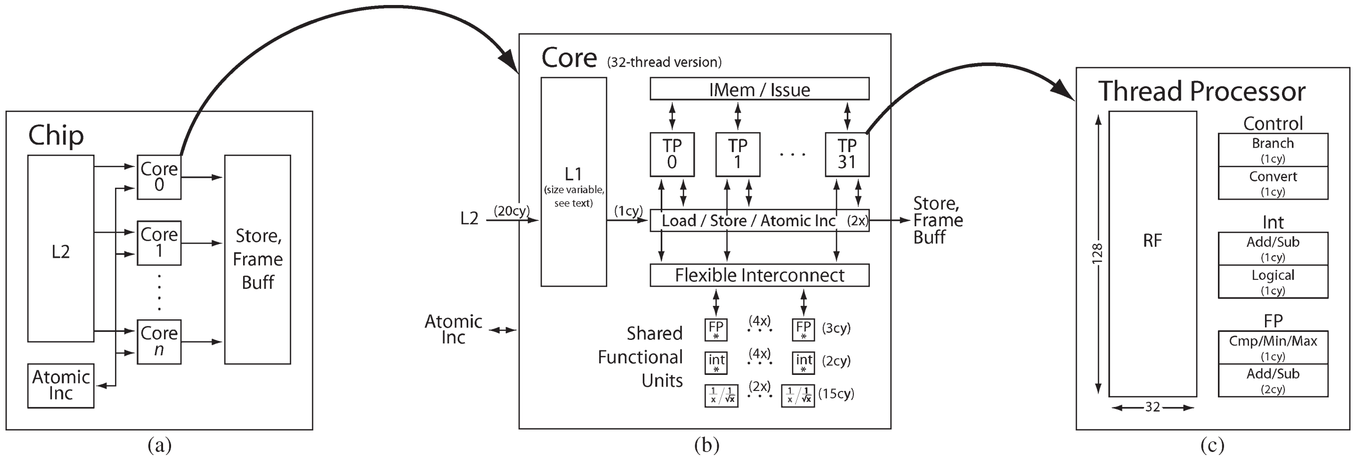

Fig. 3. (a) Multicore chip layout. (b) Core block diagram. (c) TP state.

A. TP

Each TP in a TRaX core can execute its own thread code, where a software thread corresponds to a ray. Each thread maintains a private program counter, a register file, and a small instruction cache. The register file is a simple two-read one-write SRAM block. Because of the complexity involved in forwarding data between FUs, all results are written back to the register file before they can be accessed by the consuming instruction. Fig. 3(c) shows these FUs as well as the register file. The type and number of these FUs are variable in our simulator. More complex FUs are shared by the TPs in a core.

Instructions are issued in order in each TP to reduce the complexity at the thread level. The execution is pipelined with the fetch and decode, each taking one cycle. The execution phase requires a variable number of cycles, depending on the FU required, and the writeback takes a final cycle. Instructions issue in order but may complete out of order. Thread processing stalls primarily if the needed data are not yet available in the register file (using a simple scoreboard) or if the desired FU is not available, but correct single-thread execution is guaranteed.

Because issue logic is external to the thread state (implemented at the core level), there is very little complexity in terms of dependence checking internal to each thread. A simple table maintains instructions and their dependences. Instructions enter the table in a first-in–first-out fashion, in program order, so that the oldest instruction is always the next available instruction. Issue logic checks only the status of this oldest instruction. Single-thread performance is heavily dependent on the programmer/compiler who must order instructions intelligently to hide FU latencies as often as possible.

B. Collection of Threads in a Core

Each of the multiple cores on a chip consists of a set of simple TPs with shared L1 data cache and shared FUs, as shown in Fig. 3(b). Each TP logically maintains a private L1 instruction cache (more accurately, a small set of TPs shares a multibanked I-cache). However, all threads in a core share one multibanked L1 data cache of a modest size (2048 lines of 16 B each, directly mapped, and with four banks; see Section VI-A). All cores on a multicore chip share an L2 unified instruction and data cache. Graphics processing is unique in that large blocks of memory are either read-only (e.g., scene data) or write-only (e.g., the frame buffer). To preserve the utility of the cache, write-once data are written around the cache. For our current ray-tracing benchmarks, no write data need to be read back; thus, all writes are implemented to write around the cache (directly to the frame buffer). Separate cached and noncached write assembly instructions are provided to give the programmer control over which kind of write should occur. This significantly decreases thrashing in the cache by filtering out the largest source of pollution. Hence, cache hit rates are high, and threads spend fewer cycles waiting on return data from the memory subsystem. In the future, we plan to explore using read-around and streaming techniques for certain types of data that are known to be touch-once. Currently, the choice of read-around or write-around versus normal cached memory access is made by the programmer.

Each shared FU 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 FUs, 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 FUs in a round-robin fashion.

Each TP 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 multilevel 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. Our results show that, with 32 TPs per core, close to 50% of the threads can issue on average in every cycle for a variety of different scenes using an assembly-coded Whitted-style ray tracer [7] and a path tracer coded in a C-like language [9].

TRaX is specifically designed to accelerate single-ray performance and to exploit thread-level parallelism using multiple

TPs and cores. Many other ray-tracing architectures [19], [29], [33], [34], [36], [37], [42], [43] exploit parallelism using SIMD to execute some number of the same instructions at the same time. This technique does not scale well if the rays in the SIMD bundle become less coherent with respect to the scene objects they intersect [9]. In that case, what was a single SIMD instruction will have to be repeated for each of the threads as they branch to different portions of the scene and require different intersection tests and shading operations. Because our threads are independent, we do not have to mask off results of our FU operations.

