EFFICIENT RAY TRACING ARCHITECTURES

by

Josef Bo Spjut

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2015

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Josef Bo Spjut**

has been approved by the following supervisory committee members:

Erik Brunvand	, Chair	**Dec 20, 2013** Date Approved
Alan Davis	, Member	**Dec 20, 2013** Date Approved
Rajeev Balasubramonian	, Member	**Dec 20, 2013** Date Approved
Peter Shirley	, Member	**Dec 20, 2013** Date Approved
Ingo Wald	, Member	**Sept 17, 2014** Date Approved

and by **Alan Davis** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

This dissertation presents computer architecture designs that are efficient for ray tracing based rendering algorithms. The primary observation is that ray tracing maps better to independent thread issue hardware designs than it does to dependent thread and data designs used in most commercial architectures. While the independent thread issue causes extra overhead in the fetch and issue parts of the pipeline, the number of computation resources required can be reduced through the sharing of less frequently used execution units. Furthermore, since all the threads run a single program on multiple data (SPMD), thread processors can share instruction and data caches. Ray tracing needs read-only access to the scene data during each frame, so caches can be optimized for reading, and traditional cache coherence protocols are unnecessary for maintaining coherent memory access. The resultant image exists as a write only frame buffer, allowing memory writes to avoid the cache entirely, preventing cache pollution and increasing the performance of smaller caches.

Commercial real-time rendering systems lean heavily on high-performance graphics processing units (GPU) that use the rasterization and z-buffer algorithms for rendering. A single pass of rasterization throws out much of the global scene information by streaming the surface data that a ray tracer keeps resident in memory. As a result, ray tracing is more naturally able to support rendering effects involving global information, such as shadows, reflections, refractions and camera lens effects. Rasterization has a time complexity of approximately $O(Nlog(P))$ where N is the number of primitive polygons and P is the number of pixels in the image. Ray tracing, in contrast, has a time complexity of $O(Plog(N))$ making ray tracing scale better to large scenes with many primitive polygons, allowing for increased surface detail. Finally, once the number of pixels reaches its limit, ray tracing should exceed the performance of rasterization by allowing the number of objects to increase with less of a penalty on performance.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I took a long path to get to this point where I was able to complete this dissertation. Obviously, I did not become who am I or complete all of this work on my own, so I would like to take this chance to acknowledge some of the influences on my life up to this point.

First, I feel a need to recognize my dad, Erik Spjut, for instilling in me a curiosity about this world from my youth. When I found a book on programming in elementary school, he installed a compiler and encouraged me to start programming. He also was a big part of the reason I ended up at the University of Utah, it being his undergraduate alma mater. From my childhood, I always had a desire to pursue a Ph.D. since the man I looked up to most had one.

I also would be negligent to not mention my mother, Karen Spjut, who has done her best to encourage and support me through all the years. I remember the many times in my primary and secondary education where she would stay up late or rise extremely early to provide moral support while I completed my school work. Unfortunately, I have not yet been able to break this trend, as anyone who worked on a paper deadline with me can attest to.

When it came time to select a graduate school and advisor, I really did not know what I was looking for or what would be important for me in completing graduate school. I managed to secure a position working with Erik Brunvand working on a project I was and am passionate about (much of which is represented in this dissertation). Erik was a great example for me and was the perfect advisor for keeping my mental stability even though I was not always able to complete all tasks as promptly as we would have liked. Erik helped me develop as a researcher, but he also encouraged artistic expression, and helped me develop my teaching ability as much as I let him.

I should also thank my committee, Al Davis, Rajeev Balasubramonian, Pete Shirley, and Ingo Wald for the encouragement, support, and many great discussions we had over the years. I was able to work with a number of excellent graduate students as well, but feel an extra acknowledgement should be given to Daniel Kopta, who was a co-author for all of the papers that contributed to this dissertation. The Utah architecture lab members gave me valuable feedback and provided many enjoyable conversations and diversions over the years. I should also thank Karen Feinauer and Ann Carlstrom for the excellent institutional support they provided.

CHAPTER 1

INTRODUCTION

Computer generated images are used for many professional tasks including movies, video games, computer aided drafting, visualization, medical imaging, and others. These images are useful for reducing costs, providing added insight, and increasing the effectiveness of workers in their respective fields. There are two principal metrics that the users of these applications want from image rendering: image quality and speed of image synthesis, often called framerate. In situations where image quality is the most important, offline rendering is typically used where a computer, or a cluster of computers, spends a very long time to generate one image. When framerate and interactivity are more important, shortcuts are taken and often special purpose hardware is used to generate images many times a second. A common metric to measure speed of rendering is the number of frames per second which is usually abbreviated as *fps*. In both cases, there are two primary classes of algorithms used: rasterization and ray tracing. While other methods for image synthesis do exist, this dissertation will focus primarily on these two techniques. Considering a first order approximation, rasterization algorithms scale linearly with the number of triangles or primitives in the scene while ray tracing algorithms scale linearly with the number of pixels or samples in the final image. Both algorithms have been used in both offline rendering (seconds to hours per frame), and interactive (5-15 fps) to real-time (20+ fps) rendering.

1.1 Existing Approaches to Graphics Processing

At present almost every personal computer has a dedicated processor that enables interactive 3D graphics, whether a discrete add-in card or integrated on the same die as the main CPU. These graphics processing units (GPUs) implement some version of the *z-buffer* algorithm introduced in Catmull's landmark University of

Utah dissertation [3]. 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 an image-based texture lookup or through 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 Figure 1.1). 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 commercial 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 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. The most common trick is to add an extra pass through all of the geometry in the scene, but that is inherently wasteful since it consumes communication resources for redundant data.

Modern GPUs can interactively display several million triangles in complex 3D environments with image-based (look-up) 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:

- those that have datasets much larger than a few million triangles such as vehicle design, landscape design, manufacturing, complex movie scenes, and some branches of scientific visualization;

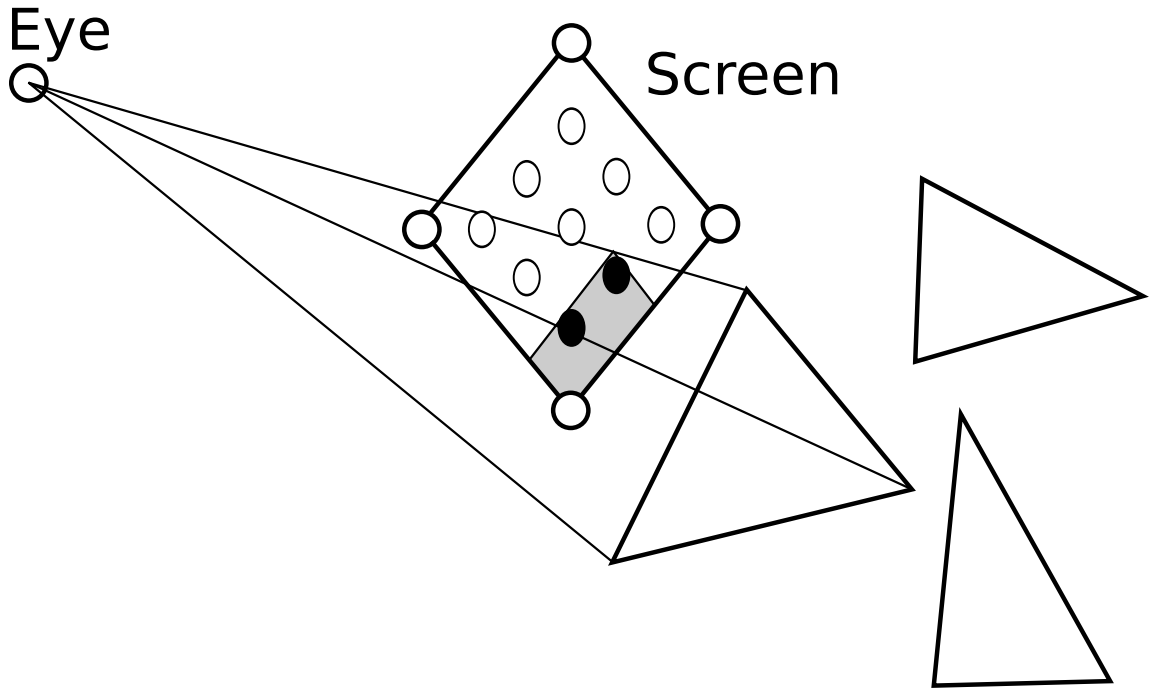


Figure 1.1: The Z-buffer Algorithm

- those that have nonpolygonal data not easily converted into triangles;
- those that demand high quality shadows, reflection, refraction, and indirect illumination effects such as architectural lighting design, rendering of outdoor scenes, realistic movie scenes, and vehicle lighting design.

These classes of applications typically use Whitted's ray tracing algorithm [4, 5, 6]. The ray tracing algorithm is better suited to huge datasets 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 datasets 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 [7, 8]. 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 provides 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.

1.2 Raster Graphics

Commercial hardware for rasterization must all implement an interface to the widely used OpenGL [9] and DirectX [10] libraries in order to be competitive. Both libraries are very similar and perform the same set of functions, but with subtle differences. To understand how graphics hardware is implemented, it is interesting to examine the OpenGL pipeline (chosen because it is an open standard). The OpenGL Programming Guide [11] discusses the key stages in the OpenGL pipeline, which include: pixel operations, per-vertex operations, texture assembly, rasterization, and per-fragment operations, as seen in Figure 1.2. In older GPU architectures, the programmable vertex and pixel shaders were handled by separate pipeline units, but modern GPUs have become unified and perform all programmable processing in the same set of units. Advanced Micro Devices (AMD) recently announced Mantle [12] technology uses the same style of processing while exposing different parts of the underlying architecture explicitly to the programmer.

Rasterization is known to work well in single instruction multiple data (SIMD) execution, which is why GPUs that are optimized for it are SIMD architectures. Each triangle is first projected to screen space, a series of operations that are repeated

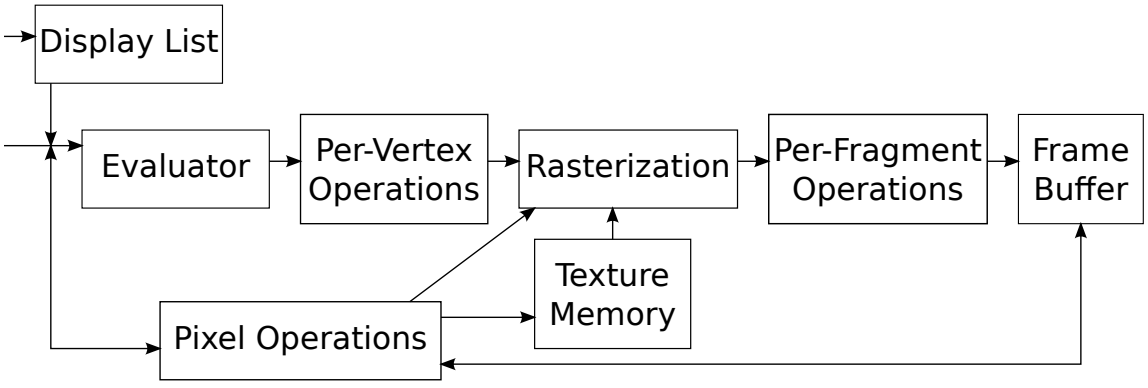


Figure 1.2: OpenGL Pipeline

for each triangle, as can be seen in Figure 1.1. This projection is performed by multiplying each vertex by a 4x4 transformation matrix to place it in the *canonical viewing volume*. Each vertex can also have operations performed on it which are repeated for each vertex, and are optimally nondivergent. Subsequently, a set of fragments are generated by an optimized scan-line conversion process, which when using the same bounding box, does not diverge. Then shading is performed on each of the fragments that graduates into the frame buffer by passing a "z" test. Each phase of typical rasterization can easily be mapped to efficient SIMD execution. Due to the fact that triangles are considered to be entirely independent, rasterization lends itself well to GPU-style SIMD.

1.3 Ray Tracing

Ray tracing as an algorithm is designed to approximate the physics of light and more easily achieves some of the complex lighting effects that are more difficult to achieve with rasterization. Much work is being done to attempt to bring the ray tracing algorithm to real time, or at least interactive frame rates. Most modern ray tracers resemble to a great extent the 1980s Whitted ray tracer [4] with improvements and optimizations.

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 3d 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 (Figure 1.3). 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, 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

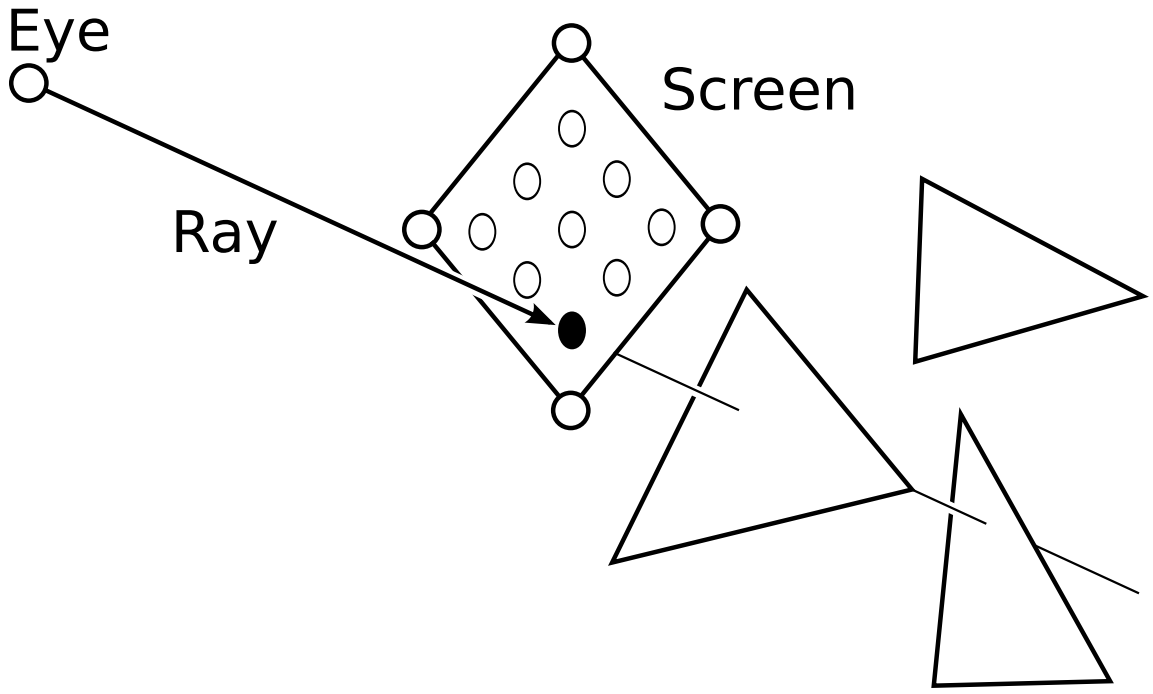


Figure 1.3: The Ray Tracing Algorithm

relatively simple memory system can keep the multiple threads supplied with data. It should be noted that some level of parallelism can be extracted by intersecting a given ray with multiple triangles at once, something that is exploited by Intel's Embree [13, 14].

The three main kernels of a ray tracer are traversal, intersection, and shading. Traversal is the first phase of ray tracing computation and involves the traversal of a tree-like structure to reduce the number of intersection tests that must be performed. These tree-like structures are called *acceleration structures* and are discussed in more detail in Section 1.3.1. Intersection is the phase of ray tracing where rays are tested for intersection with the primitive objects in the scene being rendered. Shading is the final stage of computation where the final color is determined based on the color of objects, the positions of the lights and the impact of other objects in the scene on the illumination of objects. Shading in ray tracing is essentially identical to shading in rasterization with an important difference being the casting of additional rays for secondary lighting effects. Most of the general improvements that have been made for shading on GPU-based rasterizers can be applied to shading when using a ray

tracer. Therefore the two most interesting phases to look at when improving ray tracing performance are the traversal and intersection.

Primitive intersection involves computing the intersection of a ray with a primitive, usually a triangle. A system which includes programmable intersection operations could easily be extended to intersect arbitrary types of geometric primitives, however, triangles are the most interesting for supporting legacy scenes and current production tools for video games and movies (to a lesser extent). From each point where the ray intersects an object in the scene, an additional "secondary" ray can be recursively cast into the scene to determine optical effects such as shadows, reflections, refraction, caustics (focused light from an indirect source), and other global illumination and optical effects. Ray tracing has distinct advantages over rasterization in terms of its ability to easily render these optical effects, making ray tracing the rendering algorithm of choice for highly realistic images. Ray tracing can also be effectively used for traversing volumetric data and large datasets, such as medical images and other scientific data. Work has been done to optimize the exact computation done when intersecting a ray with a triangle [15, 16], but without a smart method to choose which rays need to check for intersection with which triangles, an exhaustive search of the triangles would need to be performed.

1.3.1 Acceleration Structures

An *acceleration structure* is used by a ray tracer to greatly improve the speed and efficiency of rendering. Without an acceleration structure, the total render time for a scene is on the order of the number of pixels times the number of objects in the scene. It is the acceleration structure that allows for logarithmic time complexity in the number of objects in the scene. In order for an acceleration structure to work most efficiently, it would instantly determine which objects in the scene are intersected by the ray in question, and which ones are missed entirely. A simple example would be the case where the entire set of objects in the scene can be enclosed by a single box, and by performing a single ray-box intersection test, the ray tracer can find if a ray misses the box entirely, therefore missing the scene entirely. This single intersection test is much quicker than performing an intersection test with all but the most trivial

scenes. When the ray-box test comes back positive, then the ray tracer does not know whether an object is actually hit or not, and additional computation must be performed. The scene could then be further divided into subscenes, each with their own bounding box, in order to hopefully eliminate large chunks of the geometry from each intersection test. While this technique is useful, the performance of an acceleration structure varies greatly from scene to scene and can also depend on the ray direction and which kind of build technique was used to generate the acceleration structure.

A variety of acceleration structures are used regularly in ray tracers [17] including KD-trees, bounding volume hierarchies (BVH), and grids. While each acceleration structure has its benefits, this work uses BVHs because they guarantee that each triangle is in only one leaf node and that each leaf node has at least one triangle in it and are fairly efficient at removing unnecessary volumes of space quickly. The performance of traversing an acceleration structure can be improved by either optimizing the creation of the structure for better run-time performance [18], or by using a structure that better culls away the empty space. BVHs are typically used for most animations because a simple refit can be performed relatively cheaply without a huge loss in traversal performance over time [19, 20]. Some work is more focused on reducing the build time of structures to facilitate dynamic scenes [21].

1.3.2 Ray Tracing Optimizations

Since recent GPU and CPU architectures depend heavily on SIMD for performance, work has been done to parallelize ray traversal across 4-wide (or sometimes wider) SIMD units. Typically a set of rays are grouped together in a packet to be traced together. This packet is further simplified during traversal by using a proxy for the packet, such as a frustum [22, 23] or a subset of the rays in the packet [24]. These techniques, often called *packet tracing*, are known to work well for primary visibility in ray tracing, but are less efficient when the rays become less spatially coherent, which happens when generating the more interesting lighting effects that are desired from ray tracing. Parallelization has also been analyzed for the construction of acceleration structures [25].

NVIDIA introduced their OptiX engine for ray tracing [26] to also take advantage of SIMD compute resources to accelerate ray tracing. They divide the ray tracing algorithm up into a set of kernels that can each execute independently (though they might just be different inner loops of what is called a "megakernel") to reduce the impact of divergence in the acceleration structure traversal. This technique requires additional overhead in managing the individual threads and efficiently assigning the threads to available execution resources. In the Fermi architecture [27] and beyond, much effort has been put forth to allow for multiple simultaneous kernels to execute and for faster thread context switching. Wavefront formulation [28] has been proposed as an improvement over the more traditional megakernel approach. Other ray tracers have been made to use CPU SIMD extensions [14] and to even execute on architectures like the Cell [29]. These efforts require significant reorganization of memory and are no small task for the programmer to implement correctly.

A *path tracer* is similar in most ways to a ray tracer, especially in the overall flow of the program, however, the memory access pattern is different due to the selection of rays being traced. While the term "ray tracer" can describe a broad set of approaches involving the tracing of rays through a scene, a Cook-style path tracer [30] only traces a single path of light by randomly choosing one of the possible rays that could be chosen at each decision point. For example, when a ray hits a diffuse surface where the light is expected to bounce in any direction with equal probability, the secondary ray direction is chosen randomly in the hemisphere described by the normal at that intersection point. Similarly, when approximating blurry reflections, the ray directions will prefer to be chosen in some cone mostly facing the direction of a perfect reflection on that surface. Due to the stochastic sampling performed by Cook-style path tracers, a large number of samples should be gathered per pixel in order for the final image to converge to a set of values that are close to correct.

Some 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 [31], the display of scanned data [32], and terrain rendering [33]. While level-of-detail (LOD) techniques can sometimes make display of geometrically simplified data possible, such procedures typically require costly prepro-

cessing and can create visual errors [34]. Simulation and games demand interactivity and currently use z-buffer/rasterization 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 images of inferior quality to those generated by ray tracing. I believe that these industries based on interactive graphics rendering would use ray tracing if a solution existed that were fast enough.