C. Multicore Chip

Our overall chip design [Fig. 3(a)] is a die consisting of an L2 cache with an interface to off-chip memory and a number of repeated identical cores with multiple TPs 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.

The L2 cache is assumed to be banked similarly to the L1 cache to allow parallel accesses from the L1 caches of the many cores on the chip. A number of miss status holding registers (MSHRs) are provided per core, and both the number banks and number of MSHRs are parameterized in our simulations. It should be noted that, for the work reported in this paper, the L2 cache was not modeled explicitly (see Section V for more details). Instead, all misses in the L1 cache were treated as a fixed latency to memory intended to approximate the average L2 latency. The modeled latency to L2 was on the order of 20 times the latency of L1 hits. Ongoing simulations have added explicit models for the L2 cache and DRAM, but those numbers are not all available yet. We are finding that our original assumptions are not too far off though.

IV. RAY-TRACING APPLICATIONS

Some of our test programs are written directly in an assembly language. Others are written in a higher level language designed for our architecture. The TRaX programming language is a simple C-like language with some extensions inspired by the RenderMan shading language [44] to allow for ease of writing a ray-tracing application. The language is compiled into TRaX assembly for the simulator by our simple custom compiler.

To evaluate our architecture, we have developed two different ray-tracing systems.

- **Whitted-Style Ray Tracer.** This implements a recursive ray tracer that provides various shading methods, shadows from a single point light source, and BVH traversal. It is written in TP assembly language.
- **Path Tracer.** This application is written in TRaX language described previously. It computes global illumination in the scene using a single point light source and using Monte Carlo-sampled Lambertian shading [4].

The test scenes we are using, listed in Table I with some basic performance numbers, exhibit some important properties.

TABLE I
SCENE DATA WITH RESULTS FOR 1 AND 16 CORES, EACH WITH 32 TPs, AND PHONG SHADING ESTIMATED AT 500 MHz

Scene	Triangles	BVH Nodes	FPS (1)	FPS (16)
conference	282664	266089	1.4282	22.852
sponza	66454	58807	1.1193	17.9088
cornell	32	33	4.6258	74.012

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 65 000 triangles and uses a BVH with over 50 000 nodes. The Conference Room scene is an example of a reasonably large and complex scene with around 300 000 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.

A. Whitted-Style Ray Tracer

This is a basic recursive ray tracer that provides us with a baseline that is easily compared to other published results. In addition to controlling the depth and type of secondary rays, another parameter that can be varied to change its performance is the size of the tiles assigned to each thread to render at one time. Originally, the screen would be split into 16 $\times$ 16 pixel squares, and each thread would be assigned one tile to render. While this is a good idea for load balancing among the threads, we found that it did not produce the best performance. Instead, we changed the tiles to be single pixels and assigned those to threads in order. This seemingly minor change was able to increase the coherence of consecutive primary rays (putting them closer together in screen space) and make the cache hit rate much higher. The increased coherence causes consecutive rays to hit much of the same scene data that have already been cached by recent previous rays, as opposed to each thread caching and working on a separate part of the scene.

Currently, the pixels are computed row by row straight across the image. As we advance the ray tracer further, we will use a more sophisticated space filling method such as a Z curve. This method will trace rays in a pattern that causes concurrent rays to stay clustered closer together, which makes them more likely to hit the same nodes of the BVH, increasing cache hit rate.

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 [45], [46]. We also include simple hard shadows from a 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 particularly

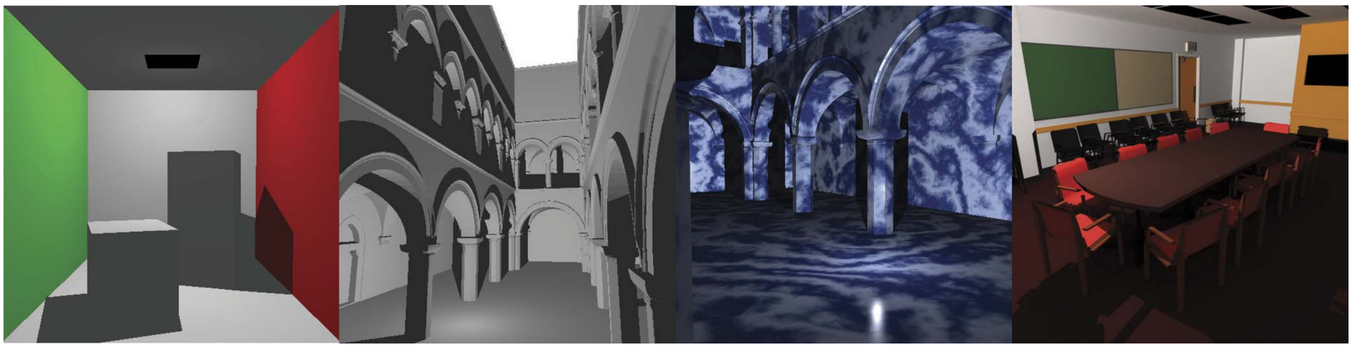
Fig. 4. Test scenes rendered on our TRaX architectural simulator. (From left to right) Cornell (rendered with our Whitted-style ray tracer), Sponza (rendered with our Whitted-style ray tracer), Sponza (rendered with our Whitted-style ray tracer with procedural textures), and Conference (rendered with our path tracer). These are standard benchmarking scenes for ray tracing.

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.

2) Procedural Texturing: We also implement procedural textures, i.e., 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 [47], [48]. These textures are computed using pseudorandom 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 precomputed pseudorandom 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 [47]) or a quintic interpolant (for improved Perlin noise [49]) to produce the final value. More complex pattern functions, such as turbulence produced through spectral synthesis, sum the multiple evaluations of Perlin noise for each point shaded. There are 672 FP operations in our code to generate the texture at each pixel. We ran several simulations, comparing the instruction counts of an image with and without noise textures. We found that there are, on average, 50% 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 bandwidths. Conventional GPUs require an extremely fast memory bus and a very large amount of RAM for storing textures [15], [16]. Some researchers believe that, if noise-based procedural textures were well supported and efficient, many applications, specifically video games, would choose those textures over the memory-intensive image-based textures that are used today [50]. An example of a view of the Sponza scene rendered with our Perlin noise-based textures can be seen in Fig. 4.

B. Path Tracer Application

In order to explore the ability of our architecture to maintain performance in the face of incoherent rays that do not respond well to packets, we built a path tracer designed so that we could carefully control the coherence of the secondary rays. Our path tracer is written in the TRaX language described previously and is designed to eliminate as many variables as possible that could change coherence. We use a single point light source and limit incoherence to Monte Carlo-sampled Lambertian shading with no reflective or refractive materials [4]. Every ray path is traced to the same depth: There is no Russian roulette or any other dynamic decision making that could change the number of rays cast. This is all to ensure that we can reliably control secondary ray coherence for these experiments. A more fully functional path tracer with these additional techniques could be written using the TRaX programming language, and we expect it would have similar performance characteristics.

Each sample of each pixel is controlled by a simple loop. The loop runs D times, where D is the specified max depth. For each level of depth, we cast a ray into the scene to determine the geometry that was hit. From there, we cast a single shadow ray toward the point light source to determine if that point receives illumination. If so, this ray contributes light based on the material color of the geometry and the color of the light. As this continues, light is accumulated into the final pixel color for subsequent depth levels. The primary ray direction is determined by the camera, based on which pixel we are gathering light for. Secondary (lower depth) rays are cast from the previous hit point and are randomly sampled over a cosine-weighted hemisphere, which causes incoherence for higher ray depths.

Secondary rays are randomly distributed over the hemisphere according to a bidirectional reflectance distribution function

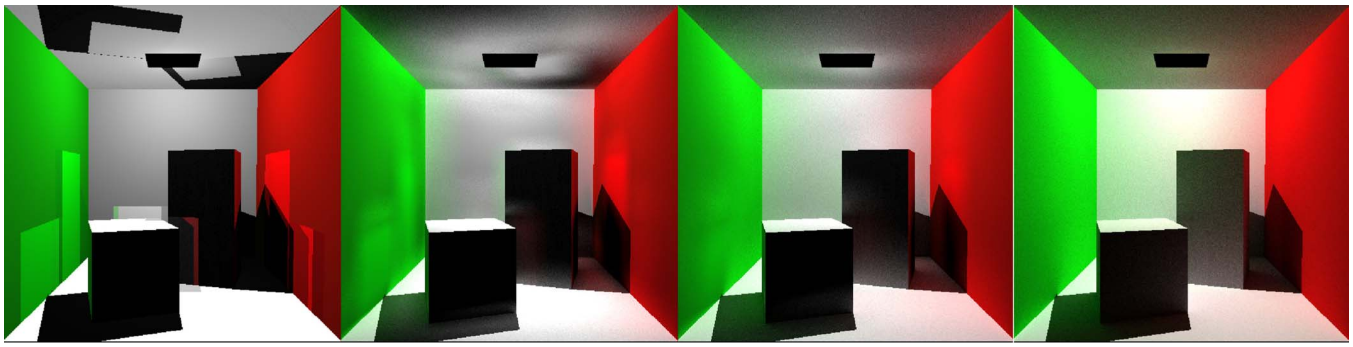
Fig. 5. Cornell box scene showing the visual change as the sampling angle increases in our path tracer. (Starting from left) 0°, 30°, 60°, and 180° on the right.

(BRDF) [51], [52]). To compute a cosine-weighted Lambertian BRDF, a random sample is taken on the area of a cone with the major axis of the cone parallel to the normal of the hit geometry and the vertex at the hit point. As an artificial benchmark, we limit the angle of this cone anywhere from 0° (the sample is always taken in the exact direction of the normal) to 180° (correct Lambertian shading on a full hemisphere). By controlling the angle of the cone, we can control the incoherence of the secondary rays. The wider the cone angles, the less coherent the secondary rays become, as they are sampled from a larger set of possible directions. The effect of this can be seen in Fig. 5.

V. Design Exploration

We have two TRaX simulators: a functional simulator that executes TRaX instructions by running them on the PC and a cycle-accurate simulator that simulates, in detail, the execution of a single core with 32 threads and associated shared FUs. The functional simulator executes much more quickly and is very useful for debugging applications and generating images.

The cycle-accurate simulator runs much more slowly than the functional simulator and is used for all performance results. Given the unique nature of our architecture, it was not reasonable to adapt available simulators to our needs. In the style of Simplescalar [53], our cycle-accurate simulator allows for extensive customization and extension. Memory operations go through the L1 cache and to the L2 with conservative latencies and variable banking strategies.

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 do not 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. Statistics provided by the simulator include total cycles used to generate a scene, FU utilization, thread utilization, thread stall behavior, memory and cache bandwidth, memory and cache usage patterns, and total parallel speedup.

Our ray-tracing code was executed on simulated TRaX cores having between 1 and 256 TPs, with issue widths of all function units except memory, varying between 1 and 64 (memory was held constant at a single issue). Images may be generated for any desired screen size. Our primary goal for the current design phase is to determine the optimal allocation of transistors to thread-level resources, including FUs 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).