As mentioned previously, shading is an important part of both ray tracing and rasterization. The computations involved are the same and the architectural and algorithmic improvements for GPU rasterizers as well as offline renderers are applicable to both algorithms. A variety of shading techniques are commonly used, such as Gouraud [35] and Phong [36] shading. Image textures or computed textures can be sampled as well to get the base color for the shading computation and can also be used to perform bump mapping [37].

1.4 Applications and Scenes

In this dissertation, there are two kinds of ray tracers used to evaluate the architectural improvements. They are discussed in more detail in Section 4.6. The first ray tracer is a Kajiya style [8] ray tracer that supports recursive ray tracing and standard surface shading techniques (lambertian and phong). This Kajiya ray tracer only supports a single light source, and casts a shadow ray during shading to determine if the surface is in light or shadow. A version of this type of ray tracer exists written entirely in our TRaX assembly language, another compiled for a variable number of registers with the call stack in registers, and a third compiled for 32 registers using only the local store memory for program stack. The second kind of ray tracer is a Cook-style path tracer. The path tracer computes global illumination in the scene by allowing many samples to be taken while tracing each sample to a fixed ray depth on each run. The shading technique used is a Monte-Carlo sample lambertian shading. Some more advanced and varied rendering techniques have also been implemented successfully on the architecture, but they will not be represented in the results of this dissertation.

A number of different scenes are regularly used to evaluate the performance of ray tracers in order to allow for comparison of different techniques and modifications. Figure 1.4 shows a number of scenes that are used in a number of chapters in this dissertation. More details on these scenes and their use in simulation can be found in Chapter 5. In general, the scenes with higher triangle counts tend to require more memory bandwidth to render since they cannot fit in the caches and trend towards a larger working set. The scenes with fewer triangles tend to have fewer misses in the caches, and result in faster render times. However, the quality and kind of acceleration structure have a large affect on the render time of the scenes, so there are some experiments where a scene with relatively few triangles causes difficulty for the acceleration structure builder, resulting in more memory bandwidth usage, lower hit rates, and longer render times.

1.5 Dissertation

Most interactive graphics applications follow a trend of models increasing in size (Greenberg has argued that typical model sizes are doubling annually [38]). Most applications also demand increasingly more visual realism in order to provide more utility to their professional users and more enjoyment for video gamers. I believe these trends favor ray tracing in the future (either alone or in combination with rasterization for some portions of the rendering process). Following the example of graphics processing units (GPUs), I 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

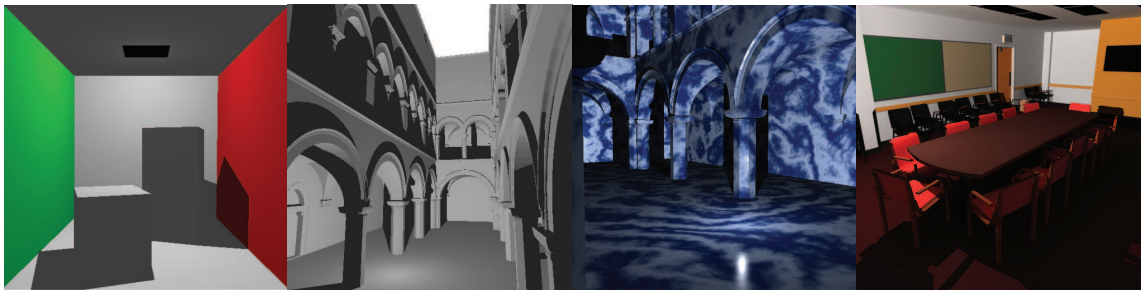


Figure 1.4: Example Ray Traced Scenes

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 and more on these studies can be found in Chapter 6.

Both rasterization and ray tracing can be expressed as SPMD (single program multiple data) applications, where a single program is executed on each independent thread. This dissertation focuses on the ray tracing algorithm as the method of producing high quality images for future applications. While many ray tracers exist that run on CPUs, the architecture of a CPU has many features for general purpose processing that are not needed for ray tracing. GPUs use rasterization for rendering and are designed to accelerate rasterization even at the expense of general purpose computation performance. The fact that GPUs use SIMD (single instruction, multiple data) to perform rasterization suggests that rasterization maps well to a SIMD architecture. Note that while GPU vendors have frequently used the term SIMT (single instruction, multiple threads) to describe the fact that each data element in effect comes from a separate logical thread with hardware support for thread divergence, this dissertation generally uses the terms, SIMD, SIMT and GPU SIMD interchangeably, except where specified. Even though recent research has shown ways to map ray tracing to SIMD architectures, the ray tracing algorithm is generally not efficiently executed on SIMD hardware because of poor SIMD utilization. This lack of compatibility with SIMD hardware stems from the way any arbitrary group of rays is likely to intersect different objects in the scene. Rasterization hardware takes advantage of the independence of the triangles to stream them through wide SIMD computation units. Ray tracing, in contrast, performs a logarithmic-time lookup of the primitives, which can perform better when each independent lookup is allowed to become out of sync with other searches. This is more naturally supported in a loosely coupled SPMD/MIMD architecture, though there is extra overhead from a naive MIMD implementation. CPU SIMD extensions have been shown to increase performance greatly for CPU architectures, primarily because the SIMD extensions are included at a small overhead to the already large CPU architecture. It is possible to use simpler individual processing units for higher thread-level parallelism, removing the need for CPU SIMD extensions. The incremental overhead of adding extra

processing units can be greatly reduced by increasing the level of sharing among a group of these units where independent execution units provide little benefit.

In general, this dissertation explores architectural approaches to improving ray tracing performance, while still retaining a high level of programmability. Specifically, I will show that a throughput-oriented architecture with many simple cores operating in SPMD/MIMD mode can have significantly better performance on applications like ray tracing than fewer heavy-weight cores, like a CPU with SIMD extensions or an architecture where the threads are more closely coupled such as in a SIMD GPU. I present an architecture called TRaX (for Threaded Ray eXecution), which is specifically designed to exploit the opportunities available in the ray tracing algorithm for independent thread parallelism. This kind of parallelism is different from the SIMD style of parallelism that is so popular in current GPUs and CPUs, but is much better suited to ray tracing and other similarly branchy applications, though I provide no explicit analysis of other applications. In fact, CPUs and GPUs both support the same kind of parallelism as TRaX, but to a much lesser extent. Each core of a CPU is independent in the same way that each thread in TRaX is independent, and each SIMD cluster in a GPU could be caused to operate in a scalar fashion to also have independent thread execution. Clearly these techniques would greatly under-utilize the hardware available in both a CPU and a GPU, but are in fact what happens when those machines encounter the worst case program divergence. Using improvements proposed for the TRaX architecture, real time rendering can much more quickly approach the quality of the offline-rendered images seen in Figure 1.5. The existence of commercial ray tracing processors would cause a new revolution in the real-time rendering community.

Figure 1.5: Path Traced Images from Lux [1] at 128 Samples Per Pixel

CHAPTER 2

A MIMD THROUGHPUT COMPUTE SOLUTION[1][2][3]

As the title of this chapter suggests, the solution I propose for ray tracing involves MIMD (multiple instruction, multiple data) processing. TRaX (for threaded ray execution) is specifically designed to accelerate single-ray performance and to exploit thread-level parallelism using multiple thread processors and cores. Ray tracing is an application with the potential for massive amounts of thread-level parallelism. For example, a single frame of a 1080p HD image contains about two million pixels, each of which represents at least one primary ray. Furthermore, high quality ray tracing greatly benefits from increased sampling beyond a single primary ray (16 samples per pixel is a good start), and secondary rays can also be traced independently to create greater opportunities for thread-level parallelism. A high quality two million pixel ray traced image with two diffuse bounces and two point light sources along with 16 samples per pixel would involve 96 million illumination rays that require full shading computations and 192 million shadow rays that contribute to lighting through the same acceleration structure traversal for a total of 288 million rays. The potential number of parallel hardware threads that may improve the performance of a ray tracer is in the millions.

---

[1]This chapter is modified with permission from J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A multicore hardware architecture for real-time ray tracing," *IEEE Transactions on Computer-Aided Design*, vol. 28, no. 12, pp. 1802 – 1815, Copyright ©2009 IEEE.

[2]This chapter is modified with permission from D. Kopta, J. Spjut, E. Brunvand, and A. Davis, "Efficient MIMD architectures for high-performance ray tracing," in *IEEE International Conference on Computer Design (ICCD)*, Copyright ©2010 IEEE.

[3]This chapter is modified with permission from J. Spjut, D. Kopta, E. Brunvand, and A. Davis, "A mobile accelerator architecture for ray tracing," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, Copyright ©2012.

2.1 Parallelism Considerations

MIMD is a style of parallelism that involves duplicating the processor to allow multiple processors to execute in parallel. While it may initially appear trivial to duplicate a processor, there are a number of difficulties in allowing multiple processors to execute simultaneously, and nearly all of the difficulties involve the system required to deliver the data values to each processor. SIMD (Single Instruction, Multiple Data) instead exploits the situation when a program needs to perform the same basic operation on multiple pieces of data in parallel. If a single executing thread performs SIMD instructions, the processor is sometimes referred to as a vector processor since the multiple data elements can be thought of as a vector. When the multiple data elements are instead considered to come from independent threads, the term SIMT (Single Instruction, Multiple Threads) is often used. Typically, SIMT also indicates that the processor has hardware support for masking off computations for threads that may have divergent execution paths due to conditional branching.

A ray tracer can be expressed in a Single Program, Multiple Data (SPMD) programming model relatively easily. This single program could be executed using a traditional, single thread of execution trivially by allowing the program to loop over all tasks until they are all completed. A natural option would be to parallelize tasks using a MIMD processor with many hardware threads with full support for thread divergence. However, most commercial architectures include support for SIMD or SIMT processing, and as a result, heroic efforts are exerted to keep the data elements, or coupled threads from diverging as much as possible to match the underlying processor's limitations. I propose only using hardware features that work well with the natural behavior of the algorithm, rather than warping the fundamental ray tracing algorithm to match hardware designed for other applications.

Chapter 6 details many other architectures used for ray tracing [39, 40, 29, 41, 42, 43, 44, 45], which 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 [46]. 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 TRaX threads are independent we do not have to wastefully mask off results of functional unit operations.

SIMD works well as long as all of the threads attempt to execute the same instruction all the time. In fact, SIMD is better in this case because there is a reduction in the instruction fetch and decode stage of the pipeline. There exists a set of applications, however, that are known to not execute well on SIMD architectures because of poor utilization of the execution units. These applications suffer from many branches where a SIMD block has a high chance of diverging, causing some of the threads to execute different instructions than the rest of the threads.

There are some ways to reduce the loss in performance from divergent branching, such as filtering threads, and causing threads to block at certain points until the divergent threads reconverge. However, these techniques still result in additional overhead and reduced SIMD utilization. A standard high-performance ray tracer uses a tree-like acceleration structure to increase the performance of the visibility query per ray. Divergence comes when the individual rays take a different path through the data structure and end up in different phases of the computation. For instance, one thread may only need to intersect 32 nodes while another thread may require 1024 node intersection tests before moving on to triangle intersections. It is highly inefficient to have the first thread stand idle for such a large amount of time when it could continue on and begin working on a new ray. In fact, the percentage of threads able to issue on average per cycle is often used to report the efficiency of such SIMD systems and is called *SIMD efficiency*.

2.1.1 Nonparallel Applications

There exist some applications that are entirely serial in nature, or have a large percentage of the execution that cannot be made parallel. These applications are not very interesting in a study of parallel architectures since they do not benefit from any of the parallel improvements. CPUs are designed to extract instruction-level parallelism and can therefore do a good job of speeding up these applications without the use of SIMD extensions. Features like out-of-order issue, super-scalar use of multiple issue slots, and deep pipelines with aggressive branch prediction can improve

performance where SIMD and MIMD multithreading designs have essentially no effect on nonparallel program run-time.

2.1.2 Parallel but Branchy

There is another set of applications that are highly parallel, often called embarrassingly parallel, that can benefit greatly from a large number of parallel threads. In many cases these applications have some outer loop that has few dependencies across iterations of the loop, or at least dependencies that still allow for parallelization. These applications are of the set that can benefit from parallel hardware implementations. SIMD is a good match for such an application as long as the time spent in one iteration, and the instructions issued within that iteration, are relatively consistent. When the core operations have a high amount of variability then a MIMD design is likely to come out on top.

MIMD is a style of execution that, in contrast to SIMD, allows each parallel thread of execution to both operate on different data and execute different instructions. This can be found in multicore CPUs as each core executes its own instruction and is not required to execute the same instruction as any other core. Even on wide SIMD architectures like GPUs, there are many separate SIMD blocks that can each execute different instructions even though all the threads within a SIMD block must have the same instruction.

Vector computing is implemented very similarly to SIMD and the two terms are often used interchangeably in the literature. There is, however, a subtle difference between them and how these terms will be used here. Vector computing is when a single thread of execution has an operation to perform on data that can be thought of as a vector data type. So while SIMD executes many scalar operations from different threads or data streams in parallel, vector computing issues a vector operation from a single thread.

2.2 The TRaX Architecture

The vast majority of related commercial and research projects used to accelerate ray tracing depend on SIMD architectures to perform ray tracing. The huge constraint all of those projects deal with is the need for the threads executing in SIMD to remain

synchronized as much as possible to achieve high SIMD utilization. Any time a thread in a SIMD block decides it needs to test for intersection of a node or primitive that the other threads do not need to test, the work is wastefully repeated for all of the other threads.

The TRaX architecture [47, 48, 49, 50, 51, 52] is a many-threaded architecture designed for fast ray tracing throughput. The approach is to optimize single-ray MIMD performance across many simple cores. This single-ray programming model loses some primary ray performance. However, it makes up for this by handling secondary rays (discussed in Section 1.3) nearly as efficiently as primary rays, where SIMD optimized ray tracers struggle. In addition to providing high performance, this approach can also ease application development by reducing the need to orchestrate coherent ray bundles. For the analysis of architectural options, a number standard ray tracing benchmark scenes are used, four of which are shown in Figure 2.1. More of the test scenes can be found in Chapter 5 along with detailed results from simulations. These scenes provide a representative range of performance characteristics.

Threads represent the smallest division of work in the ray-traced scene, so 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 thread processors, called TPs, that include some functional units in each TP with other larger functional units being shared among nearby TPs. A collection of these TPs, their shared functional units, issue logic, and shared L1 cache are collected into a "thread multiprocessor," or TM.

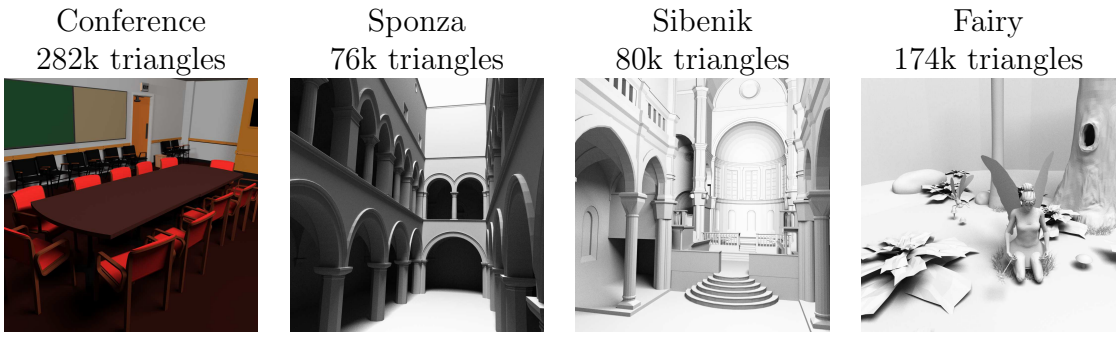


Figure 2.1: Test Scenes Used to Evaluate Performance. (a) Conference (b) Sponza Atrium (c) Sibenik Cathedral (d) Fairy Forest

A full chip consists of many TMs, each containing many TPs, sharing one or more on-chip L2 caches and off-chip memory and I/O bandwidth. Because of the parallel nature of ray-tracing, threads (and thus TPs) 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.

In TRaX, the lack of synchrony between ray threads reduces resource sharing conflicts between the cores and reduces the area and complexity of each core. With a shared multibanked Icache, the cores quickly reach a point where they are each accessing a different bank. Shared functional unit conflicts can be similarly reduced. Given the appropriate mix of shared resources and low-latency Dcache accesses, TRaX can sustain a high instruction issue rate without relying on latency hiding via thread context switching. This results in a different ratio of registers to functional resources for TRaX processing elements when compared to the hardware in commercial GPUs. The GPU approach involves sharing a number of thread states per core, only one of which can attempt to issue on each cycle. TRaX Thread Multiprocessors (TM) contain one thread state per Thread Processor (TP), each of which can potentially issue an instruction to a private per-core Functional Unit (FU) or one of the shared FUs. I believe this single thread-state approach is a more efficient use of register resources.

TRaX relies on asynchrony to sustain a high issue rate to the heavily shared resources, which enables simpler cores with reduced area, breaking the common wisdom that the SIMD approach is more area efficient than the MIMD model, at least for ray tracing. MIMD threads are allowed to progress through the program at their own pace, meaning that the Program Counters (PC) among a group of threads do not maintain the same value as is required for SIMD execution. Threads do not get significantly out of sync on the workload as a whole, thus maintaining coherent access to the scene data structure, and results in high cache hit rates.

2.2.1 A Thread Processor

Each TP in a TRaX TM can execute the single program based on its own program counter in a standard SPMD/MIMD fashion as defined in Chapter 1, where a software thread corresponds to a ray. In addition to the private program counter, each thread also maintains a private register file. The register file is a simple 2-read, 1-write static read only memory (SRAM) block. Register forwarding may be enabled in our simulator and allows operands to be available to instructions before the write-back stage of the pipeline. The type and number of independent functional units is variable in the TRaX simulator. More complex functional units are shared by the TPs in a TM. The TPs in a TM also share access to a multibanked instruction cache for the shared set of instructions. A block diagram of a thread processor can be found in Figure 2.2. Note that the local memory is optional and is not included in all TRaX designs, but it used in order to restrict the register file to 32. When the local memory is removed, the register file extends to 128 general purpose registers.

For a simple ray tracing 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 instruction set architecture (ISA). We include bitwise instructions, branching, floating point/integer conversion, memory

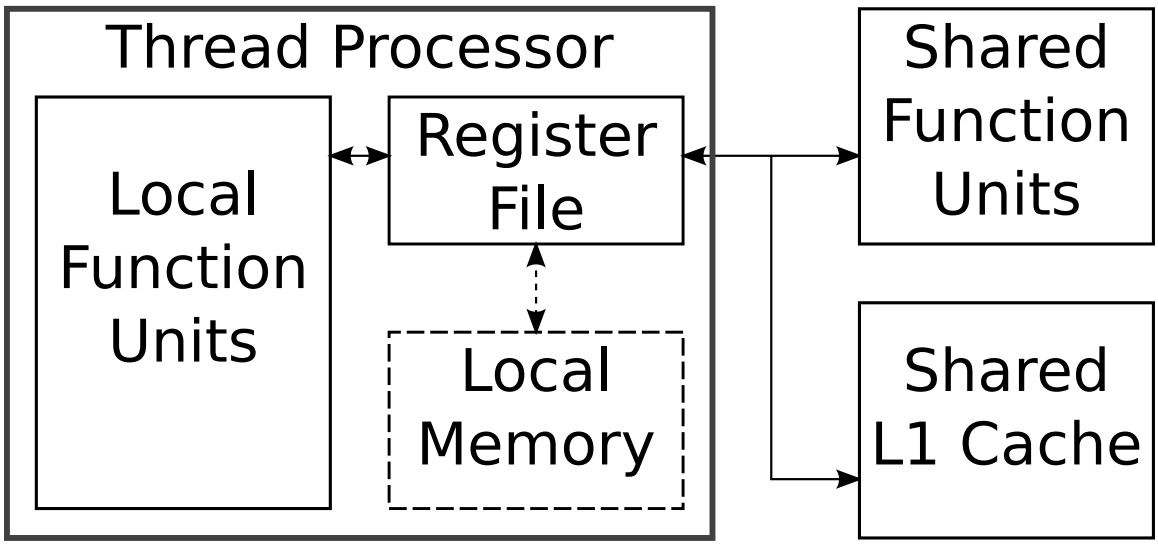


Figure 2.2: Thread Processor Block Diagram

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.

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 functional unit required, and the writeback takes a final cycle. Instructions issue in order, but may complete out of order. Thread processing stalls if needed data are not yet available for register forwarding, or if the desired functional unit is not available, but correct single-thread execution is guaranteed. Each TP is configured to issue a single instruction per cycle, although the simulator has support for higher thread issue width.

Because issue logic is split between the individual thread processor and the shared units in the TM, only some of the complexity in terms of dependence checking is internal to each thread. A simple table maintains instructions and their dependencies. Instructions enter the table in FIFO 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 dependent on the programmer/compiler who must order instructions intelligently to hide functional unit latencies as often as possible. An important point to remember though, is that the overall system throughput is what is most important for ray tracing, not the performance of an individual thread.