TABLE II
Default FU Mix (500-MHz 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

A. FUs

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 FUs with a simple but powerful instruction set architecture. We include bitwise instructions; branching; FP/integer conversion; memory operations; FP and integer add, subtract, multiply, reciprocal; and FP compare. We also include reciprocal square root because that operation occurs with some frequency in graphics code for normalizing vectors.

FUs are added to the simulator in a modular fashion, allowing us to support arbitrary combinations and types of FUs and instructions. This allows very general architectural exploration starting from our basic thread-parallel execution model. We assume a conservative 500-MHz clock which was chosen based on the latencies of the FUs that were synthesized using Synopsys Design Compiler and DesignWare libraries [54] and well-characterized commercial CMOS cell libraries from Artisan [55]. Custom-designed function units such as those used in commercial GPUs would allow this clock rate to be increased.

We first chose a set of FUs to include in our machine-level language, shown in Table II. This mix was chosen by separating different instruction classes into separate dedicated FUs. We implemented our ray casting benchmarks using these available resources and then ran numerous simulations, varying

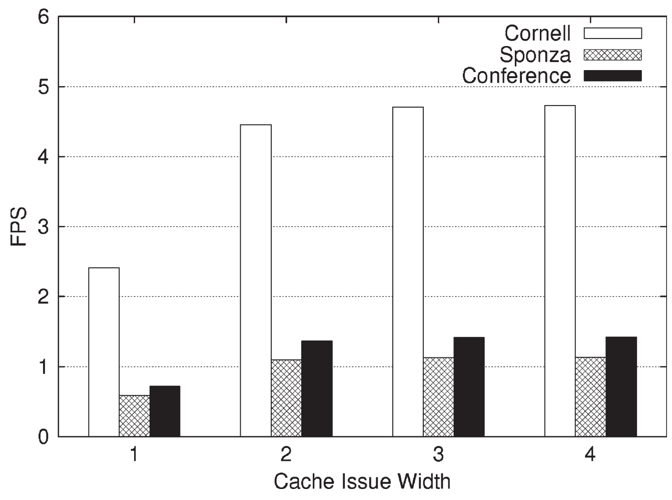

Fig. 6. Single-core performance as cache issue width is varied.

the number of threads and the width of each FU. 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 as cross and dot products, both of which require multiple FP multiplies and 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 FU 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. We are also exploring the possibility of including a custom noise function as a shared FU that would allow the rapid generation of gradient noise used for procedural texturing (see Section IV-A2).

VI. Results

Results are generated for a variety of TP configurations and using both our Whitted-style ray tracer and path tracer.

A. Single-Core Performance (Fig. 6)

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 multicore 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 the repositioning of objects, light sources, and viewpoints. The results shown here describe a preliminary analysis based on simulation.

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

TABLE III
Area Estimates (Prelayout) for FUs Using Artisan CMOS Libraries and Synopsys. The 130-nm Library Is a High-Performance Cell Library, and the 65-nm Library Is a Low-Power Cell Library. Speed Is Similar in Both Libraries

Resource Name	Area (μm^2) 130nm	65nm
2k×16byte cache (four banks / read ports)	1,527,5719	804,063
128×32 RF (1 Write 2 Read ports)	77,533	22,000(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

TABLE IV
Core Area Estimates to Achieve 30 fps on Conference. These Estimates Include the Multiple Cores, as Shown in Figs. 3(a) and (b), 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 V
Performance Comparison for Conference and Sponza Assuming a Fixed Chip Area of 150 mm^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

Basic FUs, including register files and caches, were synthesized, placed, and routed using Synopsys and Cadence tools to generate estimated sizes (Table III). 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 FUs 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 Tables IV and V are limited to 150 mm^2 in order to allow room for the components that are currently unaccounted for.

2) Performance: For a ray tracer to be considered to achieve real-time performance, it must have a frame rate of around 30 fps. The TRaX architecture is able to render the conference scene at 31.9 fps with 22 cores on a single chip at 130 nm. At 65 nm with 79 cores on a single chip, performance jumps to 112.3 fps.

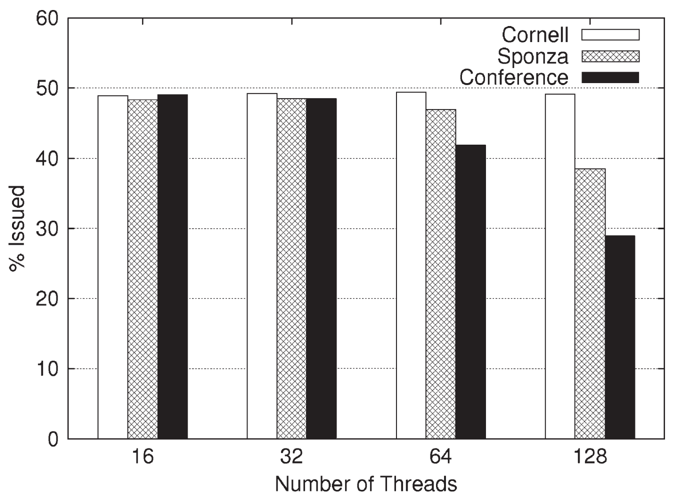

Fig. 7. Thread performance (percent issued).

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. Fig. 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 130-nm process, we fit 16 to 22 cores on a chip. Even at the low end, the number of instructions issued in each cycle can reach up to 256. With a die that shrinks 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 instruction per cycle (IPC) of more than 1024. Even recent GPUs with many concurrent threads issue a theoretical maximum IPC of around 256 (128 threads issuing two FP operations per cycle).

Another indicator of sustained performance is the average utilization of the shared FUs. 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. While a detailed study of power consumption was not performed in this paper, we expect the power consumption of TRaX to be similar to that of commercial GPUs.