Depending on which version of TRaX is being simulated, a number of thread features can be turned on or off as needed. The thread register file can be restricted to a size of 32 registers (the default in our compiler, though other register restrictions could be allowed) when using a compatible compiler, in which case a small local storage SRAM is used for thread-local stack space. However, when other compilation techniques are used, the architecture may instead be configured with a larger register file, allowing thread-local stack operations to be performed through special register offset instructions. The specifics of these configurations and their associated simulations are reported in Chapter 5.

2.2.2 A Collection of Threads in a Thread Multiprocessor

Each of the Thread Multiprocessors(TM) on a chip consists of a set of simple thread processors with shared L1 data cache and shared functional units, as shown in Figure 2.3(a). As mentioned above, the set of TPs in a TM share access to a many-banked instruction cache that can be varied in our simulations. Additionally, all threads in a TM share one multibanked L1 data cache of a modest size, also variable in our simulations. One example is 2K lines of 16-bytes each, direct mapped, with four banks. Many TMs on a multi-TM chip share a L2 unified data cache as the total code size is small enough to fit in a 4kB L1 instruction 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 needs to be read back, so 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.

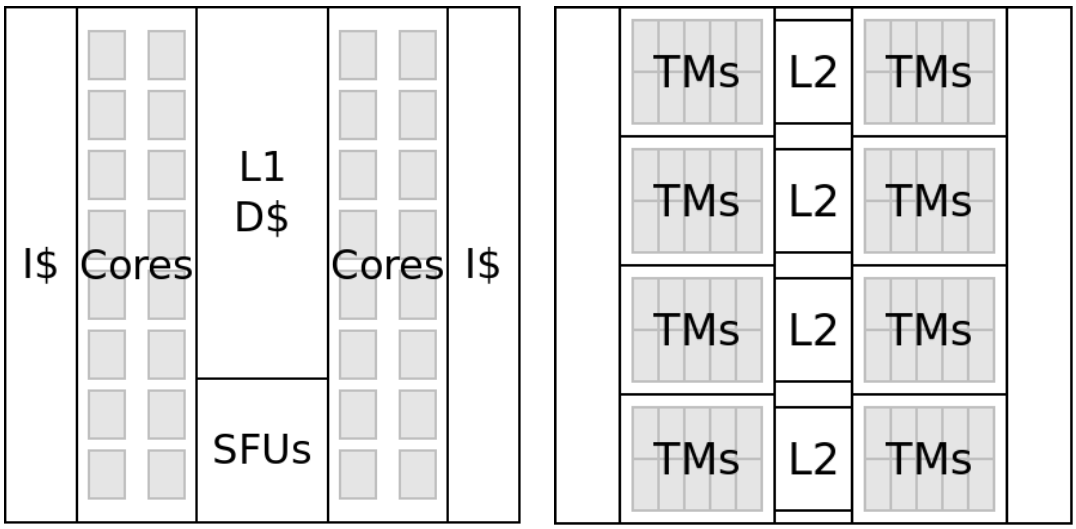


Figure 2.3: Potential TM and Multi-TM Chip Floor Plans. (a) TM Layout of 32 TPs and Shared Resources. (b) Chip with Multiple TMs Sharing L2 Caches.

Each shared 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. It is important to note that it is a good thing when the threads in a TM get a little out of sync with each other because it reduces the pressure on the shared functional units in any given cycle.

Each TP controls the execution of one ray-thread at a time. 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. Results in Chapter 5 show that with 32 TPs per TM, 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 [47] and a path tracer coded in a C-like language [46]. This metric can be considered roughly equivalent to the SIMD efficiency reported in similar studies using SIMD architectures with one important distinction. Often SIMD efficiency is inflated by allowing speculative execution to continue since the units would be in use anyway while that computation will end up being thrown away and not used in the generation of the final image.

2.2.3 Top Level Chip

Our overall chip model (Figure 2.3(b)) is a die consisting of number of blocks, each with an L2 cache, an interface to off-chip memory and a number of TMs with multiple TPs in each. Due to the low communication requirements of the threads, each TP 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 banked similarly to the L1 cache to allow parallel accesses from the L1 caches of the many TMs on the chip. A number of miss status holding registers (MSHR) are provided per TM and both the number of banks and number of MSHRs are parameterized in the simulator.

2.2.4 Functional Units

Functional units are added to the simulator in a modular fashion, allowing support for arbitrary combinations and types of functional units and instructions. This allows very general architectural exploration starting from our basic thread-parallel execution model. The clock rate is set to a conservative 500 MHz or 1 GHz clock depending on the particular TRaX design. The clock rates were chosen based on the latencies of the functional units that were synthesized using Synopsys Design Compiler and DesignWare libraries [53] and well characterized commercial complementary metaloxidesemiconductor (CMOS) cell libraries from Artisan [54] for 130nm and 65nm. Custom designed function units such as those used in commercial GPUs would allow this clock rate to be increased, or allow for increased energy efficiency.

2.3 Example TRaX Architectures

Due to the flexible nature of the TRaX architecture, and the large number of potential configurations, I only explore three primary configurations of TRaX architectures in this dissertation. These three designs are: *TM focused*, *a high-power design*, and a *low-power embedded design*. Each configuration is presented in greater detail including results and analysis in Chapter 5. In each case, the goals for analysis are different, and therefore the choices made in configuring the architecture are distinct. Focusing on the TM design first allows for the isolation of some higher level system considerations in order to perform an interesting analysis of a limited design space. The second design leverages the discoveries from the first and examines scaling to a large number of TMs in order to maximize ray throughput for a large power and area constraint. Finally, a scaled down design with fewer TMs is examined in order to evaluate the fitness of our techniques for mobile and low-power designs with much greater power constraints. Each of these designs is useful for the architect

to consider, and the final two represent proposals for potential commercial ray tracing hardware designs.

2.4 Conclusion

MIMD throughput architectures are a natural fit for ray tracing because of the divergent branching regularly found in acceleration structure traversal. Within the general TRaX framework discussed in this chapter, there is room for architectural customization and design exploration to tune the architecture for ray tracing. In order to evaluate the large number of parameters, it is useful to first simulate the expected behavior of the architecture before expending the large hardware design effort required to tape out and fabricate a single finished design. To this end, Chapter 3 describes the TRaX simulation framework in detail and enumerates the large number of parameters available in simulation. Furthermore, Chapter 4 discusses the programming model and compiler support provided for TRaX while Chapter 5 describes the architectural simulations and presents the results from those simulations.

CHAPTER 3

TRAX SIMULATION

The architecture described in Chapter 2 has been implemented as a cycle-accurate simulator, called **simtrax**. A cycle-accurate simulator simulates the behavior of an architecture one cycle after another. Other cycle-accurate simulators have been widely used for architectural research, such as Simplescalar [55]. One benefit of cycle-accurate simulation is that we can fairly accurately measure performance of the system as long as the models for each component are reasonable without incurring the huge cost of fabricating the architecture. In addition, the simple nature of modifying the software of the simulator allows the exploration of many different and closely related architectures in a short time and at low equipment cost. Furthermore, cycle-accurate simulation allows for straightforward instrumentation to gather detailed information about the behavior of various components and the primary causes of slowdown. Other methods for architectural simulation include functional simulation, trace-driven simulation, and full system simulation. While our simulator also requires the correct functionality, a functional simulator skips many of the details of the simulated hardware in order to speed up the completion of the simulation at the expense of cycle-accurate behavior. Trace-based simulators are often used for architecture research, but they ignore the possible change in program execution based on the performance of the architecture, which is exacerbated by the many parallel threads of execution in TRaX simulations. A full system simulation would include all levels of the system in the simulation, and while simtrax models many elements of the architecture and system accurately, some parts of the system are abstracted out, such as using an average access time for DRAM, and ignoring the setup time of transferring the initial dataset into the memory. This chapter describes the details of how the simtrax simulator works.

An overview of the simulation flow can be found in Figure 3.1. The simulation begins by reading in a hardware configuration file describing some of the hardware features while other parts of the configuration are specified at the command line. The initialization stage instantiates the pieces of the hardware, including setting the timing based on the configuration, and initializing the caches and memories. As part of the initialization, a model is loaded into memory along with an acceleration structure as specified by the parameters and model files. The main loop of simulation consists of having the clock rise in each of the functional units, with an issue unit handling the shared functional unit arbitration, followed by a clock fall that resets the things that need to be reset or cleared on each cycle. As each thread reaches the end of execution, the thread calls the HALT instruction and the simulator waits until all the threads have reached a HALT instruction to complete execution. Finally, the image stored in the frame buffer at the completion of execution is written to an image file to verify correct functionality, and the gathered data are written to one or more text files for analysis.

3.1 Simulation Configuration

When the TRaX simulator is run, it requires that a hardware configuration file be supplied describing the hardware units that will be simulated. For each functional unit, such as the multiplier, adder, floating point multiplier, and floating point adder, a quantity is specified to be instantiated per TM in the simulation, along with a latency of execution for each functional unit. These units are assumed to be fully pipelined, meaning a new instruction can be issued on each cycle for each unit instantiated. Additionally, area and power numbers can also be specified in the hardware configuration file, allowing the simulator to accurately report the area used by the configuration simulated. While the power numbers can be specified, they are not yet used by the simulator to report power consumption.

The hardware configuration file also contains the L1 and L2 cache sizes, the number of banks for each, and an access latency as well. Furthermore, the area and power numbers can also be included for these units, allowing the simulator to once again add the die area impact of the caches based on the simulation. The global

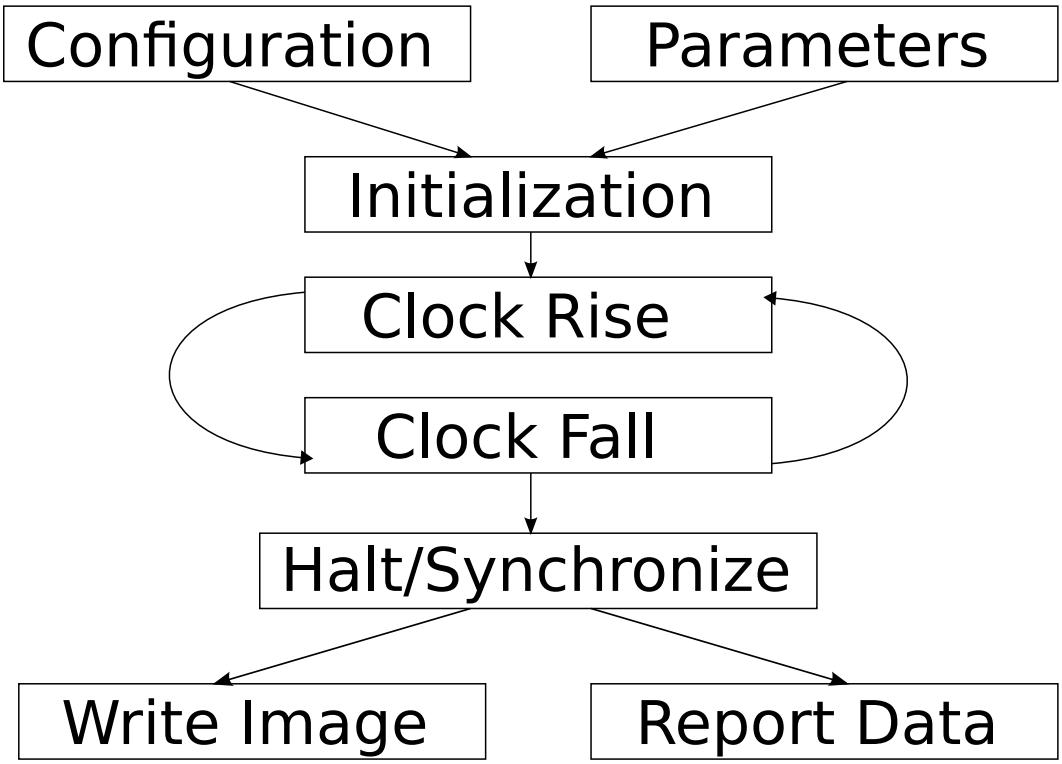


Figure 3.1: Simulator Overview

chip memory also should be specified in the hardware configuration in size and access latency. As the memory is assumed to not be held on-die, the size and power usage for the memory are not expected or considered by the simulator. An example hardware configuration file can be seen in Figure 3.2.

3.2 Simulation Parameters

A large number of command line options are available at run-time for the TRaX simulator. Table 3.1 lists the most relevant simulation parameters and their default values. The simulator options allow for simulation of a large number of different hardware configurations without recompiling the simulator itself. In particular, it has been most useful and interesting to vary the following parameters and configuration options:

- Number of threads in a TM
- Number of TMs sharing an L2 data cache

```
FPADD 1 4 3311 0.719
FPMIN 1 32 721 0.13
FPCMP 1 32 0 0.13
INTADD 1 32 450 0.101
FPMUL 1 4 10327 3.15
INTMUL 1 2 9184 2.97
FPINV 8 1 57936 45.55
CONV 1 32 0 .001
BLT 1 32 0 0.13
BITWISE 1 32 450 0.101
SPHERE 40 4
DEBUG 1 100
L1 1 8192 8 3 1490540
MEMORY 100 536870912
L2 2 8 16 3 0
```

Figure 3.2: Example Hardware Configuration File

- Number of L2s on a chip
- Number of each functional unit in a TM
- Execution latency of each functional unit
- Number of instruction caches, and banks for each instruction cache
- Capacity, latency and number of banks for each L1 and L2 data cache
- Whether the L1 and L2 data caches are on or off
- Image size (height and width)
- Model, view, light, and other rendering parameters

3.3 Simulation Initialization

Simtrax starts by checking the command line arguments and the hardware configuration file to see how the threads, cores, caches and other hardware components ought to be initialized. Once these hardware features are setup and initialized with the given values, the memory must be set up to contain the global data values that

Table 3.1: Simulation Parameters

Option	Argument	Default	Description
–width	[number]	128	width of image in pixels
–height	[number]	128	height of image in pixels
–num-regs	[number]	128	number of registers per thread
–num-globals	[number]	8	number of global registers
–num-thread-procs	[number]	4	number of TPs per TM
–threads-per-proc	[number]	1	number of hyperthreads per TP
–simd-width	[number]	1	SIMD issue width within a TM
–num-cores	[number]	1	number of cores (Thread Multiprocessors)
–num-l2s	[number]	1	number of L2s
–simulation-threads	[number]	1	number of simulator threads
–l1-off		false	turn off the L1 data cache
–l2-off		false	turn off the L2 data cache
–stop-cycle	[number]	none	cycle to halt execution
–config-file	[file]	none	config file name
–view-file	[file]	none	view file name
–model	[file]	none	model file name (.obj)
–far-value	[number]	1000	far clipping plane (rasterizer only)
–light-file	[file]	none	light file name
–output-prefix	[prefix]	out	prefix for image output
–image-type	[suffix]	png	type for image output
–ray-depth	[number]	1	depth of rays
–epsilon	[number]	1e-4	small number used for various offsets
–num-samples	[number]	1	number of samples per pixel
–print-instructions		off	print contents of instruction memory
–no-scene		off	turns off loading scene
–issue-verbosity	[number]	0	level of verbosity for issue unit
–num-icaches	[number]	1	number of icaches in a TM (power of 2)
–num-icache-banks	[number]	32	number of banks per icache
–load-assembly	[file]	none	assembly file to execute
–print-symbols		off	print symbol table generated by assembler
–triangles-store-edges		off	set flag to store 2 edges instead of 2 verts

will be required by the ray tracer. The memory can be loaded with a memory image from a previous run, which is just a file containing the values of each memory address. More commonly, the memory loader reads in the scene data files provided, creates a bounding volume hierarchy (BVH) or other acceleration structure, and then loads all of those values into the memory along with pointers to structures such as the BVH, the light, and the background color. An example of the memory layout based on the conference scene and BVH build for that scene can be found in Table 3.2. Pointers are stored in memory to allow the number of materials and triangles to be variable while allowing the ray tracer to still be able to locate them at run time. The frame buffer is the region of memory where the final colors are stored for each pixel. The materials, scene and triangle data store the information necessary to trace rays around the scene based on the camera and light information. The background color is used for rays that do not intersect with any objects in the scene.

The final step of the initialization is to spawn a number of software simulation threads that will each be in charge of a group of TMs. Simtrax does its best to find a balanced workload division, putting at most one extra TM on the heavily loaded simulation threads than the ones with a light load. Note that there are two kinds of threads discussed in this chapter. **Simulation threads** are threads of the simulator program that are each tasked with simulating a number of TMs that each contain a number of simulated threads. **Simulated threads** are actual TRaX threads that exist within the simulated architecture and are grouped into TMs which are tiled on a TRaX chip depending on the configuration. The smallest possible simulation consists of a single simulation thread of one TM with one simulated thread. Larger simulations involve multiple simulation threads (typically around the number of cores available on the simulating machine) simulating tens to hundreds of TMs, each with 16 or more simulated threads.

3.4 Execution

After initialization, simtrax begins executing on the first cycle. A cycle consists of a clock rise and a clock fall being sent to each of the functional units in each of the TMs. These events allow the functional units to perform a number of different

Table 3.2: Example Memory Layout

Item	Memory Word Address	
	Decimal	Hexadecimal
Constants and Pointers	0	0x00000000
Frame Buffer	38	0x00000026
Materials	16422	0x00004026
Scene	17523	0x00004473
Triangles	2147136	0x0020c340
Camera	13837940	0x00d32674
Background Color	13837975	0x00d32697
Light	13837978	0x00d3269a
Memory End	13839005	0x00d32a9d

functions, the most important of which is the issue unit. Each TM has an issue unit that is in charge of determining which threads in that TM will issue. On each cycle, within each simulation thread, the TM with the longest number of stall cycles among its threads is chosen to attempt to issue first in order to ensure fairness of issue among the different TMs when considering shared resources, such as memory bandwidth. Within each TM, a similar fairness mechanism is used to attempt to issue from the thread with the highest stall time first. Experiments have shown that these fairness mechanisms are effective at balancing out the workload and allowing threads and TMs to issue approximately the same number of instructions as each other, even when experiencing heavy resource contention.

As an issue unit attempts to find threads to issue, it will loop through all of the threads in the corresponding TM. For each thread, it checks the instruction cache to verify that there would be no bank conflicts in attempting to fetch the instruction. It then checks all of the available functional units as described in the initialization of the system, whether they are shared or private units. Once an appropriate unit is found that is capable of accepting the next instruction for that thread, the issue unit checks if the register values required to issue that instruction will be available in time, whether stored in the physical registers, or available through register forwarding. Upon finding an instruction with a matching unit and available source registers, the issue unit passes all of the important data to the functional unit to attempt to issue. Most simple functional units will succeed and schedule a write to the appropriate

registers to be completed on the cycle corresponding to the execution time of that unit. Functional units have the ability to generate their own stalls if there is something that would prevent an instruction from issuing even if it appears to be ready otherwise. For example, an L1 cache will often encounter a bank conflict when attempting to issue, and stall the thread attempting to issue even when the required register values are available. Assuming the instruction is able to issue, the results of the success are scheduled, the success is recorded, and the program counter is incremented to the next instruction. In the case of a failure to issue, the resources are not consumed, but the failure to issue is recorded, along with as much information as possible about the reason for the failed issue. When a branch occurs, the instruction in the delay slot is forced to execute before the program counter takes the branch.

The final step for the cycle is to proceed to the next cycle. Since the simulation threads are run in parallel, the simulator enforces a global barrier on all simulation threads at the end of each cycle. This barrier means that all TMs within any given L2 will be on the same cycle, and all TMs in the system will be on the same cycle. A looser constraint could be used to speed up simulation times as a global barrier can be quite expensive, however, the simulation accuracy would be highly degraded by allowing different TMs to be thousands of cycles apart.

3.5 End of Simulation

The main loop of most TRaX programs uses the atomic increment to assign tasks among a number of tasks to be completed. Once the number returned by the atomic increment instruction is greater than the total number of tasks to be completed, the thread will branch to a `HALT` instruction. This `HALT` instruction causes the thread to no longer attempt to issue in the issue unit. Furthermore, each thread is checked to see if it is halted, and when all threads in a TM have halted, that TM is considered halted. `HALT` should only be called for a thread once all work from atomic increment assignments has been completed. Once all TMs have halted, the simulation stops and the frame is considered complete. For animations, the triangles will be moved by the simulator, the BVH will be rebuilt, and the simulation threads will all start at the beginning again. However, for the simulations in this dissertation, animations

were not used, therefore the simulation finishes after a single frame is rendered. This simulation methodology is accurate assuming the ray tracer has little to no setup or other overhead that would not have to be repeated from frame to frame. A side effect of enforcing the global synchronization at the end of a frame is that threads that finish the frame before the final thread reaches the end sit idle for some number of cycles. In practice, the number of idle halted cycles is insignificant to the overall system performance on scenes of a reasonable size.

Once all threads are found to be halted and the simulation completes, simtrax generates an image representing the values stored in the frame buffer locations. In addition, all of the issue statistics are reported along with the framerate given the current system configuration. Furthermore, area for the hardware configuration specified is reported, and hit rates for the caches in the system as well as any bank conflicts when accessing the caches. The final piece of data reported are utilization numbers for each of the functional units in the system, which is useful for finding which functional units are highly constrained and for computing power consumption of the system.

3.6 Example Simulation

This section provides and example of how the simtrax simulator can be used to explore various hardware configurations through the data gathered. The following example shows how, through the use of shared functional units, area can be saved from each TM, resulting in the potential to include more threads on the same sized device. The simulation parameters used are held constant for both tests except for a change in the number of functional units, as can be seen in Table 3.3. Holding most of the parameters constant allows for simple examination of how certain changes will affect system performance, however, results could potentially vary given different values for some of the parameters that have been held constant in this example. For instance, if the instruction caches were under-provisioned to only allow one thread to issue at a time, then it would be more beneficial to only have one thread per TM. Both tests here use a simple ray tracing application and render the conference scene at 512×512.

Table 3.3: Example Simulation Configurations

Parameter	Config 1	Config 2
Threads per TM	32	32
Instruction Cache	4 x 16 banks	4 x 16 banks
L1 Data Cache	16 kB, 4 banks	16 kB, 4 banks
L2 Data Cache	512 kB, 32 banks	512 kB, 32 banks
Standard Functional Units	32	32
Special Functional Unit	1	1
FPMUL, FPADD, INTMUL	32	8
TM Size	1.11	0.44

As can be seen from the functional unit utilization reported by the simulator in Figure 3.3, the functional units are underutilized and over-provisioned in the first example. The numbers for module utilization reported in the figure represent the percentage of possible issues to that unit that were actually used on average over the duration of the simulation. The sizes are reported in mm^2 and the FPS is determined by the total clock cycles of the longest running TRaX thread. Despite the large difference in the number of units used in these two examples, the overall system performance remains quite stable with an overall frame rate reduction of less than 1%. These simulations were run using the following command with the only variations coming in the modification of the config file as detailed in Table 3.3:

```
./simtrax --num-regs 36 --num-thread-procs 32 --threads-per-proc 1 \\
 --num-cores 2 --width 512 --height 512 --config-file configs/example.config \\
--num-icaches 4 --num-icache-banks 16 --num-l2s 1 \\
--view-file views/conference.view --model test_models/conference.obj \\
--light-file lights/conference.light --load-assembly assembly/prog5_sol.s
```

The results and architectural explorations reported in Chapter 5 use the same simulation technique and use the reported statistics for comparison of different configurations.

```
Configuration 1                          | Configuration 2
Module Utilization                       | Module Utilization
                                         |
           FP AddSub:    8.01            |          FP AddSub:   32.04
           FP MinMax:    6.95            |           FP MinMax:    6.94
          FP Compare:    4.26            |          FP Compare:    4.26
          Int AddSub:   19.55            |          Int AddSub:   19.54
              FP Mul:    8.49            |              FP Mul:   33.97
             Int Mul:    0.02            |             Int Mul:    0.10
          FP InvSqrt:    5.87            |          FP InvSqrt:    5.87
              FP Div:   10.04            |              FP Div:   10.05
     Conversion Unit:    0.02            |     Conversion Unit:    0.02
                                         |
Core size: 1.2891                        | Core size: 0.6134
L2 size: 2.6500                          | L2 size: 2.6500
1-L2 size: 2.6500                        | 1-L2 size: 2.6500
2-core chip size: 5.2282                 | 2-core chip size: 3.8768
FPS Statistics:                          | FPS Statistics:
Total clock cycles: 49599916             | Total clock cycles: 49603956
  FPS assuming 1000MHz clock: 20.1613    |   FPS assuming 1000MHz clock: 20.1597
```

Figure 3.3: Example Functional Unit Utilization

CHAPTER 4

PROGRAMMING TRAX

TRaX can be thought of as a general purpose CPU from the point of view of the programmer that just happens to run a number of copies of its program concurrently. Parallelism is supported through a few limited primitives, and the fact that every thread is loaded with the same program on startup and executes from the same point. The programmer benefits from thinking about the architecture from the SPMD model, where work assignments and thread specialization are provided within the same program. The SPMD programming model allows a single program to control a large swarm of small computers.

The real benefit to TRaX is that the programmer does not need to think as much about how the work will be divided among the hardware threads since there is much less of a penalty for divergent branches among threads than other programming models (e.g., SIMD). TRaX memory is loaded at startup by the host CPU with the relevant scene data (including acceleration structure) and the programmer must take this into account when writing a program for TRaX. In order to support a variety of ray tracers, a variety of constants and pointers to important data structures are placed at known locations towards the front of the memory space. Memory coherence is not provided except through explicit programmer control by evicting cache lines that are expected to have been modified by other threads. By pinning data to the L2 cache through explicit evictions, coherence can be provided to all of the L1s that share the L2 in a cluster. While this limits TRaX's ability as a general purpose compute engine, it allows for the memory system to ignore the need for expensive invalidate messages. The TRaX instruction set architecture (ISA) provides all of the standard mathematical and logical operations that are common to reduced instruction set computer (RISC) architectures, including register and

immediate versions of most arithmetic and bitwise instructions. In addition, a number of nonstandard instructions have been added to support parallelism, including an atomic increment instruction to assist with work assignments. The programming environment encapsulates some intrinsics to provide direct access for the programmer to use the special functionality available in the TRaX ISA.

In designing the programming model for TRaX, the primary motivation was a desire to make parallel ray tracing code as easy to develop as possible. Due to the naturally parallel nature of naive ray tracing algorithms, we decided to target a simple ray tracer with a fixed-depth ray tree rather than a fully recursive ray tracer, or one using packets. Packets have been shown to effectively increase SIMD efficiency, particularly on CPU-style SIMD architectures [56, 41]. Another approach to SIMD ray tracing is the use of a multi-BVH [57, 58], which allows for multiple traversal decisions to be processed in parallel. In contrast, TRaX is designed to naturally allow for the behavior that ray tracing exhibits naturally, that of unpredictable branching among a group of rays. These SIMD acceleration techniques work best for primary viewing rays and sometimes shadow rays, while the major visual advantages in ray tracing come from the less predictable secondary rays. As TRaX is designed to use only independent threads, the benefits from packets and multi-BVHs would be less than for SIMD architectures. The result of these decisions is that code for TRaX is written as a single thread of execution, but with an atomic increment being used to retrieve new ray, pixel, or sample assignments in the outer loop. This allows the exact same code to run on every single thread, but for each thread to execute on independent final writeback positions in screenspace.

There are two fundamentally different compilation techniques presented here and used in this dissertation. The first uses a large register file to implement a stack as explained in Section 4.2. The second scheme, presented in Section 4.3, opts to reduce the size of the register file by using a small local memory to store the call frames. This allows the programming environment to set up the memory to correspond to the initial state at which the ray tracer should begin execution.

4.1 TRaX Helper Functions

In order to help programmers to more easily target the TRaX architecture with separate local and global memory spaces, we provide a number of helper functions, most of which use compiler intrinsics to generate special instructions that correspond to the operations that should be performed. Table 4.1 lists the most useful of these functions and describes how they are used and what each one does. A different header file is provided for the TRaX target as well as the native target used for rapid development. More about these two targets can be found in Sections 4.3 and 4.4. In order to more easily support both targets, and to allow the native target to function with the same global memory space as the TRaX target, programs written for TRaX must use `void trax_main()` instead of `int main()`.

4.2 Register Stack

The first backend for TRaX compilation does not use the standard LLVM [59] technique for stack management and instead opts to place stack data in the register file. In order to support register-based stacks, we provide instructions that allow for reading and writing the register file using one register value as a pointer to a register and another register value as an offset into the register file. When performing compilation for register stacks, the compiler attempts to keep as many values in registers as possible instead of pushing those values on the stack. As a result of using registers for the stack and keeping values in registers longer, the register stack compilation results in a much higher quantity of registers required in the architecture, sometimes exceeding even 256 registers in some cases.

4.3 LLVM Backend

In order to reduce the number of required registers in the architecture, we also have a backend for TRaX in the standard LLVM infrastructure. This backend uses LLVM 2.9, and is based on the Microblaze backend, since much of the instruction set was similar, and it allowed for leveraging an existing backend to ensure compatibility with LLVM. Microblaze [60] is a simple reduced instruction set computer (RISC) architecture built as a soft-core microprocessor available from Xilinx for synthesis and use on their field programmable gate arrays (FPGA). This is useful as a target because

Table 4.1: Helper Functions

Function	Return	Arguments	Description
loadi	int	int base, int offset	Load global address as an integer
loadf	float	int base, int offset	Load global address as a float
storei	void	int value, int base, int offset	Store an integer to global address
storef	void	float value, int base, int offset	Store a float to global address
trax_getid	int	int value	Returns an id based on value 1=threadID,2=TMID,3=L2ID
atomicinc	int	int global_reg	Returns value of global_reg and adds 1 after
global_reg_read	int	int global_reg	Returns value of global_reg
min	float	float left, float right	Returns the minimum of left and right
max	float	float left, float right	Returns the maximum of left and right
invsqrt	float	float value	Returns $\frac{1}{\sqrt{value}}$
trax_rand	float		Returns a random value between 0 and 1
GetXRes	int		Returns screen width in pixels
GetYRes	int		Returns screen height in pixels
GetInvXRes	float		Returns inverse screen width in pixels
GetInvYRes	float		Returns inverse screen height in pixels
GetFrameBuffer	int		Returns the address of the frame buffer
GetBVH	int		Returns the address of the BVH
GetMaterials	int		Returns the address of the materials
GetCamera	int		Returns the address of the camera
GetBackground	int		Returns the address of the background color
GetLight	int		Returns the address of the light
GetStartTriangles	int		Returns the address of the triangles
GetNumTriangles	int		Returns the number of triangles
GetThreadID	int		Returns the thread ID
GetCoreID	int		Returns the TM ID
GetL2ID	int		Returns the L2 ID

different architectural configurations are available for Microblaze, so the backend already has to support hardware variations. The compiler front end that is used to generate LLVM bytecode is `llvm-gcc`. A benefit to using this workflow is that branch delay slot filling was implemented in the Microblaze backend and therefore the TRaX backend also includes it. Previous to the current compiler, we used a custom compiler backend that would manage call stacks in registers, but it had become unruly and difficult to update. This newer backend was developed partially for use in a course taught in the fall of 2011, to expose more students to the TRaX architecture. In this class, a group of around 8 students were able to implement a basic path tracer for TRaX, followed by a variety of additional projects. A few of the class projects include beam tracing, photon mapping, and metropolis light transport, showing that TRaX can be used for more advanced rendering techniques than those that will be presented here.

4.4 Functional Simulation

While the TRaX simulator is a useful tool for analyzing the behavior of the TRaX architecture, any cycle accurate simulator that gathers many useful data about performance will be much slower than the hardware it simulates. In order to keep programmers from having to wait 4 or more hours between compiling their code and getting a result back to see if it was functional, a functional simulator was desired. Since the TRaX programming model is a naive single thread that gets assignments from atomic increments, a custom native implementation of the functionality of the TRaX intrinsics is provided that could be optionally included when compiling any TRaX code. In order to have the global memory behave the same way in functional simulation, we leverage the existing memory loading functionality from the TRaX simulator. The result is that a programmer can compile code for both the TRaX target, and a functional simulation target at the same time and use the functional simulation target while developing a test for the general functionality of the code with a much more rapid feedback loop. The examples we provided in the class included a Makefile that provides both build targets. A selection of these examples is also published with the public release of `simtrax`.

4.5 Example TRaX Programs

Figure 4.1 shows a simple gradient fill program written for TRaX. This example shows how the `atomicinc()` intrinsic should be used to allow thread assignments to be distributed among the large number of threads in the system. In the gradient example, the workload is naturally balanced among the pixels, however, in other cases it is important to remember that work assignments should be small enough to allow the system to dynamically balance the workload across all the threads in the system. The other main interesting part of this example is the usage of built-in functions to load the screen height and width from global memory and the use of the `storef()` intrinsics to write to the frame buffer in global memory.

Due to the code length of a full path tracer, only the main function of the ray tracer is included in Figure 4.2. The important differences when compared with the simple gradient example include many more pointers being loaded from global memory in the set up. It should be noted that the ray generation function itself does not require

```
#include "trax.hpp"

// Only include stdio for printf on the non-trax version
#if TRAX==0
#include <stdio.h>
#endif

// Utility function to store a color at a pixel offset in the frame buffer
inline void StorePixel(const int &fb, const int &pixel, const float &r,
                       const float &g, const float &b) {
  storef(r, fb + pixel*3, 0);
  storef(g, fb + pixel*3, 1);
  storef(b, fb + pixel*3, 2);
}

void trax_main()
{
  // Load the pointer to the frame buffer in global memory
  int framebuffer = GetFrameBuffer();

  // Load the screen size
  int width = GetXRes();
  int height = GetYRes();

  for (int pixel = atomicinc(0); pixel < width * height;
       pixel = atomicinc(0)) {
    // Store a color based on screen location
    int i = pixel % height;
    int j = pixel / width;
    StorePixel(framebuffer, pixel, (float)j/height, (float)i/width, 0.0f);
  }

  // Conditional to only execute the following on the CPU version
#if TRAX==0
  printf("Thread %d one drawing.\n", trax_getid(1));
#endif
}
```

Figure 4.1: Gradient Fill Example

```
int trax_main(){
  BoundingVolumeHierarchy bvh(loadi(0, 8)); // Set BVH pointer
  PointLight light = loadLightFromMemory(loadi(0, 12)); // Load light
  int xres = loadi( 0, 1 ); int yres = loadi( 0, 4 );  // Load screen size
  float inv_width = loadf(0, 2); float inv_height = loadf(0, 5);
  int start_fb = loadi( 0, 7 ); // Frame Buffer write location
  int start_matls = loadi(0, 9); // Materials pointer
  int num_samples = loadi(0, 17); // Load constants
  int max_depth = loadi(0, 16);
  Image image(xres, yres, start_fb); // Configure write interface
  PinholeCamera camera(loadi(0, 10)); // Load camera

  for(int pix = atomicinc(0); pix < xres*yres; pix = atomicinc(0)) {
      // Compute x and y based on work assignment
      int i = pix / xres;
      int j = pix % xres;
      Color result(0.f, 0.f, 0.f); // Background color
      float x = 2.0f * (j - xres/2.f + 0.5f)/xres; // center in pixel
      float y = 2.0f * (i - yres/2.f + 0.5f)/yres; // center in pixel
      for(int i=0; i < num_samples; i++) { // Number of samples per pixel
          Color attenuation(1.f, 1.f, 1.f); // start with full attenuation
          int depth = 0; Ray ray;
          if(num_samples == 1)
            camera.makeRay(ray, x, y); // Create ray
          else { // Random sample within the pixel
              float x_off = (trax_rand() - 0.5f) * 2.f;
              float y_off = (trax_rand() - 0.5f) * 2.f;
              x_off *= inv_width;
              y_off *= inv_height;
              camera.makeRay(ray, x + x_off, y + y_off); // Offset ray
            }
          while(depth++ < max_depth) { // stop when max_depth is reached
              HitRecord hit(100000.f);
              Color hit_color;
              Vector normal;
              Vector hit_point;
              bvh.intersect(hit, ray); // intersect with scene
              // Compute color contribution for this ray
              result += shade(hit, ray, bvh, light, start_matls, hit_point,
                              normal, hit_color) * attenuation;
              if(!hit.didHit())   break; // stop if nothing hit
              ray.org = hit_point;
              ray.dir = randomReflection(normal); // reflect on hit
              attenuation *= hit_color; // path loses energy on each bounce
            }
        }
      result /= num_samples; // normalize result based on sample count
      image.set(i, j, result); // write color to image
  }
}
```

Figure 4.2: Path Tracer Example

any call to global memory since it uses the camera values that are stored locally at each thread as well as the pixel assignment currently being processed by the thread. The `bvh.intersect()` function performs all of the global loads required in order for the thread to determine which object should be considered for shading from among all of those in the scene. The `shade()` function uses the hit information as well as the light and material properties to compute the color that should be accumulated into the frame buffer for the current ray. This path tracer then computes a random direction to sample for the following ray, and attenuates that ray's contribution to the final pixel color. The final value to be written to the frame buffer is accumulated locally in thread local memory and registers, only to be written back a single time per color channel for each pixel. For an image of 512×512 pixels, only 786,432 writes will be performed across the entire system per frame. This small number of global memory stores allows the TRaX memory system to write around the cache for all writes, thereby eliminating all cases where frame buffer writes would cause other data values to be evicted from the caches.

4.6 Ray Tracing Software

In the results presented in this dissertation a number of iterations of essentially the same ray tracer were used. Some of the test programs are written directly in assembly language. Others are written in a higher level language designed for our architecture. Many TRaX programs use header based extensions inspired by the RenderMan shading language [61] to allow for ease of writing a ray tracing application. Some high level code was compiled using a custom LLVM backend that interprets LLVM bytecode independent of the LLVM tool flow, while other high level code was compiled using an LLVM backend functioning within the LLVM tools. The results in Section 5.1 use both hand coded assembly as well as the custom LLVM backend outside of the LLVM tools. Section 5.2 uses the LLVM backend outside of the LLVM tools, while Section 5.3 uses the LLVM backend inside of the LLVM tools.

For Section 5.1, two different ray tracing systems were used as follows.

- **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 thread processor assembly language.

- **Path Tracer:** This application is written in C++ with TRaX library extensions. It computes global illumination in the scene using a single point light source and using Monte-Carlo sampled Lambertian shading [6].

4.6.1 Shading Methods

All of the ray tracers presented in this work implement two of the most commonly used shading methods in ray tracing: simple diffuse scattering, and Phong lighting for specular highlights [62, 63]. 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 especially increase complexity, as they involve taking an integer power, as can be seen in the Blinn-Phong lighting model [64]:

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

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 size of the specular highlights. The i terms are the intensities of the ambient, diffuse, and specular components of the light sources.

4.6.2 Procedural Texturing

We also implement procedural textures in our ray tracer. 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 [65, 66]. 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 precomputed 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 [65]) or a quintic interpolant (for improved Perlin noise [67]) 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% 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 [68, 69]. 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 [70]. An example of a view of the Sponza scene rendered with a TRaX implementation of Perlin noise-based textures can be seen in Chapter 5.

4.6.3 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 [6]. 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 towards 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 (BRDF) [71, 72]. 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 degrees (the sample is always taken in the exact direction of the normal) to 180 degrees (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 Figure 4.3.

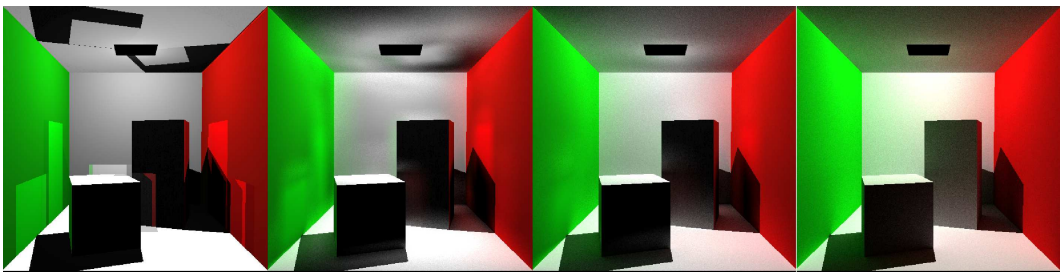


Figure 4.3: The Cornell Box Scene Showing the Visual Change as the Sampling Angle Increases in our Path Tracer. Starting on the Left: 0 Degrees, 30 Degrees, 60 Degrees, and 180 Degrees on the Right.

CHAPTER 5

EVALUATION OF RAY TRACING ON TRAX[1][2][3]

In order evaluate a variety of configurations of the TRaX architecture, three related but distinct explorations will be presented in this chapter. The three stages of exploration of the TRaX architecture are as follows:

- **TM Exploration:** A TM centric view assuming perfect chip-wide scaling. The goal is to discover which kind of TM configurations are in the realm of reasonable to be used as a basis for later studies.
- **Chip Exploration:** Based on the same TM principles, adding chip-wide details in the simulation results. The goal is to fill out enough details in the chip-wide system to report the expected performance of a high-performance ray tracing processing unit.
- **System on Chip:** A full TRaX chip designed as a low-power IP block that could be used in a mobile SoC. The goal of this design is to show the potential for ray tracing on current, and future mobile devices.

The primary metric used in this work to measure the effectiveness of a ray tracing system is the number of rays per second (RPS, or MRPS for millions, a common

[1]This chapter is modified with permission from J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A multicore hardware architecture for real-time ray tracing," *IEEE Transactions on Computer-Aided Design*, vol. 28, no. 12, pp. 1802 – 1815, Copyright ©2009 IEEE.

[2]This chapter is modified with permission from D. Kopta, J. Spjut, E. Brunvand, and A. Davis, "Efficient MIMD architectures for high-performance ray tracing," in *IEEE International Conference on Computer Design (ICCD)*, Copyright ©2010 IEEE.