3) Cache Performance: We varied data cache size and issue width to determine an appropriate configuration offering high performance balanced with reasonable area and complexity. For scenes with high complexity, a cache with at least 2048 lines (16 B each) satisfied the data needs of all 32 threads executing in parallel with hit rates in the 95% range. We attribute much of this performance to low cache pollution because all stores go around the cache. Although performance continued to increase slightly with larger cache sizes, the extra area required to implement the larger cache meant that the total silicon needed to achieve 30 fps actually increased beyond a 2-kB L1 data cache size. To evaluate the number of read ports needed, we simulated a large (64 kB) cache with between 1 and 32 read ports. Three read ports provided sufficient parallelism for 32 threads. This is implemented as a four-bank directly mapped cache.

The L2 cache was not modeled directly in these experiments. Instead, a fixed latency of 20 cycles was used to conservatively estimate the effect of the L2 cache. Ongoing simulations include detailed L2 and DRAM models, where it appears that a 512-kB L2 cache shows good hit rates. Although those simulations are not complete, initial indications are that our estimate was, in fact, conservative. The ongoing simulations are currently showing memory bandwidths between L1 cache and the register file that ranges from 10–100 GB/s, depending on the size of the scene. The L2–L1 bandwith ranges from 4–50 GB/s and DRAM–L2 from 250 Mb/s to 6 GB/s for reads. These clearly cover a broad range, depending on the size and complexity of the scene, and we are currently running additional simulations to better understand the memory system.

The I-caches are modeled as 8-kB directly mapped caches, but because the code size of our current applications is small enough to fit in those caches, we assume they are fully warmed and that all instruction references come from those caches. The ongoing more detailed simulations do not make this assumption, but because of the current code size, there are few impacts of L1 I-cache on processing times.

4) Comparison: Comparing against the Saarland RPU [34], [35], 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 [29] 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 500 MHz compared to the Cell performance at 3.2 GHz. Table VI shows these comparisons.

B. Secondary Ray Performance

We call the initial rays that are cast from the eye point into the scene to determine “visibility rays” (sometimes, these are called “primary rays”) and all other rays that are recursively cast from that first intersection point “secondary rays.” This is something of a misnomer, however, because it is these secondary rays, used to compute optical effects, that differentiate ray-traced images from images computed using a z-buffer. The secondary rays are not less important than the visibility rays. They are, in fact, the essential rays that enable the highly realistic images that are the hallmark of ray tracing. We believe that any

TABLE VI
PERFORMANCE COMPARISON FOR CONFERENCE AGAINST CELL AND RPU. COMPARISON IN FRAMES PER SECOND AND MRPS. ALL NUMBERS ARE FOR SHADING WITH SHADOWS. TRaX AND RPU NUMBERS ARE FOR 1024 × 768 IMAGES. CELL NUMBERS ARE FOR 1024 × 1024 IMAGES, AND THUS, THE CELL IS BEST COMPARED USING THE MRPS METRIC WHICH FACTORS OUT IMAGE SIZE

	TRaX		IBM Cell [29]		RPU [35]	
	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	≈ 200	≈ 440	≈ 200	≈ 190
Clock	500MHz	500MHz	3.2GHz	3.2GHz	266MHz	400MHz

specialized hardware for ray tracing must be evaluated for its ability to deal with these all-important secondary rays.

A common approach to accelerating visibility rays is to use "packets" of rays to amortize cost across sets of rays [23], [56], [57]. However, secondary rays often lose the coherency that makes packets effective and performance suffers on the image as a whole. Thus, an architecture that accelerates individual ray performance without relying on packets could have a distinct advantage when many secondary rays are desired.

To study this effect, we use our path tracer application, which we have designed so that we can control the degree of incoherence in the secondary rays (see Section IV-B). We do this by controlling the sampling cone angle for the cosine-weighted Lambertian BRDF used to cast secondary rays.

We compare our path tracer to Manta, which is a well-studied packet-based ray/path tracer [57]. Manta uses packets for all levels of secondary rays, unlike some common ray tracers that only use packets on primary rays. The packets in Manta shrink in size as ray depth increases, since some of the rays in the packet become uninteresting. We modified Manta's path tracing mode to sample secondary rays using the same cone angles as in our TRaX path tracer, so that comparisons could be made.

Manta starts with a packet of 64 rays. At the primary level, these rays will be fairly coherent as they come from a common origin (the camera) and rays next to each other in pixel space have a similar direction. Manta intersects all of the rays in the packet with the scene BVH using the DynBVH algorithm [22]. It then repartitions the ray packet in memory based on which rays hit and which do not. DynBVH relies on coherence with a frustum-based intersection algorithm and by using Streaming SIMD Extension (SSE) instructions in groups of four for ray–triangle intersection tests. If rays in the packet remain coherent, then these packets will stay together through the BVH traversal and take advantage of SSE instructions and frustum-culling operations. However, as rays in the packet become incoherent, they will very quickly break apart, and almost every ray will be traversed independently.

To test how our path tracer performs relative to the level of coherence of secondary rays, we ran many simulations, incrementally increasing the angle of our sampling cone and measuring rays per second and speedup (slowdown) as the angle was increased and secondary rays become less coherent. For all of our tests, we used a ray depth of three (one primary ray and two secondary rays). We believe that three rays taken randomly on a hemisphere are sufficient for complete incoherence and will allow secondary rays to bounce to any part of the scene data. This will cause successive rays to have a widely ranging origin and direction, and packets will become very incoherent.