[3]This chapter is modified with permission from J. Spjut, D. Kopta, E. Brunvand, and A. Davis, "A mobile accelerator architecture for ray tracing," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, Copyright ©2012.

magnitude for these studies). Estimates for scenes with moderately high quality global lighting range from 4M to 40M rays/image [45, 2, 44]. At even a modest real-time frame rate of 30Hz, this means that performance in the range of 120 to 1200 MRPS will be desirable. Throughout this chapter, the simulated clock rates are based on timings given by synthesis of actual functional units. When 500 MHz is used, it is because the cell libraries were for older processes or designed for lower power, while the 1 GHz clock rates assume newer cell libraries with a variety of low power and high performance cells determining the functional unit latency. Clock rates are reported along with each set of results.

5.1 Design of a Threaded Multiprocessor

The objective of the first evaluation is to explore the design configuration of a single TM. Issues include how many threads should be placed within a single TM, the sizes and configurations of the caches, allocation and sharing of functional units, and other features of the basic MIMD-SPMD thread multiprocessor. For the TM exploration, the number of threads in each TM was varied from 16 to 128 using powers of 2. These results show that with 32 thread processors per TM, 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 [47] and a path tracer coded in a C-like language [46]. Note that both of these ray tracers use an extended register file that is used for the program call stack, resulting in an estimated register file size of 128 registers per thread. In reality, the ray tracing application used more registers for this study, but we assume that a better compiler might be able to reduce register usage to only 128 including the stack registers.

The resulting multiple-thread TM can be repeated on a multi-TM chip because of the independent nature of the computation threads. Performance for the TM exploration is thus estimated by performing a simulation of a single TM, and extrapolating performance given linear scaling to the number of TMs that would be able to fit in the given chip size. We evaluate performance of the TM architecture using two different ray tracing applications: a recursive Whitted-style ray tracer [4, 5, 6] that allows us to compare directly to other hardware ray tracing architectures, and a path tracer [8, 73]

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 [46].

This work 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 for these experiments, so the performance we measure is for rendering a single frame. We have, however, explored the possibility of dynamic scene updating on the TRaX architecture using techniques such as tree rotations [74]. Results for tree rotations on TRaX show high issue rates at the expense of increased memory coherence traffic. Other studies have shown that it is also possible to rebuild slightly lower quality BVHs on GPUs [75, 76, 77]. These rebuilds should also work efficiently on TRaX-style architectures, and may be even more efficient with MIMD execution.

The test scenes used for this first exploration, seen in Figure 5.1 with some basic performance numbers in Table 5.1, exhibit some important properties. The Cornell Box (left) is important because it represents the simplest type of scene that would be rendered. It gives us an idea of the maximum performance possible with the TRaX ray tracing hardware. Sponza (middle), on the other hand, has over 65000 triangles and uses a BVH with over 50000 nodes. Also included is a version of the Sponza scene displaying our procedural noise texturing, the details of which can be found in the journal version of this work [48]. The Conference Room scene (right) is an example of a reasonably large and complex scene with around 300k triangles. This is similar to a typical video game scene from a few years ago. Some more complicated scenes have results on different configurations that can be found in Sections 5.2 and 5.3.

5.1.1 Multi-TM Chip

In order to focus this first exploration solely on TM performance, system wide features, such as the L2 cache and atomic increment, were not modeled explicitly, and instead were approximated by assumed hit rates (see Section 5.1.3 for more details). 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 twenty times the latency of L1 hits. While not entirely accurate, these approximations allow for quicker experimentation within the TM. Results based on explicit chip-wide simulations can be found in Section 5.2.

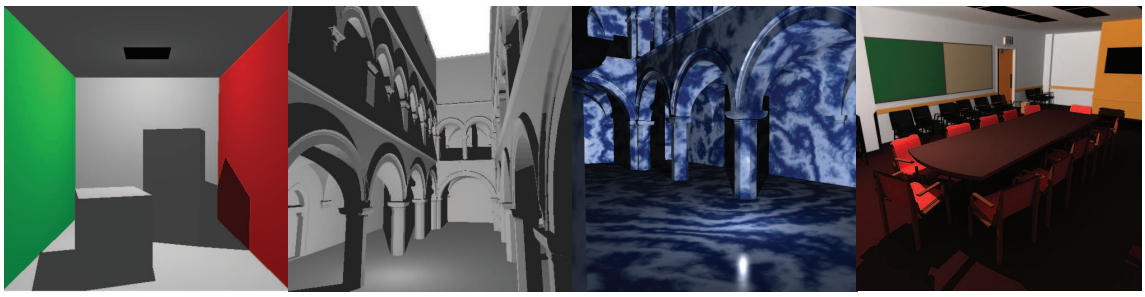


Figure 5.1: Test Scenes Rendered on Our TRaX Architectural Simulator

5.1.2 Whitted-Style Ray Tracer

This section describes a basic recursive ray tracer that provides us with a baseline that is easily compared to other published results, as described in Section 4.6. 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. The first iteration of the software would split the screen into 16×16 pixel squares and each thread would be assigned one tile to render, similar to a packet ray tracer. While this is a good idea for reducing the number of atomic increments, we found that it did not produce the best performance. Instead, for the results in this section, single pixels are assigned 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 since simultaneously executing threads are placed closer together in screen space. 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, resulting in interthread cache conflicts. The pixels are computed row-by-row straight across the image, though more sophisticated space filling methods such as a Z curve have the potential to further increase the ray tracing performance by contributing to memory access locality resulting in better cache hit rates. The important finding here is that, for threads sharing a data cache, simultaneous ray assignments should be as close together as possible to increase the cache line reuse of that data cache.

Table 5.1: Scene Data with Results for 1 and 16 TMs, Each with 32 Thread Processors, and Phong Shading Estimated at 500MHz

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

5.1.3 Design Exploration

For each simulation we render one frame in one TM 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, functional unit utilization, thread utilization, thread stall behavior, memory and cache bandwidth, memory and cache usage patterns, and total parallel speedup, as described in Chapter 3.

Nearly all hardware features of the TRaX architecture can be varied in our simulations, including clock rate, latency of all units, and quantity and organization of the hardware components. For this exploration, ray tracing code was executed on simulated TRaX TMs having between 1 and 256 thread processors, with issue widths of all function units except memory varying between 1 and 64 (memory was held constant at single-issue). Images may be generated for any desired screen size, though a size of 1024x768 is used for this study. The primary goal for the first design phase is to determine the optimal allocation of transistors to thread-level resources, including functional units and thread state, in a single TM to maximize utilization and overall parallel speedup. The study in Section 5.2 includes better memory models for memory and cache usage to feed the parallel threads (and parallel TMs at the chip level).

5.1.4 Functional Units

We first chose a set of functional units to include in our machine-level language, shown in Table 5.2. This mix was chosen by separating different instruction classes into separate dedicated functional units. We implemented our ray casting benchmarks using these available resources, then ran numerous simulations varying the number of threads and the issue width of each functional unit. All execution units are assumed to be pipelined including the memory unit. The area estimates for these functional units can be found in Table 5.3.

Table 5.2: 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 5.3: Area Estimates (Prelayout) for Functional Units Using Artisan CMOS Libraries and Synopsys. The 130nm Library is a High Performance Cell Library and the 65nm is a Low Power Cell Library. Speed is Similar in Both Libraries.

	Area (μm^2)	
Resource Name	130nm	65nm
2k×16byte cache (4 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

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, 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 are required by the 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. It would also be possible to include a custom noise function as a shared functional unit that would allow the rapid generation of gradient noise used for procedural texturing (see Section 4.6.2), though that is not the focus of this section.

5.1.5 Single TM Performance

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-TM results to multi-TM 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 an 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 Saarland RPU [40, 78] and achieve roughly the same clock frequency as the low power 65nm libraries.

Basic functional units, including register files and caches, were synthesized, placed and routed using Synopsys and Cadence 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 TMs 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 5.4 and Table 5.5 are limited to $150mm^2$ in order to allow room for the components that are unaccounted for.

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 TMs on a single chip at 130nm. At 65nm with 79 TMs 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 5.2 shows, for a variable number of threads in a single TM, 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. Figure 5.3 shows how TM performance varies as the number of issue ports is changed. We conclude that at least two ports are required to have acceptable performance, but any more than four ports are unnecessary.

Considering a 32 thread TM with 50% of the threads issuing each cycle, we have 16 instructions issued per cycle per TM. In the 130nm process, we fit 16 to 22 TMs 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 TMs 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 recent GPUs with many concurrent threads issue a theoretical maximum IPC of around 256 (128 threads issuing 2 floating point operations 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

Table 5.4: TRaX Area Estimates to Achieve 30 FPS on Conference. These Estimates Include Multiple TMs, but not the Chip-Wide L2 Cache, Memory Management, or Other Chip-Wide Units.

Threads/TM	TM Area mm^2		Single	# of	Die Area mm^2	
	130 nm	65 nm	TM FPS	TMs	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 5.5: Performance Comparison for Conference and Sponza Assuming a Fixed Chip Area of $150mm^2$, not Including the L2 Cache, Memory Management, and Other Chip-Wide Units.

Threads/TM	# of TMs		Conference FPS		Sponza FPS	
	130 nm	65 nm	130 nm	65 nm	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

25% range. While a detailed study of power consumption was not performed in this work, we expect the power consumption of TRaX to be similar to that of commercial GPUs.

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 2K lines (16-bytes 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 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. This is implemented as a four-bank direct mapped cache.

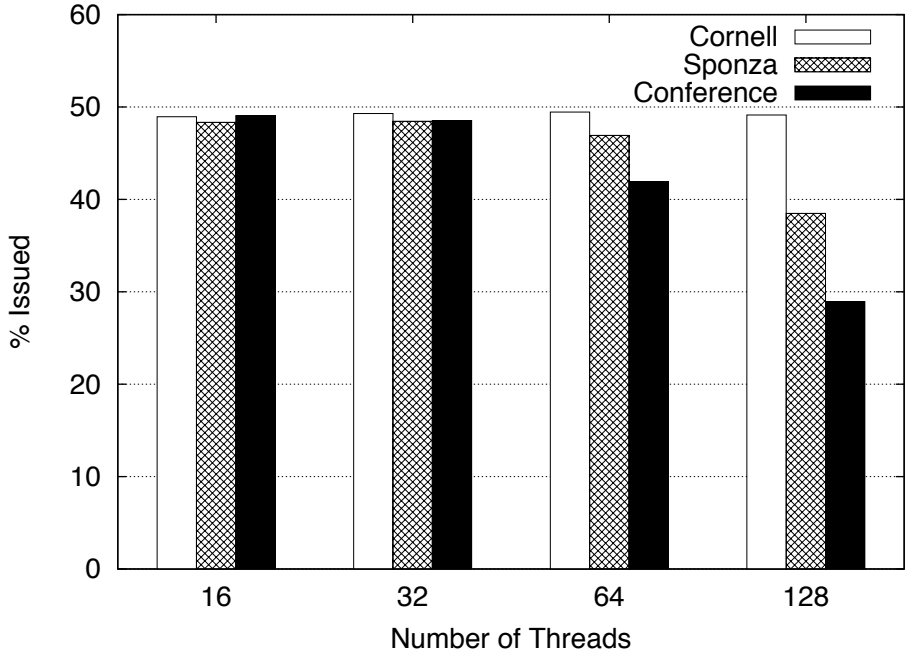


Figure 5.2: Thread Performance (% Issued)

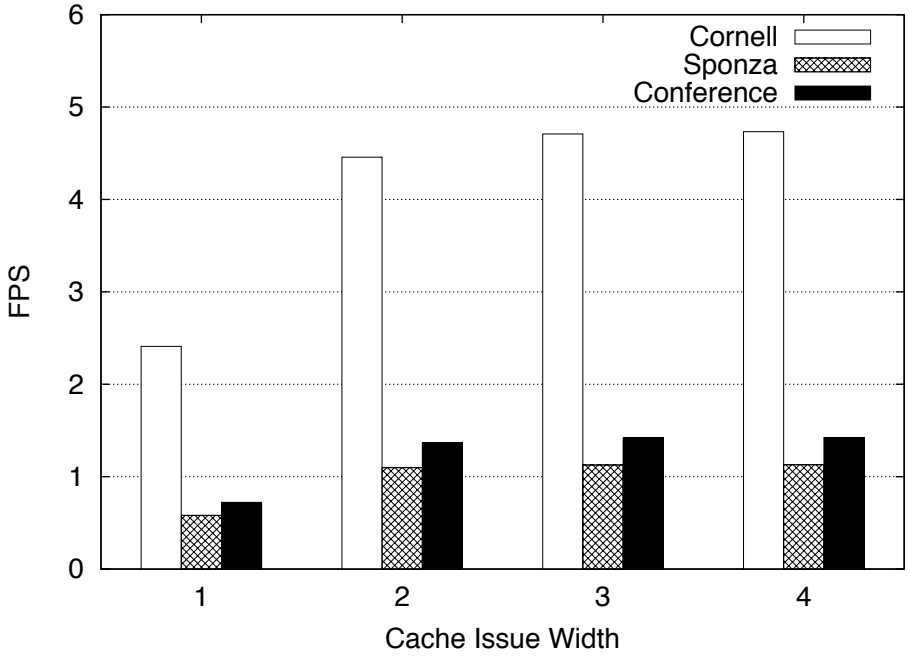


Figure 5.3: Single TM Performance as Cache Issue Width is Varied

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. The simulations in this section show memory bandwidths between L1 cache and the register file that range from 10-100 GB/s depending on the size of the scene. The L2-L1 bandwidth ranges from 4-50 GB/s, and DRAM-L2 from 250Mb/s to 6GB/s for reads. These clearly cover a broad range depending on the size and complexity of the scene. The instruction caches are modeled as 8 kbyte direct 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. Sections 5.2 and 5.3 do not make this assumption, but because of the small code size the L1 I-cache has little impact on processing times.

Comparing against the Saarland RPU [40, 78], TRaX has higher frame rates in the same technology, while also providing enhanced flexibility by allowing all parts of the ray tracing algorithm to be programmable instead of just the shading computations. In contrast, the RPU had fixed function acceleration structure traversal restricted to only using KD trees. The programmability in TRaX allows the 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 and demonstrates the limitations of an architecture not specifically designed for ray tracing. In particular, TRaX 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 5.6 shows these comparisons.

5.1.6 Secondary Ray Performance

We call the initial rays that are cast from the eye-point into the scene to determine visibility "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

Table 5.6: 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 1024×768 Images. Cell Numbers are for 1024×1024 Images. The Cell is Best Compared Using the MRPS Metric Which Factors Out Image Size.

	TRaX		IBM Cell[29]		RPU[78]	
	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

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 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 [79, 56, 80]. 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 4.6.3). 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, a well-studied packet based ray/path tracer [80]. 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 became 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 bounding volume hierarchy (BVH) using the DynBVH algorithm [24]. 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 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 became 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 is 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 degrees, 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 thereby 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 degrees 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 above, 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 TM that was simulated. The images themselves can be seen in Figure 5.1 with data about the images shown in Table 5.1.

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 5.7. 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.

Table 5.7: 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 TM with 32 Thread Processors Running at a Simulated 500 MHz. Manta Numbers are Measured Running on a Single TM of an Intel Core2 Duo at 2.0GHz. Speed Results are Normalized to Path Tracing with a 10 Degree 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 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. So, 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 2.0GHz. The TRaX numbers are from our cycle-accurate simulator assuming a 500MHz speed and using just a single TM with 32 thread processors. We expect these numbers to scale very well as we tile multiple TMs on a single die. As mentioned in Section 5.1, chips with between 22 to 78 TMs per die would not be unreasonable.

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 5.7. 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 [47], which indicates that smaller TMs (TMs with fewer thread processors) may be interesting for path tracing.

5.2 Overall Chip Design

Based on the results of the TM exploration in Section 5.1, this section analyzes more of the chip-wide issues associated with the interaction between the groups of TMs. Instead of exploring the full range of threads from 1-128, this section narrows the search significantly to only those numbers of threads that were found to be most compelling (16-64). As a result, we simulate a number of full chip configurations based on this narrowed set TM configurations.

5.2.1 Architectural Exploration Procedure

In this section, we analyze TRaX architectural options using four standard ray tracing benchmark scenes, shown in Figure 5.4, that provide a representative range of performance characteristics, and were also reported in [2]. This design space

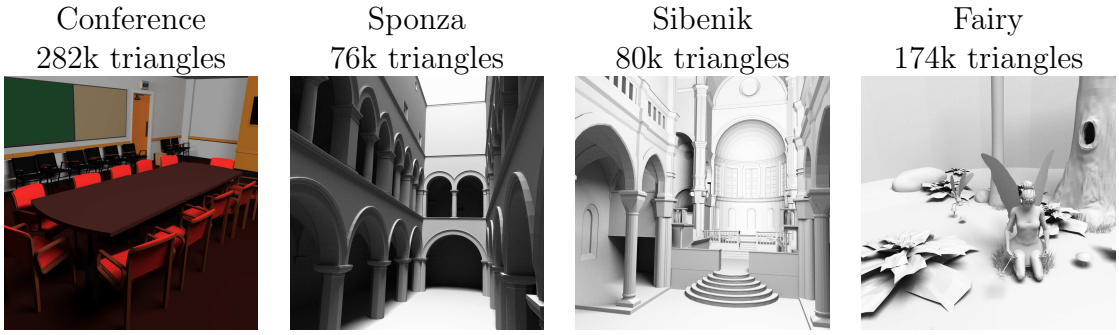


Figure 5.4: Test Scenes Used to Evaluate Performance. (a) Conference (b) Sponza Atrium (c) Sibenik Cathedral (d) Fairy Forest

exploration is based on 128x128 resolution images with one primary ray and one shadow ray per pixel. This choice reduces simulation complexity to permit analysis of an increased number of architectural options. The low resolution will have the effect of reducing primary ray coherence, but with the beneficial side-effect of steering our exploration towards a configuration that is tailored to efficiently handle incoherent rays. However, the final results of this section are based on the same images, the same image sizes, the same mixture of rays, and the same shading computations as reported for the SIMD GPU [2]. Our overall figure of merit is performance per area, reported as MRPS/mm^2, and is compared with other designs for which area is either known or estimable.

As part of this second exploration, we examine an unrealistic, exhaustively-provisioned TRaX multiprocessor as a starting point. This serves as an upper bound on raw performance, but requires an unreasonable amount of chip area. We then explore various multibanked Dcaches and sharing Icaches using Cacti v6.5 to provide area and speed estimates for the various configurations [81]. Next, we consider sharing large functional units which are not heavily used in order to reduce area with a minimal performance impact. Finally, we explore a chip-wide configuration that uses shared L2 caches for a number of TMs.

The ray tracer application can be run as a simple ray tracer with ambient occlusion, or as a path tracer which enables more detailed global illumination effects using Monte-Carlo sampled Lambertian shading [6]; this generates many more secondary rays. Our ray tracer supports fully programmable shading and texturing and uses a

bounding volume hierarchy acceleration structure. In this section, we use the same shading techniques as in [2], which does not include image-based texturing. As in Section 5.1, this ray tracer application is compiled to use a large number of registers and has no thread-local stack space.

5.2.2 Thread Multiprocessor (TM) Design

The primary comparison for this section is against the NVIDIA GTX285 [2] of the GT200 architecture family. The GT200 architecture operates on 32-thread SIMD "warps," allowing a reasonable comparison between the GPU SIMD performance and our TRaX MIMD behavior. The "SIMD efficiency" metric is defined in [2] to be the percentage of SIMD threads that perform computations. Note that some of these threads perform speculative branch decisions which may perform useless work, but this work is always counted as efficient. In our architecture the equivalent metric is thread issue rate. This is the average number of independent threads that can issue an instruction on each cycle. These instructions always perform useful work. The goal is to have thread issue rates as high or higher than the SIMD efficiency reported on highly optimized SIMD code. This implies an equal or greater level of parallelism, but with more flexibility and due to our unique architecture, less area.

We start with 32 threads in a TM based on the TM exploration in Section 5.1, but switch to 1 GHz due to the high performance target. Each thread processor has 128 registers, issues in order, and employs no branch prediction, while keeping the call stack in the register file. To discover the maximum possible performance achievable, each initial thread contains all of the resources that it can possibly consume. In this configuration, the data caches are overly large (enough capacity to entirely fit the dataset for two of our test scenes, and unrealistically large for the others), with one bank per thread. There is one functional unit (FU) of each type available for every thread. Our ray tracing code footprint is relatively small, which is typical for most advanced interactive ray tracers (ignoring custom artistic material shaders) [5, 6] and is similar in size to the ray tracer evaluated in [2]. Hence the Icache configurations are relatively small and therefore fast enough to service two requests per cycle at 1 GHz according to Cacti v6.5 [81], so 16 instruction caches are sufficient to service

the 32 threads. This configuration provides an unrealistic best-case issue rate for a 32-thread TM.

Table 5.8 shows the area of each functional component in a 65nm process, and the total area for a 32-thread TM, sharing the multibanked Dcache and the 16 single-banked Icaches. Note that the total area is for the baseline over-provisioned 1 GHz 32-thread TM configuration where each thread has a copy of every functional unit. Memory area estimates are from Cacti v6.5[1]. Memory latency is also based on Cacti v6.5: 1 cycle to L1, 3 cycles to L2, and 300 cycles to main memory. FU area estimates are based on synthesized versions of the circuits using Synopsys DesignWare/Design Compiler and a commercial 65nm CMOS cell library. These functional unit area estimates are conservative as a custom-designed functional unit would certainly have smaller area. All cells are optimized by Design Compiler to run at 1 GHz and multicycle cells are fully pipelined. The average thread issue rate is 89%, meaning that an average of 28.5 threads are able to issue on every cycle out of the 32 threads available. The raw performance of this configuration is very good, but the area is huge. The next step is to reduce thread resources to save area without

[1]Note that Cacti v6.5 has been specifically enhanced to provide more accurate size estimates than previous versions, for relatively small caches of the type we are proposing.

Table 5.8: Functional Unit Areas and Performance

Unit	Area (mm^2)	Cycles	Total Area (mm^2)
4MB Dcache (32 banks)		1	33.5
4KB Icaches	0.07	1	1.12
128x32 RF	0.019	1	0.61
FP InvSqrt	0.11	16	3.61
Int Multiply	0.012	1	0.37
FP Multiply	0.01	2	0.33
FP Add/Sub	0.003	2	0.11
Int Add/Sub	0.00066	1	0.021
FP Min/Max	0.00072	1	0.023
Total			39.69
Avg thread issue	MRPS/thread	MRPS/mm^2	
89%	5.6	0.14	

sacrificing performance. With reduced area the MRPS/mm^2 increases and provides an opportunity to tile more TMs on a chip.

5.2.3 Exploring Constrained Resource Configurations

We now consider constraining caches and functional units to evaluate the design points with respect to MRPS/mm^2. Cache configurations are considered before shared functional units, and then revisited for the final multi-TM chip configuration. All performance numbers in our design space exploration are averages from the four scenes in Figure 5.4.

Our baseline architecture shares one or more instruction caches among multiple threads. Each of these Icaches is divided into one or more banks, and each bank has a read port shared between the threads. Our ˜1000-instruction ray tracer program fits entirely into 4KB instruction caches and provides a 100% hit-rate while being double pumped at 1 GHz. This is virtually the same size as the ray tracer evaluated in [2].

Our data cache model provides write-around functionality to avoid dirtying the cache with data that will never be read. The only writes the ray tracer issues are to the write-only frame buffer; this is typical behavior of common ray tracers. Our compiler stores all temporary data in registers, and does not use a call stack. Stack traversal is handled with a special set of registers designated for stack nodes. Because of the lack of writes to the cache, we achieve relatively high hit-rates even with small caches, as seen in Figure 5.5. The data cache is also banked similarly to the instruction cache. Data cache lines are 8 4-byte words wide.

We explore L1 Dcache capacities from 2KB to 64KB and banks ranging from 1 to 32, both in power of 2 steps. Similarly, numbers and banks of Icaches range from 1 to 16. First the interaction between instruction and data caches needs to be considered. Instruction starvation will limit instruction issue and reduce data cache pressure. Conversely, perfect instruction caches will maximize data cache pressure and require larger capacity and increased banking. Neither end-point will be optimal in terms of MRPS/mm^2. This interdependence forces us to explore the entire space of data and instruction cache configurations together.

Other resources, such as the FUs, will also have an influence on cache performance, but the exponential size of the entire design space is intractable. Since we have yet

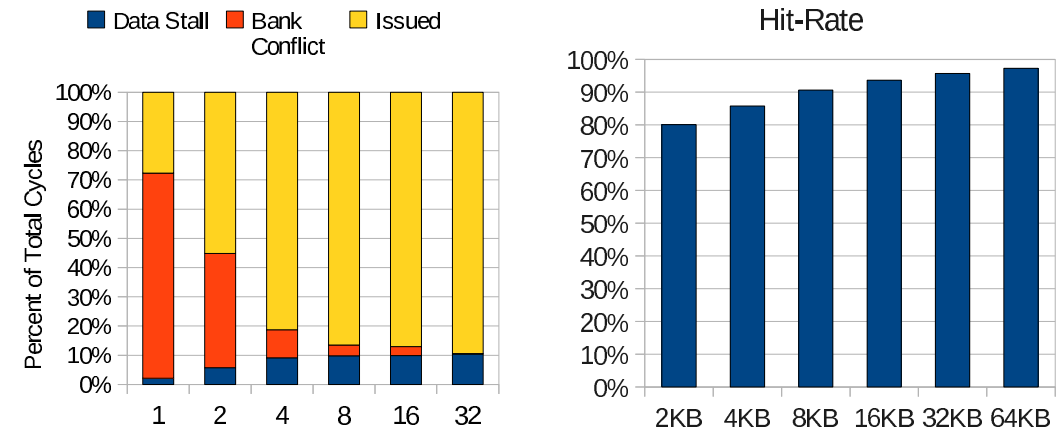


Figure 5.5: L1 Data Cache Performance for a Single TM with Over-Provisioned Functional Units and Instruction Cache. (a) Issue Rate for Varying Banks in a 2KB Data Cache. (b) Dcache Hit%, 8-banks and Varying Capacities.

to discover an accurate pruning model, we have chosen to evaluate certain resource types in order. It is possible that this approach misses the optimal configuration, but our results indicate that our solution is adequate. After finding a "best" TM configuration, we revisit Dcaches and their behavior when connected to a chip-wide L2 Dcache shared among multiple TMs. For single-TM simulations we pick a reasonable L2 cache size of 256KB. Since only one TM is accessing the L2, this results in unrealistically high L2 hit-rates, and diminishes the effect that the L1 hit-rate has on performance. We rectify this inaccuracy in Section 5.2.3, but for now this simplified processor, with caches designed to be as small as possible without having a severe impact on performance, provides a baseline for examining other resources, such as the functional units.

The next step is to consider sharing lightly used and area-expensive FUs for multiple threads in a TM. The goal is area reduction without a commensurate decrease in performance. Table 5.8 shows area estimates for each of our functional units. The integer multiply, floating-point (FP) multiply, FP add/subtract, and FP inverse-square-root units dominate the others in terms of area, thus sharing these units will have the greatest effect on reducing total TM area. In order to maintain a reasonably sized exploration space, these are the only units considered as candidates for sharing.

The other units are too small to have a significant effect on the performance per area metric.

We ran many thousands of simulations and varied the number of integer multiply, FP multiply, FP add/subtract and FP inverse-square-root units from 1 to 32 in powers of 2 steps. Given N shared functional units, each unit is only connected to $32/N$ threads in order to avoid complicated connection logic and area that would arise from full connectivity. Scheduling conflicts to shared resources are resolved in a round-robin fashion. Figure 5.6 shows that the number of FUs can be reduced without drastically lowering the issue rate, and Table 5.9 shows the top four configurations that were found in this phase of the design exploration. All of the top configurations use the cache setup found in Section 5.2.3: two instruction caches, each with 16 banks, and a 4KB L1 data cache with 8 banks and approximately 8% of cycles as data stalls for both our TM-wide and chip-wide simulations.

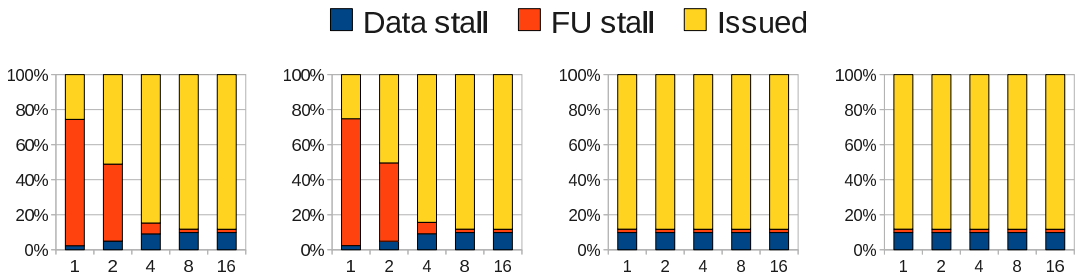


Figure 5.6: Effect of Shared Functional Units on Issue Rate Shown as a Percentage of Total Cycles. (a) FP Add/Sub (13% of Issued Insts). (b) FP Multiply (13% of Issued Insts). (c) FP Inverse Square Root (0.4% of Issued Insts). (d) Int Multiply (0.3% of Issued Insts)

Table 5.9: Optimal TM Configurations in Terms of MRPS/mm^2.

INT MUL	FP MUL	FP ADD	FP INV	MRPS/ thread	Area (mm^2)	MRPS/ mm^2
2	8	8	1	4.2	1.62	2.6
2	4	8	1	4.1	1.58	2.6
2	4	4	1	4.0	1.57	2.6
4	8	8	1	4.2	1.65	2.6

Area is drastically reduced from the original over-provisioned baseline, but performance remains relatively unchanged. Note that the per-TM area is quite a bit smaller than the area we estimate for a GTX285 SM (streaming multiprocessor). Table 5.10 compares raw compute and register resources for our TM compared to a GTX285 SM. This is primarily due to the aggressive resource sharing in TRaX, and the smaller register file since TRaX does not need to support multithreading in the same way as the GT200. While many threads on the GT200 are context switched out of activity and do not attempt to issue, every single thread in the 32 thread TRaX TM attempts to issue on each cycle, thereby remaining active. Our design space included experiments where additional thread contexts were added to the TMs, allowing context switching from a stalled thread. These experiments resulted in 3-4% higher issue rate, but required much greater register area for the additional thread contexts.

Given the TM configurations found in Section 5.2.3 that have the minimal set of resources required to maintain high performance, we now explore the impact of tiling many of these TMs on a chip. Our chip-wide design connects one or more TMs to an L2 Dcache, with one or more L2 caches on the chip. Up to this point, all of our simulations have been single-TM simulations which do not realistically model L1 to L2 memory traffic. With many TMs, each with an individual L1 cache and a shared L2 cache, bank conflicts will increase and the hit-rate will decrease. This will require

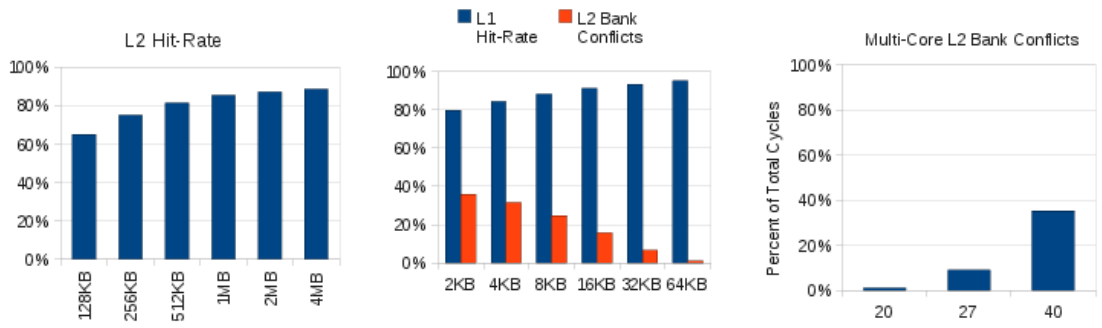
Table 5.10: GTX285 SM vs. MIMD TM Resource Comparison. Area Estimates Are Normalized to Our Estimated FU Sizes from Table 5.8, and Not From Actual GTX285 Measurements.

	GTX285 SM (8 cores)	MIMD TM (32 threads)
Registers	16384	4096
FPAdds	8	8
FPMuls	8	8
INTAdds	8	32
INTMuls	8	2
Spec op	2	1
Register Area (mm^2)	2.43	0.61
Compute Area (mm^2)	0.43	0.26

a bigger, more highly banked L2 cache. Hit-rate in the L1 will also affect the level of traffic between the two levels of caches so we must explore a new set of L1 and L2 cache configurations with a varying number of TMs connected to the L2.

Once many TMs are connected to a single L2, relatively low L1 hit-rates of 80-86% reported in some of the candidate configurations for a TM will likely put too much pressure on the L2. Figure 5.7(b) shows the total percentage of cycles stalled due to L2 bank conflicts for a range of L1 hit-rates. The 80-86% hit-rate, reported for some initial TM configurations, results in roughly one-third of cycles stalling due to L2 bank conflicts. Even small changes in L1 hit-rate from 85% to 90% will have an effect on reducing L1 to L2 bandwidth due to the high number of threads sharing an L2. We therefore explore a new set of data caches that result in a higher L1 hit-rate.

We assume up to four L2 caches can fit on a chip with a reasonable interface to main memory, allowing a 64 bit memory bus per L2, resulting in a total of 256 bits total, less than many commercial GPUs. Our target area is under 200mm^2, so 80 TMs (2560 threads) will fit even at 2.5mm^2 each. Section 5.2.3 shows a TM area of 1.6mm^2 is possible, and the difference provides room for additional exploration. The 80 TMs are evenly spread over the multiple L2 caches. With up to four L2 caches per chip, this results in 80, 40, 27, or 20 TMs per L2. Figure 5.7(c) shows the percentage



(a) (b) (c)

Figure 5.7: L2 Performance for 16 Banks and TMs with the Top Configuration Reported in Table 5.9. (a) Hit-rate for Varying L2 Capacities with 20 TMs Connected to Each L2. (b) Percentage of Cycles not Issued Due to L2 Bank Conflicts for Varying L1 Capacities (and Thus Hitrates) for 20 TMs. (c) L2 Bank Conflicts for a Varying Number of TMs Connected to Each L2. Each TM Has a 64KB L1 Cache with 95% Hitrate.

of cycles stalled due to L2 bank conflicts for a varying number of TMs connected to each L2. Even with a 64KB L1 cache with 95% hit-rate, any more than 20 TMs per L2 results in >10% L2 bank conflict stalls. We therefore choose to arrange the proposed chip with four L2 caches serving 20 TMs each.

Figure 5.8 shows how individual TMs of 32 threads might be tiled in conjunction with their L2 caches. The result of the design space exploration is a set of architectural configurations that all fit in under 200mm^2 and maintain high performance. A selection of these are shown in Table 5.11 and are what we use to compare to the best known GPU ray tracer for the GTX285 in Section 5.2.4. Note that the GTX285 has close to half the die area devoted to texturing hardware, and none of the benchmarks

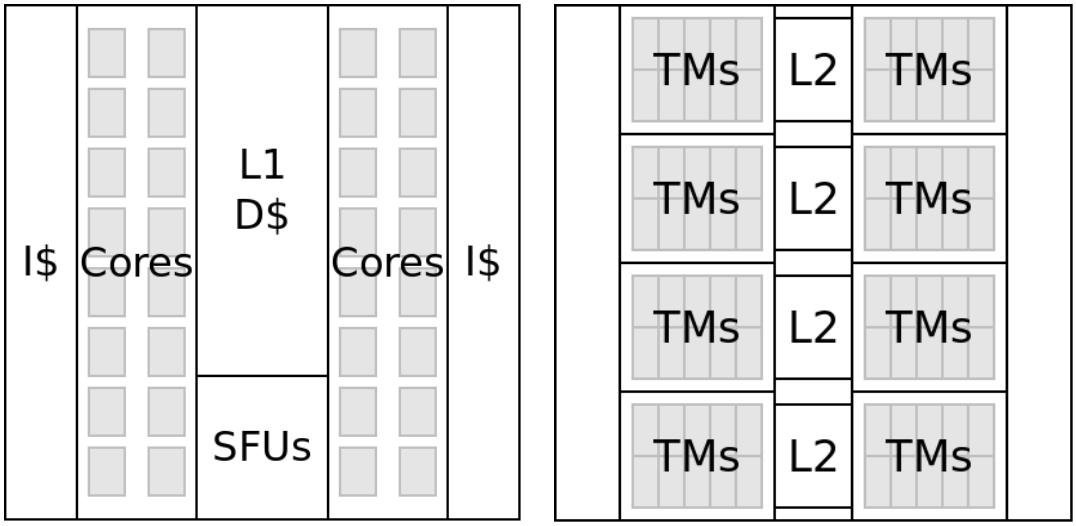


Figure 5.8: Potential TM and Multi-TM Chip Floor Plans. (a) TM Layout of 32 Threads and Shared Resources. (b) Chip with Multiple TMs Sharing L2 Caches.

Table 5.11: A Selection of Our Top Chip Configurations and Performance Compared to an NVIDIA GTX285 and Copernicus.

L1 Size	L1 Banks	L2 Size	L2 Banks	L1 Hitrate	L2 Hitrate	Per Cache Bandwidth (GB/s) L1→reg	L2→L1	main→L2	Thread Issue	Area (mm^2)	MRPS	MRPS/ mm^2
32KB	4	256KB	16	93%	75%	42	56	13	70%	147	322	2.2
32KB	4	512KB	16	93%	81%	43	57	10	71%	156	325	2.1
32KB	8	256KB	16	93%	75%	43	57	14	72%	159	330	2.1
32KB	8	512KB	16	93%	81%	43	57	10	72%	168	335	2.0
64KB	4	512KB	16	95%	79%	45	43	10	76%	175	341	1.9
GTX285 (area is from 65nm GTX280 version for better comparison)									75%	576	111	0.2
GTX285 SIMD core area only — no texture unit (area is estimated from die photo)									75%	~300	111	0.37
Copernicus at 22nm, 4GHz, 115 Core 2-style cores in 16 tiles									98%	240	43	0.18
Copernicus at 22nm, 4GHz, with their envisioned 10x SW improvement									98%	240	430	1.8
Copernicus with 10x SW improvement, scaled to 65nm, 2.33GHz									98%	961	250	0.26

reported in [2] or in our own studies use image-based texturing. Thus it may not be fair to include texture hardware area in the MRPS/mm^2 metric. On the other hand, the results reported for the GTX285 do use the texture memory to hold scene data for the ray tracer, so although it is not used for texturing, that memory (which is a large portion of the hardware) is participating in the benchmarks.

Optimizing power is not a primary goal of this exploration, and because we endeavor to keep as many units busy as possible we expect power to be relatively high. Using energy and power estimates from Cacti v6.5 and Synopsys DesignWare, we calculated a rough estimate of our chip's total power consumption on the benchmark scenes. Given the top chip configuration reported in Table 5.11, and activity factors reported by our simulator, we roughly estimate a chip power consumption of 83 watts which we believe is in the range of power densities for commercial GPUs. Copernicus [44] is an architecture design somewhat similar to TRaX that consists of many tiles of basic processing elements. However, Copernicus uses much more traditional CPU cores without the resource sharing in TRaX. In the table, Copernicus area and performance are scaled to 65nm and 2.33 GHz to match the Xeon E5345, which was their starting point. Each TRaX MIMD thread multiprocessors (TM) has 2 integer multiply, 8 FP multiply, 8 FP add, 1 FP invsqrt unit, and 2 16-banked Icaches.

5.2.4 Results

To evaluate the results of our design space exploration we chose two candidate architectures from the top performers: one with small area (147mm^2) and the other with larger area (175mm^2) but higher raw performance (as seen in Table 5.11). We ran detailed simulations of these configurations using the same three scenes as in [2] and using the same mix of primary and secondary rays. Due to the widely differing scenes and shading computations used in [2] and [44], a direct comparison between both architectures is not feasible. We chose to compare against [2] because it represents the best reported performance to date for a ray tracer running on a GPU, and their ray tracing application is more similar to ours. We do, however, give a high level indication of the range of performance for our MIMD architecture, GTX285,

and Copernicus in Table 5.11. In order to show a meaningful area comparison, we used the area of a GTX280, which uses a 65nm process, and other than clock frequency, is equivalent to the GTX285. Copernicus area is scaled up from 22nm to 65nm. Assuming that their envisioned 240mm^2 chip is 15.5mm on each side, a straightforward scaling from 22nm to 65nm would be a factor of three increase on each side, but due to certain process features not scaling linearly, we use a more realistic factor of two per side, giving a total equivalent area of 961mm^2 at 65nm. We then scaled clock frequency from their assumed 4GHz down to the actual 2.33GHz of the 65nm Clovertown core on which their original scaling was based. The 10x scaling due to algorithmic improvements in the Razor software used in the Copernicus system is theoretically envisioned in their paper [44].

The final results and comparisons to GTX285 are shown in Table 5.12. It is interesting to note that although GTX285 and Copernicus take vastly different approaches to accelerating ray tracing, when scaled for performance/area they are quite similar. It is also interesting to note that although our two candidate configurations perform differently in terms of raw performance, when scaled for MRPS/mm^2 they offer similar performance, especially for secondary rays.

When our raw speed is compared to the GTX285 our configurations are between 2.3x and 5.6x faster for primary rays (average of 3.5x for the three scenes and two MIMD configurations) and 2.3x to 9.8x faster for secondary rays (5.6x average). This supports our view that a MIMD approach with appropriate caching scales better

Table 5.12: Comparing Our Performance on Two Different Configurations to the GTX285 for Three Benchmark Scenes [2]. Primary Ray Tests Consisted of 1 Primary and 1 Shadow Ray Per Pixel. Diffuse Ray Tests Consisted of 1 Primary and 32 Secondary Global Illumination Rays Per Pixel.