With a cone angle close to 0°, secondary rays will be limited to bouncing close to the normal, which will force rays to a limited area of the scene. In a packet-based system using a narrow cone angle, successive samples will have a much higher probability of hitting the same BVH nodes as other samples in the packet, allowing for multiple rays to be traced at the same time with SIMD instructions. Increasing the angle of the cone will decrease this probability, allowing for fewer, if any, SIMD advantages. With a cone angle of 180°, a packet of secondary rays will be completely incoherent, and the probability of multiple rays hitting the same primitives is very slim. We used the same cone angle sampling scheme in Manta and tested TRaX versus Manta on common benchmark scenes to show the degrees of slowdown that each path tracer suffers as rays become incoherent.

As explained before, we used a fixed ray depth of three. We varied the size of the image and the number of samples per pixel and gathered data for the number of rays per second for each test for both path tracers. For TRaX, we also recorded L1 cache hit rates and thread-issue rates within the single core that was simulated. The images themselves can be seen in Fig. 4, with data about the images shown in Table I.

Our primary interest is the speed for each path tracer relative to itself as the cone angle is modified. The results are shown in Table VII. We show that, as the secondary rays become incoherent, the TRaX architecture slows to between 97% and 99% of the speed with a narrow cone angle. On the other hand, the Manta path tracer on the same scene with the same cone angles slows to between 47% to 53% of its speed on the narrow angle cone. We believe that this validates our approach of accelerating single-ray performance without relying on packets and SIMD instructions.

In addition to showing that the TRaX architecture maintains performance better than a packet-based path tracer in the face of incoherent secondary rays, we need to verify that this is not simply due to TRaX being slow overall. Thus, we also measure millions of rays per second (MRPS) in each of the path tracers. The Manta measurements are made by running the code on one core of an Intel Core2 Duo machine running at 2.0 GHz. The TRaX numbers are from our cycle-accurate simulator assuming a 500-MHz speed and using just a single core with 32 TPs. We expect these numbers to scale very well, as we tile multiple cores on a single die. As mentioned in Section III, chips with between 22 to 78 cores per die would not be unreasonable.

TABLE VII

RESULTS ARE REPORTED FOR THE CONFERENCE AND SPONZA SCENES AT TWO DIFFERENT RESOLUTIONS WITH A DIFFERENT NUMBER OF RAYS PER PIXEL. PATH TRACED IMAGES USE A FIXED RAY DEPTH OF THREE. TRAX RESULTS ARE FOR A SINGLE CORE WITH 32 TPS RUNNING AT A SIMULATED 500 MHZ. WE EXPECT THESE NUMBERS TO SCALE WELL AS THE NUMBER OF TRAX CORES IS INCREASED. MANTA NUMBERS ARE MEASURED BY RUNNING ON A SINGLE CORE OF AN INTEL CORE2 DUO AT 2.0 GHZ. SPEED RESULTS ARE NORMALIZED TO PATH TRACING WITH A 10° CONE

Conference: 256×256 with 4 samples per pixel					
	ray casting only	10 degrees	60 degrees	120 degrees	180 degrees
Manta MRPS	1.61	0.8625	0.5394	0.4487	0.4096
Manta speed	1.87	1	0.63	0.52	0.47
TRaX MRPS	1.37	1.41	1.43	1.43	1.40
TRaX speed	.97	1	1.01	1.01	0.99
Cache hit %	88.9	85.1	83.9	83.5	83.2
Thread issue %	52.4	52.4	52.5	52.5	52.4

Sponza: 128×128 with 10 samples per pixel					
	ray casting only	10 degrees	60 degrees	120 degrees	180 degrees
Manta MRPS	1.391	0.7032	0.4406	0.3829	0.3712
Manta speed	1.98	1	0.63	0.54	0.53
TRaX MRPS	0.98	1.01	0.98	0.97	0.98
TRaX speed	0.97	1	0.97	0.96	0.97
Cache hit %	81.5	77.4	76.3	76.0	76.0
Thread issue %	50.6	50.9	50.9	50.7	50.9

In order to show why TRaX slows down as it does, we also include the cache hit rate from our simulator and the average percentage of total threads issuing per cycle in Table VII. As the cone angle increases, rays are allowed to bounce with a wider area of possible directions, thus hitting a larger range of the scene data. With a smaller cone angle, subsequent rays are likely to hit the same limited number of triangles, allowing them to stay cached. As more threads are required to stall due to cache misses, we see fewer threads issuing per cycle. This is a smaller thread-issue percentage than we saw in previous work [7], which indicates that smaller cores (cores with fewer TPs) may be interesting for path tracing.

Because of the time required to run a cycle-accurate simulation, the results from this paper are restricted to relatively low resolution and ray depth. However, if we consider the effect of dynamic ray depth computations on an average scene, rays often lose enough energy to be cut off on or before three bounces, particularly if Russian roulette is employed. If deeper ray depths are required, this would likely have the effect of improving the TRaX advantage over a packet-based path tracer like Manta, as the percentage of incoherent rays would increase (the primary rays would be a smaller percentage of the total rays cast).

VII. CONCLUSION

We have shown that a simple, yet powerful, multicore multithreaded architecture can perform real-time ray tracing running at modest clock speeds on achievable technology. By exploiting the coherence among 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 [58]. We will also improve the memory system in the simulator to more accurately simulate L2 cache performance.

It is, of course, not completely clear yet that our non-SIMD approach is superior to an SIMD approach. The main overhead of a non-SIMD core is the replication of the I-cache and decode logic. We are currently exploring the sharing of a multibanked I-cache among a number of TPs to amortize this overhead. However, the size of the I-caches is small compared to the D-caches and the FUs; thus, we believe that the general overhead of including more I-caches for a non-SIMD approach will be fairly small. More importantly, the performance advantage on noncoherent secondary rays seems to be large, and TRaX seems to scale well for these very important rays.