		Conference (282k triangles)		Fairy (174k triangles)		Sibenik (80k triangles)	
MIMD	Ray Type	MIMD Issue Rate	MIMD MRPS	MIMD Issue Rate	MIMD MRPS	MIMD Issue Rate	MIMD MRPS
147mm^2	Primary	74%	376	70%	369	76%	274
	Diffuse	53%	286	57%	330	37%	107
175mm^2	Primary	77%	387	73%	421	79%	285
	Diffuse	67%	355	70%	402	46%	131
SIMD	Ray Type	GTX SIMD eff.	GTX MRPS	GTX SIMD eff.	GTX MRPS	GTX SIMD eff.	GTX MRPS
GTX285	Primary	74%	142	76%	75	77%	117
	Diffuse	46%	61	46%	41	49%	47

MIMD MRPS/mm^2 ranges from 2.56 (Conference, primary rays) to 0.73 (Sibenik, diffuse rays) for both configs
SIMD MRPS/mm^2 ranges from 0.25 (Conference, primary rays) to 0.07 (Fairy, diffuse rays)
SIMD (no texture area) MRPS/mm^2 ranges from 0.47 (Conference, primary) to 0.14 (Fairy, diffuse)

for secondary rays than SIMD. We can also see that our thread issue rates do not change dramatically for primary vs. secondary rays, especially for the larger of the two configurations. When scaled for MRPS/mm^2 our configurations are between 8.0x and 19.3x faster for primary rays (12.4x average), and 8.9x to 32.3x faster for secondary rays (20x average). Even if we assume that the GTX285 texturing unit is not participating in the ray tracing, and thus using a 2x smaller area estimate for that processor, these speed-ups are still approximately 6x-10x on average. The fact that our MIMD approach is better in terms of performance per area than the SIMD approach is nonintuitive at first glance. This is mostly because we keep our cores very small due to aggressive resource sharing and by not including extra register resources for multithreading (see Table 5.10).

We believe that MRPS and MRPS/mm^2 are fair units of measurement for ray tracing hardware because they are relatively independent of the resolutions at which the scenes are rendered. To put these MRPS numbers into perspective, if an interesting image is assumed to take between 4-10m rays to render (see Section 5.2.1), then our MIMD approach would render between 13 (10M rays / 131 MRPS) and 100 (4M rays / 402 MRPS) frames per second (fps) depending on the mix and complexity of the rays. A scene requiring 8M rays (which is a relatively complex scene) at 300 MRPS would achieve 37.5fps.

5.3 Mobile Ray Tracing

In order to target a modern mobile computing environment for ray tracing, a number of changes need to be made to the TRaX configurations discussed previously. First, a smaller number of threads should be included to keep the power and area consumption down. Additionally, we change the basic thread processor model to use a smaller, 32 registers in the register file while adding on a small local store SRAM to use as the stack. This causes a slight instruction bloat as values need to be moved to and from the local store memory, however, it has the benefit of less expensive register operations.

As with the other TRaX designs, mobile TRaX has Clusters of TPs that share multiply and add floating point units (FPUs). The size of the cluster is varied to provide high utilization of the FPUs for the mobile context. TPs in a TM still share

a floating point divide and inverse square root unit, which is even more area and energy expensive, yet rarely utilized. Each TM shares a banked instruction cache to allow TPs that do not conflict for a particular bank to proceed in parallel. Multiple TM tiles share a banked data cache which contains the global, shared scene data and frame buffer. An example TM with 32 TPs can be seen in Figure 5.9. Experimental group and cluster size details can be found in Section 5.3.1.

5.3.1 Architecture and Methodology

The overall architecture follows the same ideas as Sections 5.1 and 5.2 based on a simple, in-order integer thread execution model for TPs grouped into TMs that share resources. That TM tile can be replicated to increase the total compute power. Since the floating point units are shared within the architecture, we strive to find a design point that is capable of achieving high utilization of these shared units. To a great extent the floating point utilization depends on the particular application executing on the system. In this section, we consider a ray tracer that traces primary visibility as well as shadow rays.

For the mobile version, we use a customized LLVM back end to emit code compatible with the TRaX ISA. The mobile compiler targets an architecture with 32 registers only, and therefore a small local store memory is included to hold all thread-local stack values. In order to execute architecture specific instructions, we expose a few simple compiler intrinsics to the programmer. The single executable is then run on each thread independently. The primary form of communication among threads is a simple atomic increment instruction that each thread uses to find a unique assignment. Global memory operations are managed by the programmer and the acceleration structure is built by the host CPU and made available in the accelerator's memory space.

The simulation environment and architecture is mostly the same as in the other explorations. Each TP consists of a simple, in-order, single-issue integer processor as before, with the register file changed to 32 general purpose registers and a small 512-byte local memory added. The local memory acts as an extended register file for local stack operations. We do not employ branch prediction and rely instead on thread parallelism to achieve higher performance and to keep the shared floating point units

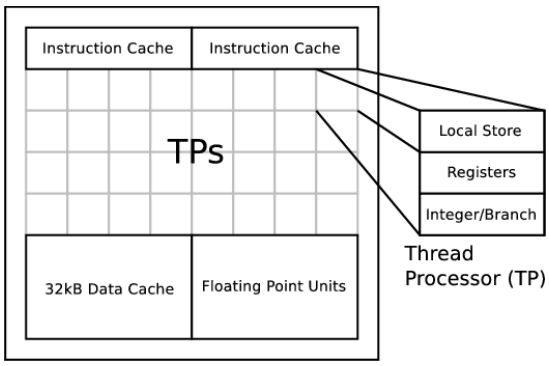


Figure 5.9: A 32-thread TM with Shared Caches and FPUs

busy. In every mobile configuration the FPU is shared by 8 TPs. We find empirically that this is sufficient since each TP spends execution on pointer chasing and waiting for memory requests to return, which keeps it from issuing FPU instructions on every cycle.

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

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

For each TM, we use a 16kB banked data cache that caches data from the global shared memory. We find that one bank per 8 TP cluster is the appropriate choice. The global memory segment includes all of the scene data, acceleration structure, and frame buffer. Because the thread assignment gives one pixel at a time to each thread,

we force all frame buffer writes to go around the data cache, thereby preventing pollution of the cache by lines that are write only.

We limit the off-chip bandwidth to 8 GB/s based on the fact that upcoming mobile SoCs, such as the Samsung Exynos 5250 [82], achieve up to 12.8 GB/s of memory bandwidth with a 64 bit memory bus. We believe 8 GB/s is a reasonable assumption for a compute-bound GPU in the near future because SoCs also share that memory bandwidth with other IP blocks. We note that if the GPU and host CPU are both in memory bound computational segments, the shared bandwidth will impose performance restrictions. Section 5.3.3 considers a future SoC with more available memory bandwidth. For area and performance estimates, we use Synopsys DesignWare/Design Compiler [53] and a commercial 65nm CMOS cell library to synthesize functional units, and Cacti 6.5 [81] for our cache and memory analysis.

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

5.3.2 Results

We simulated the execution times for a number of configurations of the proposed architecture on a simple ray tracer application to gather performance and utilization data. We consider TM configurations with 32, 48 and 64 TPs per TM and for 1, 2, 4, 6 or 8 TM tiles. Results in all tables are ordered by the number of total threads across all TMs and are annotated by the number of TMs and the number of TPs per TM in parentheses. The test scenes in Figure 5.4 were run on each configuration and the results presented are an average across the benchmark scenes unless indicated otherwise. Note that while some of the images shown have textures, the ray tracer used to report results does not perform texturing. Each scene was rendered at a resolution of 1280x720 with primary rays and shadow rays for a single light source. For every eight threads in a TM we provide one floating point multiplier and one floating point adder while the entire TM shares one special functional unit regardless

of the number of threads. Thus a 32-thread TM has a maximum 9 FLOP per cycle capability while the 48 and 64 TP TMs have 13 and 17 FLOPs, respectively.

A comparison of floating point capabilities of our architecture and commercial rasterization architectures can be found in Table 5.13. The "RT GFLOPS" column is the simulated floating point performance when running our ray tracer and is not reported for the commercial architectures because ray tracers are not readily available for comparison on those architectures. The "RT GFLOPS" entry for MRTP [83, 84] is approximated based on the thread issue data provided in their papers. Only multiplies and adds are considered in the floating point compute capabilities of the various architectures, and do not include the rarely used FPDIV/iSQRT special function unit. The commercial architectures included in this table include PowerVR SGX543 by Imagination Technologies and Tegra 2 from NVIDIA. These commercial chips are

Table 5.13: Comparison of Mobile Graphics Accelerator Architectures. All Accelerators Are Scaled to 65nm and 500 MHz Naively for Better Comparison with Our Configurations. *Tegra 2 Die Size is Estimated from a Die Photo.

Architecture	Size(mm^2)	GFLOPS	RT GFLOPS
PowerVR SGX543MP1	8.0	18.0	
PowerVR SGX543MP2	16.0	36.0	
NVIDIA Tegra 2	6*	8.0	
MRTP [83] (130nm)	16.0	4.3	≈1.2
MRTP (naively scaled)	4.0	21.5	≈6.0
32 (1x32)	1.9	4.0	2.5
48 (1x48)	2.5	6.0	3.7
64 (2x32)	3.8	8.0	4.9
64 (1x64)	3.2	8.0	4.9
96 (2x48)	5.1	12.0	7.2
128 (4x32)	7.6	16.0	9.3
128 (2x64)	6.3	16.0	9.2
192 (6x32)	11.4	24.0	12.6
192 (4x48)	10.1	24.0	12.7
256 (8x32)	15.2	32.0	15.5
256 (4x64)	12.6	32.0	15.7
288 (6x48)	15.2	36.0	17.1
384 (8x48)	20.2	48.0	20.3
384 (6x64)	18.9	48.0	20.3
512 (8x64)	25.3	64.0	23.1

not capable of efficient ray tracing, but are used in mobile phones and tablet designs, so their floating point performance is included as the only point of reference.

Figure 5.10 shows the scenes used to study the performance of the mobile version of TRaX. Table 5.14 gives a comparison of the ray processing capabilities of the various configurations that were simulated for those scenes. As the number of threads increases, so does the raw performance of the configuration. In the case of the dragon scene, the memory access pattern is such that even with only 128 threads, the computation is memory bandwidth limited, preventing further increases in ray tracing performance. Section 5.3.3 goes into more depth on the bandwidth concern. In order to provide a reasonable comparison to the MRTP, we consider the only scene we share in common with them, viz. the dragon. We choose a 128-thread configuration because the area is similar to what the MRTP would use when scaled to a 65nm process. We also scale their performance up to 500 MHz assuming the change to the 65nm process would allow for a faster clock rate, although a 5x increase is likely optimistic. Our 128 thread configuration is able to perform 6.18 million rays per second while the MRTP achieves only 0.515 million rays per second, giving our architecture a 13x speedup for the same circuit area.

For an HD resolution of 1280x720 pixels, mobile TRaX can ray trace images with full shadows at 3.4 frames per second. While this is not a real-time frame rate, it is still interactive enough for most medical imaging and visualization applications. Furthermore, frameless techniques can be used for applications where some image quality degradation is preferable to losing interactivity and the quality provided at interactive rates is determined by the number of rays per second.

5.3.3 Memory Bandwidth Concerns

Our architecture performs ray tracing well and is capable of utilizing the available floating point units effectively until the memory bandwidth limit is reached. In particular, the performance of the dragon scene stops scaling because it reaches the bandwidth limit with only 128 total threads for any TM count. However, the bandwidth available to mobile SoCs is likely to grow in the future due to increasing memory clock rates as well as larger memory buses. Table 5.15 shows the increases in performance that can be achieved when bandwidth is raised to 16GB/s. The dragon

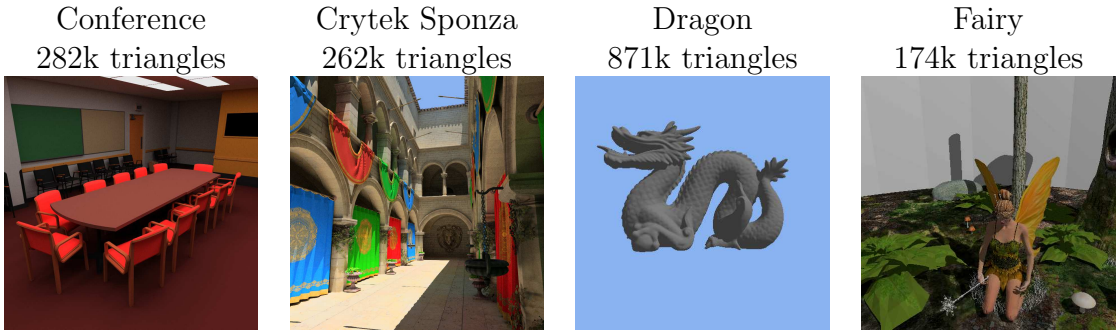


Figure 5.10: Test Scenes Used to Evaluate Mobile Performance. (a) Conference (b) Crytek Sponza (c) Dragon (d) Fairy Forest

Table 5.14: Ray Tracing Performance, Shown in Millions of Rays Per Second.

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

scene achieves almost double performance since it is primarily memory bandwidth constrained. It is likely that increasing the size of the cache would also decrease the pressure on the memory bus.

Pure SIMD is not the most efficient ray tracing architecture due to the divergent execution and memory patterns induced by traversing the acceleration structure and the intrinsic nature of secondary rays [2]. The competing MRTP architecture [83] addresses this limitation by allowing their architecture to dynamically reconfigure to accommodate smaller SIMT blocks. The MRTP relies on single-thread vector

Table 5.15: Performance in Millions of Rays Per Second with the Baseline and Increased Memory Bandwidth for the Dragon Scene as Well as an Average Across All Scenes Tested.

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

operations to maintain performance while avoiding the extra overhead of moving to a full MIMD architecture. Our alternative approach embraces this divergent behavior and allows threads to execute in MIMD fashion and recovers efficiency through resource sharing. Instead of giving each thread its own floating point multiplier and adder, we decouple those units, sharing them among a group of threads. This type of sharing is not possible in a typical SIMD architecture. Rarely will all threads need the same unit at the same time. Furthermore, we share banked instruction and data caches to enable parallel access when threads are not strictly synchronized. The normal MIMD overhead is greatly reduced, and we are able to find a 13x speedup over the reconfigurable SIMT architecture. Samsung has also recently shown some interest in mobile ray tracing, including showing a prototype running on an field programmable gate array (FPGA) that includes much more fixed-function logic than TRaX [85, 86, 87].

5.4 Conclusion

The TRaX architecture provides a framework for ray tracing computation that is more effective than current commercially available architectures and other research architectures. The effectiveness of the TRaX design primarily comes from embracing the natural program behavior found in ray tracers, rather than an attempt to warp the ray tracing algorithm to fit the hardware. While TRaX is tuned towards the functional requirements of our ray tracer, it still retains a large amount of programmability, allowing future algorithmic optimizations and improvements. A programmer can decide which advanced ray tracing features should be included, thereby increasing

the image quality at the expense of performance. Alternatively, image quality can be sacrificed in order to improve the performance of the system.

This chapter has shown that, by using the same basic architectural approach, ray tracing hardware can be designed to target both high-power, high-performance applications and low-power, low-performance applications. Importantly, TRaX allows traditional, single-threaded ray tracing applications to scale to many threads of execution trivially, and at run time. The limitation for the number of parallel threads that would allow for good scaling trivially for TRaX is on the order of one-tenth the number of primary samples in an image. That means that a 1080p HD image could scale relatively well up to around 200,000 threads, giving this approach many years of usefulness in the future.

CHAPTER 6

RELATED WORK

A number of architectural designs from both commercial and research environments can be viewed as related to TRaX in one way or another. In fact, nearly every architecture design in the last 30 years that allows for general purpose computation in any way has had some form of ray tracer implementation at some point. The architectural innovations that have led to increased ray tracing performance include any method for increasing the number of instructions that can be issued over time, increasing instruction *throughput*. A number of different techniques for improving throughput exist, including SIMD, multi-threading, very large instruction word, super scalar execution, out of order execution, and deep CPU pipelines. While general purpose CPU architectures have implemented nearly all of these features at some point, CPU innovations have primarily been concerned with increased performance of a single thread. In this chapter, I discuss a number of different approaches to increased execution performance, and reduced power consumption and how those approaches compare with the TRaX architecture.

6.1 High Performance GPU Architectures

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 [88, 89, 90, 91]. 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 [68, 69]. Current GPUs have up to hundreds and thousands of floating point units on a single GPU and aggregate memory bandwidth of nearly 300 Gbytes per second from their local memories. That impressive local memory

bandwidth is largely to support framebuffer access and 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.

Early Graphics Processing Units(GPU) were implemented as a fixed function pipeline that would take in objects according to a programming interface and provide the output that the programmer expected. Over time, more and more knobs were added to these pipelines that gave the programmer more control over how the objects would eventually be displayed on the screen. Eventually, a good amount of programmability was enabled, whereby programmers could implement their own pixel and vertex *shaders*, which are simple programs that would replace that stage of the standard graphics pipeline. Some efforts were made under this model to implement ray tracers with varying levels of success [92, 93], however, limitations in the shader programming model prevented high performance results. Eventually GPU architects decided (with encouragement from programmers wanting to add more programmable stages to the pipeline) that moving to a unified shader model, one where both pixel and vertex shaders would be executed on the same execution hardware, would be beneficial for load balancing, and for future workloads. The G80 was NVIDIA's first unified architecture [94], and it introduced the CUDA [95] programming methodology for general purpose computing. CUDA requires threads be grouped into "warps" of 32 threads that can be scheduled together to a compute block (AMD uses the term "wavefront" for the same concept). While these threads are coupled for scheduling, all unified architectures only execute 16 of the threads at a time on the SIMD execution units, often called SIMT in this context to emphasize the use of independent threads as the data elements. The result of this wide SIMD execution model is that branching can cause reduced SIMD performance as some threads must stall while waiting for other threads to complete their portion of the branch. The worst case scenario, which can occur often in ray tracing, would see a reduction in performance to $\frac{1}{16}$ of the maximum execution possible. Ray tracers have been analyzed on these unified architectures [41, 96], though limitations still exist as will be discussed later.

Even with the increase of general compute capabilities on GPUs, there still remain a number of special purpose functional units to support the traditional graphics pipeline. In particular, rasterization, raster output, and texture computations are carried out by special hardware designed for that purpose. The greatest comparison to the TRaX architecture comes from the most recent GPUs from NVIDIA and ATI(AMD), however, TRaX draws inspiration from all of the architectures discussed in this chapter. As the high performance GPUs are currently the highest performing throughput focused compute solutions, they are the most similar to the TRaX architecture.

6.1.1 NVIDIA Fermi

NVIDIA's previous top end architecture, named Fermi and sometimes called GF100 or GF110 (the name for the high end designs with subtle differences), is discussed in some detail in NVIDIA white papers [97, 27]. The main selling point of this architecture is the 512 CUDA cores, each of which can execute its own thread. In Fermi, these cores are gathered in streaming multiprocessors (SM) of 32 cores as 2 groups of 8-wide SIMD that executes twice per fetch as the cores are in a separate clock domain from the instruction fetch. This results in 64 independent fetches that occur on each cycle on a GF100. The scheduler allows a single SM to execute two independent warps simultaneously. A warp is a group of threads that is set up by the programmer. Due to the SIMD nature of the hardware, performance is optimal when all of the threads in a warp branch together. In the case where any threads diverge from the rest of the warp, those threads must execute serially and many of the cores can end up sitting idle.

Another distinction of the Fermi architecture is a much improved memory system over older GPUs. There is a 768 KB L2 cache along with L1 caches per SM that decrease the access latency for many memory operations that are common to ray tracing. The L1 cache and SM shared memory each have 16 KB of independent memory and an additional 32 KB that can be assigned to one or the other. There is also improved 64-bit arithmetic performance, but much of the computation done by ray tracing and rasterization does not require 64-bit precision. NVIDIA created

CUDA [95] as a method of general purpose programming for the G80 architecture and it has also been used on the GT200 (Tesla) architecture before Fermi [98, 99]. Scientific applications in CUDA are the primary use for 64-bit arithmetic operations on the GPU.

6.1.2 NVIDIA Kepler

NVIDIA's most recent architecture goes by the codename Kepler [100], and includes a number of specific chips such as the GK104 and GK107. Similar to other GPU architectures, it depends heavily on a SIMD execution model while providing a few improvements over previous NVIDIA GPUs. The biggest change is the removal of a separate clock domain for the compute cores, meaning that instead of issuing 8-wide SIMD twice for the same fetch, a 16-wide SIMD group is provided that is clocked at the same rate as the rest of the chip. This change was made for power reasons, but has the added marketing benefit of increasing the number of cores relative to previous NVIDIA architectures to approximately double. Therefore, a single GK104 has 1536 cores, grouped into 8 streaming multiprocessors (SMX) with 192 cores in each. A SMX has 12 independent 16-wide SIMD units that are capable of executing SPMD code. As a result, a single GK104 can fetch and execute up to 96 different instructions on each cycle. Compared to Fermi, the Kepler architecture has a higher amount of compute per SMX than the GF100. Kepler also exists as a number of lower powered parts designed for laptops and mass market computing devices. The top end Kepler part is the GK110 [101], which has a similar SMX to the GK104, but with 15 SMXs on a chip for a maximum of 2880 CUDA cores. Additionally, the L2 cache is 1536KB, which is twice as large as the GF100 or GF110.