In order to explore whether our TRaX architecture performs well with incoherent secondary rays, we have implemented a path tracer with an artificially narrowed Lambertian BRDF benchmark as a simple way to quantify ray coherence. We have found that TRaX has only a minor slowdown of 97% to 99% of top speed on our test scenes when the secondary rays become highly incoherent. Manta slowed down to 47% to 53% of top speed on the same scenes with the same mechanism for controlling coherency. We attribute the difference to the overhead of dealing with small packets and the breakdown of the SIMD operation as the packets become highly incoherent.

We are in the process of improving our ray-tracing applications to drive architectural exploration further. The goal is to allow for Cook style ray tracing [59] with support for multisampling. We will also add support for image-based textures as a comparison against procedural textures and explore hardware support for gradient noise used in 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 procedural textures [50]. Procedural textures using, for example, Perlin noise techniques [47], [48] increase FP operations by about 50% in the worst case but have a negligible impact on memory bandwidth. This can have a positive impact on 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.

Acknowledgment

The authors would like to thank the other current and former members of the Hardware Ray Tracing group: S. Boulos, A. Davis, S. Kellis, S. Parker, K. Ramani, and P. Shirley.

References

[1] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," Ph.D. dissertation, Univ. Utah, Salt Lake City, UT, Dec. 1974.

[2] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[3] A. Glassner, Ed., *An Introduction to Ray Tracing*. London, U.K.: Academic, 1989.

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

[5] D. S. Immel, M. F. Cohen, and D. P. Greenberg, "A radiosity method for non-diffuse environments," in *Proc. SIGGRAPH*, 1986, pp. 133–142.

[6] J. T. Kajiya, "The rendering equation," in *Proc. SIGGRAPH*, 1986, pp. 143–150.

[7] J. Spjut, D. Kopta, S. Boulos, S. Kellis, and E. Brunvand, "TRaX: A multi-threaded architecture for real-time ray tracing," in *Proc. 6th IEEE SASP*, Jun. 2008, pp. 108–114.

[8] E. Lafortune and Y. D. Willems, "Bi-directional path-tracing," in *Proc. Compugraphics*, Alvor, Portugal, Dec. 1993, pp. 145–153.

[9] D. Kopta, J. Spjut, E. Brunvand, and S. Parker, "Comparing incoherent ray performance of TRaX vs. Manta," in *Proc. IEEE Symp. Interactive Ray Tracing (RT)*, Aug. 2008, p. 183.

[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 *Proc. SIGGRAPH*, 1997, pp. 477–494.

[11] J. H. Clark, "The geometry engine: A VLSI geometry system for graphics," in *Proc. 9th Annu. Conf. SIGGRAPH*, 1982, pp. 127–133.

[12] J. Poulton, H. Fuchs, J. D. Austin, J. G. Eyles, J. Heineche, C. Hsieh, J. Goldfeather, J. P. Hultquist, and S. Spach, "PIXEL-PLANES: Building a VLSI based raster graphics system," in *Proc. Chapel Hill Conf. VLSI*, 1985, pp. 135–160.

[13] H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spach, J. D. Austin, J. Frederick, P. Brooks, J. G. Eyles, and J. Poulton, "Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes," in *Proc. 12th Annu. Conf. SIGGRAPH*, 1985, pp. 111–120.

[14] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: A VLSI system for high performance graphics," in *Proc. 15th Annu. Conf. SIGGRAPH*, 1988, pp. 21–30.

[15] ATI, *ATI Products From AMD*. [Online]. Available: http://ati.amd.com/products/index.html

[16] nVidia Corporation. [Online]. Available: www.nvidia.com

[17] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 703–712, 2002.

[18] D. Balciunas, L. Dulley, and M. Zuffo, "GPU-assisted ray casting acceleration for visualization of large scene data sets," in *Proc. IEEE Symp. Interactive Ray Tracing (RT)*, Sep. 2006.

[19] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *Proc. IEEE/Eurographics Symp. Interactive Ray Tracing (RT)*, Sep. 2007, pp. 113–118.

[20] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar./Apr. 2008.

[21] *nVidia SIGGRAPH Ray Tracing Demo*, Aug. 2008. [Online]. Available: http://developer.nvidia.com/object/nvision08-IRT.html

[22] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Trans. Graph.*, vol. 26, no. 1, 2007.

[23] 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. Graph. Interface*, May 2007, pp. 177–184.

[24] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, Mar./Apr. 2003.

[25] R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 chip: A dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, Mar./Apr. 2004.

[26] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar./Apr. 2005.

[27] IBM, *The Cell Project at IBM Research*. [Online]. Available: http://www.research.ibm.com/cell

[28] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *Proc. 11th Int. Symp. HPCA*, 2005, pp. 258–262.

[29] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, "Ray tracing on the CELL processor," in *Proc. IEEE Symp. Interactive Ray Tracing (RT)*, Sep. 2006, pp. 15–23.

[30] H. Kobayashi, K. Suzuki, K. Sano, and N. Oba, "Interactive ray-tracing on the 3DCGiRAM architecture," in *Proc. 35th ACM/IEEE MICRO*, 2002.

[31] D. Hall, "The AR350: Today's ray trace rendering processor," in *Proc. EUROGRAPHICS/SIGGRAPH Workshop Graph. Hardware—Hot 3D Session*, 2001, pp. 13–19.

[32] J. Schmittler, I. Wald, and P. Slusallek, "SaarCOR—A hardware architecture for realtime ray-tracing," in *Proc. EUROGRAPHICS Workshop Graph. Hardware*, 2002, pp. 27–36. [Online]. Available: http://graphics.cs.uni-sb.de/Publications

[33] J. Schmittler, S. Woop, D. Wagner, P. Slusallek, and W. J. Paul, "Real-time ray tracing of dynamic scenes on an FPGA chip," in *Proc. Graph. Hardware*, 2004, pp. 95–106.

[34] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A programmable ray processing unit for realtime ray tracing," in *Proc. Int. Conf. Comput. Graph. Interactive Tech.*, 2005, pp. 434–444.

[35] S. Woop, E. Brunvand, and P. Slusallek, "Estimating performance of an ray tracing ASIC design," in *Proc. IEEE Symp. Interactive Ray Tracing (RT)*, Sep. 2006.

[36] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a multicore architecture for real-time ray-tracing," in *Proc. IEEE/ACM Int. Conf. Microarchitecture*, Oct. 2008, pp. 176–187.

[37] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, Aug. 2008.

[38] E. Reinhard, C. Hansen, and S. Parker, "Interactive ray tracing of time varying data," in *Proc. Eurographics Workshop Parallel Graph. Vis.*, 2002, pp. 77–82.

[39] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive ray tracing for isosurface rendering," in *Proc. IEEE Vis.*, 1998, pp. 233–238.

[40] W. Martin, P. Shirley, S. Parker, W. Thompson, and E. Reinhard, "Temporally coherent interactive ray tracing," *J. Graph. Tools*, vol. 7, no. 2, pp. 41–48, 2002.

[41] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney, *Level of Detail for 3D Graphics*. New York: Elsevier, 2002.

[42] M. Anid, N. Bagherzadeh, N. Tabrizi, H. Du, and M. Sanchez-Elez, "Interactive ray tracing using a SIMD reconfigurable architecture," in *Proc. SBAC-PAD*, 2002, pp. 20–28.

[43] H. Du, A. Sanchez-Elez, N. Tabrizi, N. Bagherzadeh, M. Anido, and M. Fernandez, "Interactive ray tracing on reconfigurable SIMD morphoSys," in *Proc. ASP-DAC*, Jan. 21–24, 2003, pp. 471–476.

[44] *The RenderMan Interface*. [Online]. Available: http://renderman.pixar.com/products/rispec/rispec_pdf/RISpec3_2.pdf

[45] P. Shirley, *Fundamentals of Computer Graphics*. Natick, MA: A. K. Peters, 2002.

[46] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. Boston, MA: Addison-Wesley, 1990.

[47] K. Perlin, "An image synthesizer," *ACM SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296, 1985.

[48] J. C. Hart, "Perlin noise pixel shaders," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graph. Hardware (HWWS)*, 2001, pp. 87–94.

[49] K. Perlin, "Improving noise," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 681–682, 2002.

[50] P. Shirley, K. Sung, E. Brunvand, A. Davis, S. Parker, and S. Boulos, "Rethinking graphics and gaming courses because of fast ray tracing," in *Proc. ACM SIGGRAPH Educators Program*, 2007.

[51] F. E. Nicodemus. (1965, Jul.). Directional reflectance and emissivity of an opaque surface. *Appl. Opt.* [Online]. *4(7)*, pp. 767–773. Available: http://ao.osa.org/abstract.cfm?URI=ao-4-7-767

[52] G. J. Ward, "Measuring and modeling anisotropic reflection," in *Proc. 19th Annu. Conf. Comput. Graph. Interactive Tech. (SIGGRAPH)*, 1992, pp. 265–272.

[53] D. Burger and T. Austin, "The simplescalar toolset, version 2.0," Univ. Wisconsin-Madison, Madison, Tech. Rep. TR-97-1342, Jun. 1997.

[54] Synopsys Inc. [Online]. Available: http://www.synopsys.com

[55] *Artisan CMOS Standard Cells*. [Online]. Available: http://www.arm.com/products/physicalip/standardcell.html

[56] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," *Comput. Graph. Forum (Proc. EUROGRAPHICS 2001)*, vol. 20, no. 3, pp. 153–164, 2001.

[57] J. Bigler, A. Stephens, and S. Parker, "Design for parallel interactive ray tracing systems," in *Proc. IEEE Symp. Interactive Ray Tracing*, Sep. 2006, pp. 187–196.

[58] C. Gribble and K. Ramani, "Coherent ray tracing via stream filtering," in *Proc. IEEE Symp. Interactive Ray Tracing (RT)*, Aug. 2008, pp. 59–66.

[59] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *Proc. SIGGRAPH*, 1984, pp. 165–174.

Josef Spjut (S'04) received the B.S. degree in computer engineering from the University of California, Riverside, in 2006. He is currently working toward the Ph.D. degree at the University of Utah, Salt Lake City.

His research interests include computer architecture, very large scale integration circuits, and computer graphics.

Andrew Kensler received the B.A. degree in computer science from Grinnell College, Grinnell, IA, in 2001. He is currently working toward the Ph.D. degree in the School of Computing, University of Utah, Salt Lake City.

He is also a Research Assistant with the Scientific Computing and Imaging Institute, University of Utah. His research focuses on interactive ray tracing, with interests in hardware ray tracing and photorealistic rendering.

Daniel Kopta (S'08) received the M.S. degree from the University of Utah, Salt Lake City, in 2008, where he is currently working toward the Ph.D. degree.

His research interests include computer graphics, ray tracing, and machine learning.

Erik Brunvand (S'89–M'94) received the M.S. degree from the University of Utah, Salt Lake City, in 1984 and the Ph.D. degree from Carnegie Mellon University, Pittsburgh, in 1991.

He is currently an Associate Professor with the School of Computing, University of Utah. His research interests include computer architecture, very large scale integration circuits, asynchronous and self-timed circuits and systems, and computer graphics.