6.1.3 AMD Cypress and Cayman

The ATI/AMD architectures that directly competed with Fermi are known by the codenames, Cypress [102], and Cayman. Cypress is based on a VLIW set of four single-precision ALUs along with a special purpose ALU (for things like branches and advanced arithmetic), often called VLIW5. These VLIW blocks are placed in clusters of 16 that share a 8 KB L1 cache and 32 KB of local memory. Each cluster is called a SIMD core, which leads to the same problems with divergent branching that exist in

the Fermi architecture. A set of 20 SIMD cores share a global register file and a 256 KB L2 cache resulting in 320 VLIW cores capable of issuing up to 1600 operations per cycle.

Cayman [103] switched to a VLIW4 implementation, opting to drop one of the single-precision ALUs while keeping the special purpose ALU. This switch makes sense as high utilization of a VLIW execution model requires either highly optimized assembly code, or a very efficient compiler capable of extracting instruction level parallelism (ILP) from the source code. The step to VLIW4 can be seen as a step from the Cypress-style VLIW5 towards the purely SIMD model found in GCN and in the NVIDIA competition. Since VLIW4 has fewer execution units per SIMD cluster than VLIW5, while Cayman increases from 20 to 24 SIMD cores, the total number of execution cores drops from 1600 to 1536. AMD still makes and sells GPUs that use the VLIW4 and VLIW5 execution models, and they can be found in any of the AMD Fusion APUs [104] among other products. Cayman has a 512KB L2 texture cache as well as smaller 8KB L1 texture caches for each SIMD cluster.

6.1.4 AMD Graphics Core Next (GCN)

AMD's Graphics Core Next is their most recent architecture, and it is set to compete at the top end with NVIDIA's Kepler. The biggest change in the GCN architecture is the departure from VLIW-based execution that AMD has espoused in their GPUs previous to GCN. This major shift in execution models is largely driven by a desire to provide much more efficient execution for general purpose computing which is becoming more important for GPUs. While many graphics shaders have traditionally been able to leverage compiler scheduling to extract high levels of instruction level parallelism (ILP), more general purpose code has a harder time discovering ILP. Instead, they have chosen to use pure SIMD of single execution units, replacing SIMD with VLIW execution units. Similar to Kepler, GCN uses 16-wide SIMD, but groups four SIMD clusters in what they call a compute unit (CU). This CU has a 16KB L1 data cache as well as some read only texture and instruction caches shared among four CUs. The top end GCN parts have a total of 2048 cores, which indicates that 128 independent instructions can be fetched and

issued on each cycle. This is a huge increase from the 24 independent instructions issued that were capable on the previous VLIW4 design.

6.2 Low Power Commercial Architectures

Recent years have seen a huge increase in the computational power and popularity of mobile devices. Smart phones are dominating the cell phone market and low-power tablet computing devices are becoming increasingly popular. Some of the advantages over more traditional computing platforms are that these devices are always available, are usually connected to the network, and have support for advanced graphics. Graphics support is important not only for graphics-intensive user interfaces, but increasingly because the applications themselves require high-quality graphics. Mobile computing applications are being deployed in situations ranging from medical, to scientific applications where visualizing data quickly and accurately is essential. Even in mobile computing, interactive computer graphics architectures are currently dominated by single instruction, multiple data (SIMD) hardware accelerators executing some variant of triangle rasterization [3].

A potential advantage of ray tracing for mobile platforms is that first-order performance scales linearly with the number of screen pixels. The inner loop of a ray tracer iterates over the pixels, which can each be processed independently. The hierarchical acceleration structure allows the search of the scene data to behave roughly logarithmically in the number of primitives, whereas first-order rasterization performance scales linearly with the number of scene primitives. Some culling based on scene partitioning is possible, but in general rasterization time grows with the number of geometric primitives. For a mobile device, the number of pixels is not expected to grow dramatically. An iPhone4 Retina display (640x960) is reported to be roughly at the resolution of the human eye already [105]. A tablet such as an iPad (1024x768) [106] or Samsung Galaxy (1280x800) [107] has somewhat higher pixel count because of larger screen size. Scene data can be expected to increase in size and complexity as new applications are explored [38]. Mobile ray tracers will be able to handle larger scenes as the memory capacity of mobile devices increases.

6.2.1 Tegra

Tegra [108] is a commercial System on Chip (SoC) design from NVIDIA targeting mobile computing devices such as cell phones, media players and tablets. An important part of the SoC is the inclusion of a graphics accelerator intended for rasterization. While rasterization and ray tracing share some of the same shading requirements, ray tracing more naturally handles hidden surface removal, indirect lighting, and shadow effects. Ray tracing has been performed on NVIDIA's discrete GPU solutions, however, current Tegra chips do not have the same unified compute architecture yet and would likely perform ray tracing poorly. A comparison of mobile accelerator compute capabilities, including the graphics accelerator from Tegra 2, can be found in Table 5.13. While Tegra 3 and Tegra 4 have since been released to the market, they were released after this comparison was performed.

6.2.2 PowerVR

PowerVR [109] is an architecture that does very similar computations to those done by Tegra chips. The main distinction of the PowerVR parts is that they separate the image into a set of screen tiles that can be independently processed. Triangles that overlap each screen tile are placed into corresponding geometry bins prior to hidden surface removal. Visibility is then determined by performing a simple ray cast for each pixel and each primitive in the tile. The professed benefit of the tile-based approach is that with accurate depth information, the renderer can avoid processing fragments for many of the hidden surfaces that would not contribute to the final display color. Ray tracing similarly removes hidden surfaces prior to processing fragments, but is capable of retaining access to global scene data. Similar to the Tegra, PowerVR chips are designed for rasterization and can perform ray tracing with some difficulty, despite the use of ray casting for hidden surface removal, because global scene data are not retained.

6.3 General Purpose Architectures

General purpose Central Processing Units(CPU) are the basis of nearly all computing platforms these days. Their history is long and varied, and in recent years the trends have been towards an increase in the throughput processing capabilities

without compromising the single threaded performance. One way this can be achieved is through the addition of multiple hardware supported thread contexts which can be switched to in a fine-grained manner (every cycle), in a course grained manner (every N cycles), or can both issue on the same cycle. Some commercial architectures that have included support for multiple threads include the Intel Netburst architecture [110], the IBM Power5 architecture [111], and the Sun Niagara [112]. While CPU architectures are able to reduce the time it takes to trace a single ray, the ray processing capabilities of these architectures are limited due to the increased area and power overhead of accelerating that single ray.

6.3.1 SIMD Extensions to CPUs

CPU SIMD is a fundamentally different way of performing SIMD operations than the way GPUs do SIMD, despite the similarities in actually building the units. In a GPU, there are independent threads of execution that will hopefully always perform the same operations with just a difference in the data being used. In order to do these operations on a GPU the functional units must be replicated N times for N-wide SIMD. In contrast, CPU-style SIMD comes from a single thread of execution. While executing a single thread, if there is some data parallelism available within that thread's data then special SIMD instructions invoke a special mode of execution for the wide ALUs that are also used for regular arithmetic. For instance, a 64-bit ALU could perform four 16-bit operations in parallel with a very small overhead to reconfigure the unit. It is this kind of SIMD that is done on CPUs and that can also be called Vector processing.

The Cray architecture [113] was one of the early vector processing machines. It introduced the use of vector registers that would be used to perform vector operations similar to how SIMD works in CPUs today. The primary use of these vector registers and functional units were to load a set of data once and perform a series of operations on it before storing the results back in memory. The Cray architecture also had a feature called "chaining" which is essentially the same as register forwarding.

6.3.2 Cell Architecture

The IBM Cell processor [114, 115] is an example of an architecture that seems 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 128×128 register file, 256kb of local memory (not a cache), and four floating point units operating in vector mode. 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 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. TRaX is able to improve performance while not relying on the use of coherent packets to extract SIMD performance, also resulting in less programmer effort.

6.3.3 Larrabee, Intel MIC, and Xeon Phi

Larrabee [45] is an architecture from Intel that consists of many simple x86 cores connected in a bidirectional ring network that is used to keep the caches for each individual core coherent. Each core implements a 16-wide SIMD execution unit similar to the SIMD extensions for CPUs where a single thread issues the instruction for all 16 pieces of data. Since Larrabee is based on x86 cores, it 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. The use of a ring network that communicates between local caches adds complexity to the architecture. Larrabee has since changed focus from visual computing to target the High Performance Computing (HPC) field instead. The Knight's Corner chips, also called MIC (many integrated cores) [116] and commercially branded as "Xeon Phi," are scheduled to be released in early 2013 with around 50 Larrabee cores. These Xeon

Phi chips may well be used for high performance real time ray tracing, though their price point will likely be much higher than that of current commercial GPUs.

6.4 High Performance Research Architectures

Researchers have developed special-purpose hardware for ray tracing [117, 118]. The most complete of these are the SaarCOR [119, 39] and Ray Processing Unit (RPU) [40, 78] architectures from Saarland University. While research architectures rarely turn into commercial products, they provide a good range of exploration of the kinds of techniques that might be useful in the future.

6.4.1 StreamRay

StreamRay [120, 121] takes a different approach to solving the coherence problem in ray tracing. Instead of designing an architecture that can handle divergent threads, they try to group rays into groups that will be coherent. This is done by some special purpose filtering hardware that then allows the use of wide SIMD execution. Each stage of execution includes a filter operation that separates active and inactive rays into groups. The next stage of execution chooses only rays from the active group to continue. They consider SIMD widths of 8, 12, and 16 for their execution model, so in many ways StreamRay proposes a method that could be used to increase the performance of other SIMD architectures. The thing those architectures lack compared to StreamRay is the addition of ray filtering hardware and a group of address generation units.

6.4.2 Rigel

Rigel [122] is a 1000-core tiled architecture that is in many ways similar to TRaX. Both have a large shared cache per tile, with an interface to the rest of the chip. In addition, the individual cores in both have the ability to execute independent of the other threads in their cluster. Rigel does have some important differences when compared to TRaX. For instance, Rigel does not share functional units or instruction caches like TRaX does. These features of TRaX are what allows high utilization to continue even with a reduced number of resources consuming area. In addition, Rigel does not have a ray tracer written specifically for it and the one ray tracing

benchmark they test does not report frame rates or numbers for how many rays it can process per second. Rigel is a dual-issue in-order architecture with many threads tiled on the chip.

The later work of [123] involves a more in-depth exploration of the memory model for Rigel. They implement a hardware cache coherence protocol as well as maintaining coherence through software and allowing for switching between the two methods for coherence. Their set of benchmarks does not include ray tracing, but they show results that are positive for the benchmarks they do test.

6.4.3 RPU

One of the first custom architecture designs specifically for ray tracing was implemented on an FPGA [39, 40, 78]. The RPU (and SaarCOR before it) took the fixed-function approach of the original GPUs but targeted at ray tracing, that being the direct translation of the algorithm to fixed-function hardware components that pass data along as they are processed. A set of programmable execution units was also provided to allow code to be written for different kinds of intersection and shading computations to be performed in 4-wide SIMD. After a single set of these units was shown to work, the design was replicated four times to fill the FPGA that was used. A downside to the fixed function parts of the RPU is that only kd-tree acceleration structures could be used. Another drawback is that programming for the RPU is clumsy and resembles shader programming for GPUs, something that only the most ambitious of hackers even attempt.

6.4.4 Copernicus

Govindaraju et al. [44] presented a design, called Copernicus, that is also similar to this work. Similar to Rigel, they present a many-core tiled architecture with fairly standard CPU cores. In fact, many of the features of CPU cores that are absent in TRaX cores are still present in Copernicus. Each core has a full set of functional units and caches provisioned for it. They are able to fit 128 Copernicus cores using a 22 nm process in about the same area as the proposed TRaX design. Since they use 4-wide multithreading, the direct comparison is 512 threads at 22 nm for Copernicus

and 2560 threads at 65 nm for TRaX. Some Copernicus ray tracing results can be found in Table 5.11.

6.5 Low-Power Ray Tracing Research

While ray tracing is known for requiring large amounts of processing power to work in real time, there have been some efforts to consider what would be needed for mobile chips to perform ray tracing. The mobile space is interesting because a number of devices are already reaching pixel densities that are high enough that the human eye cannot benefit significantly from increased density. Such a fixed resolution target allows for interesting trade-off analysis for real time ray tracing.

6.5.1 ENCORE

Lohrmann [124] presents a method for performing ray tracing on more traditional GPU-style architectures. Ray tracing is expressed as vertex and fragment shader programs that execute within the traditional rasterization pipeline and operate on scene data stored within the texture and buffer memories of the GPU. While Lohrmann's approach is a useful way to repurpose existing hardware, our architecture is designed to have the exact hardware resources needed for ray tracing. In addition, Chang et al. [125] find that bounding volume hierarchies (BVH) are the most energy efficient acceleration structure on both CPUs and GPUs. This dissertation uses a BVH exclusively to reduce power consumption.

6.5.2 MRTP

Kim et al. [83, 84] demonstrate their Mobile Ray Tracing Processor (MRTP), which is similar to most SIMD targeted ray tracers in that they experience the difficulty of dealing with the SIMT execution model. Their approach is to allow the architecture to dynamically reconfigure a hybrid vector SIMD configuration with fewer dependent threads of execution. However, to ensure high vector utilization, the SIMD threads must be able to find opportunities to issue 3-wide vector operations. While this dynamic reconfigurability is interesting, we employ a MIMD design to allow for more thread flexibility. The MRTP achieves a peak performance of 673K rays/sec using 16 mm^2 in a 0.13 μm process running at 100 MHz on a small scene.

The MRTP only executes 103K rays/sec for the much larger dragon model, which is representative of the size of modern scenes. Their work is the best point of comparison for mobile ray tracing accelerators, hence Table 5.14 provides comparison with their best case performance, naively area and frequency scaled to 65nm and 500 MHz. Anido et al. [126] also synthesize an architecture for interactive ray tracing in a 0.13 μm process that only consumes 0.125 mm^2. However, their work tests only very simple scenes and does not use an acceleration structure, making it a poor point for comparison to the work presented here.

6.6 Conclusion

There are many different architectures that can be used to perform ray tracing. While many of them show interesting and effective uses of hardware for ray tracing in particular, there is still room for further exploration. In particular, general purpose architectures have been designed to accelerate all kinds of computation, but at the expense of specific applications. Special purpose graphics processing units are very good at rasterization operation, but fall short of achieving peak hardware utilization for ray tracing due to differences in the two algorithms. Some specialized pipelines have been designed for ray tracing, but they restrict the development of future ray tracing algorithms and software optimizations by including large portions of fixed function in the design. There is room in the existing space for an application-tuned programmable architecture along with algorithmic changes that can map well to efficient execution hardware.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation, I have presented an exploration of a number of different architectural techniques that have been used to accelerate graphics applications in the past and present. I have also proposed a direction for future graphics architectures to progress in order to provide higher performance for ray tracing in particular. The nature of the ray tracing algorithm differs from the rasterization techniques used by current and past dedicated graphics architectures enough to justify novel techniques. In particular, the traversal of acceleration structures naturally creates divergence of both control flow, and memory access between different rays. General purpose caches are able to reduce the impact of divergent memory access when coupled with smart ray-thread assignment. MIMD computation is better suited to divergent control flow than the SIMD compute that is so prevalent in both CPU and GPU architectures that are in production today. The TRaX model of allowing threads to diverge while reducing overhead by sharing lesser-used functional and memory units is more effective than the resulting reduction in SIMD utilization that necessarily occurs with competing architectures.

With the help of a cycle-accurate simulator, I have demonstrated a number of results indicating the above claims. When compared to a highly optimized CPU ray tracer that uses CPU-style SIMD, the slowdown in TRaX is much reduced when operating on highly divergent groups of rays. TRaX slows down to 97% to 99% of top speed on the test scenes in Section 5.1 when the secondary rays become highly incoherent, while the CPU ray tracer, Manta, slows down to 47% to 53% of top speed on the same scenes with the same mechanism for controlling ray coherency. I attribute the difference to the overhead of dealing with small packets and the breakdown of the SIMD operation as the packets become highly incoherent. An added benefit of the

TRaX method when compared to the CPU ray tracer is that the programmer effort is reduced by only writing a single-threaded ray tracer with very simple mechanisms for parallelization and no need for SIMD intrinsics or data management.

In comparison to GPU-style SIMD compute, TRaX has also been shown to compare favorably. The TRaX solutions demonstrate speed-ups from 2.3x to 9.8x in raw performance and 8x-32x faster (6x-10x on average with generous area scaling for the GPU) in performance per area over the best reported GPU-based ray tracer. While GPU architectures have improved somewhat over the particular GPU in question (GTX 285), the most important step for ray tracing, that of providing large quantities of MIMD threads, has not yet appeared in any GPU. The improved caches in the Fermi (GF100) architecture from NVIDIA [97] bring performance gains of 4x with twice the compute resources, representing an improvement to the memory divergence problem. Therefore, further TRaX scaling to newer process sizes would produce similar performance per area improvements for ray tracing, perhaps reduced by a factor of 2.

In comparison to other architectures designed for graphics on an SoC, TRaX is able to provide similar maximum numbers of floating point operations per second, while more effectively using those computation units in performing ray tracing. The mobile space is compelling when looking at the future due to the ubiquity of these devices, and the potential applications for high performance graphics. There exists more of a market for exploration of new ideas through the inclusion of small ray tracing oriented IP blocks along with associated application developer support. It is possible that the first systems to begin to ship with hardware ray tracing support will be mobile or embedded devices where backwards compatibility with existing software is not as essential as desktop computing environments. In particular, a low-power home console entertainment device could be produced that would target the high definition resolution of 720p available on all current HD televisions with real time ray tracing capabilities.

7.1 TRaX-style Programs

While this dissertation has focused almost entirely on ray tracing and accelerating ray tracing, the TRaX architecture itself is completely capable of general purpose

computation. Due to the optimizations of the architecture for ray tracing, some applications will not run as well on a TRaX-style architecture as on a general purpose CPU or GPU. The class of applications that would work well on TRaX should exhibit the following application features:

- A majority of the run-time can be performed by parallel threads.
- The threads all execute the same program.
- Infrequent need for synchronization among the threads.
- Mostly read-only shared memory, ideally with independent threads writing to nonshared memory only.
- A mostly nonuniform dynamic instruction pattern (e.g., not 90% fpadd).

For applications that do not exhibit all of these features, adjustments could be made to the TRaX architecture to facilitate them. For programs that are mostly scalar, meaning they gain little to no benefit from additional threads, the cores would need to support instruction level parallelism features, such as out-of-order and super scalar processing. When a number of different programs are desired to be running in parallel, additional instruction caches could be provided, or mechanisms could be added to allow for rapid instruction cache context switching. In order to support frequent thread synchronization, a number of hardware synchronization options could be pursued, most of which would be expensive to invoke, but would likely cause little overhead when not used. If an application needs a large number of shared memory pages, then the caches and memory system could be supplemented with coherence techniques that are well known and standard at an increase in the area and power consumed. SIMD is likely to be a good solution to high levels of uniform instruction issue.

7.1.1 SIMD Efficiency in TRaX

An experimental SIMD mode has been implemented in the simulator to enable SIMD experiments. When SIMD execution is enabled, the threads within a TM arrange themselves in SIMD blocks for execution. After one instruction is fetched by

one thread in a block, further fetching is stalled until all of the threads have been able to issue that instruction. This ensures that SIMD blocks do not diverge due to contention for functional units or variable latency operations, such as memory loads. This synchronization on coherent fetches is essential to prevent the performance in SIMD mode from approaching 1/N of the MIMD performance for N-wide SIMD (essentially each block issuing only one instruction at a time).

Since the TRaX compiler does not have any concept of SIMD clusters of threads, there is no automated way for SIMD blocks to recognize when their execution should be forced to rejoin after diverging. A SIMD synchronization instruction exists that the programmer can place in the assembly to suggest to the threads that they should synchronize at that point. This instruction allows threads that have diverged for one reason or another to return to a good state where they will be able to once again fetch and issue with SIMD efficiency that is greater than 1/N. The TRaX SIMD experiments, seen in Table 7.1, have shown performance as high as 1.5/8 SIMD utilization for 8-wide SIMD running our ray tracer with a few synchronization instructions. Ultimately, adding SIMD support to the compiler is outside the scope of this dissertation, but it would enable groups of threads to be able to regain their synchrony. A good indication for the best SIMD efficiency that compiler and programmer effort could provide to a ray tracer can be found in the results reported on commercial GPUs [2]. For primary rays, they achieve SIMD efficiency of 56-90% for primary rays and 30-64% for less coherent diffuse rays on what is effectively 16-wide SIMD.

7.1.2 TRaX Rasterizer

I wrote a basic rasterizer using the compilation tools discussed in Chapter 4 based on ideas from Capens [127] as well as Shirley [62]. While the rasterizer did not produce interesting results to present in this dissertation, the implementation exposed a few weaknesses of the TRaX architecture with respect to rasterization. First, rasterization requires a region of shared memory that is not found in ray tracers, the z-buffer. Advanced rasterizers also add other buffers that are shared written regions of memory. All of these buffers either require some tricks or some amount of memory coherence

Table 7.1: SIMD Performance: Conference Scene

Image Size	MIMD	SIMD4	SIMD8
32x32	1	0.302	0.175
256x256	1	0.303	0.185

to function correctly, and these features are not readily available in the basic TRaX architecture. Memory coherence is something we have been able to essentially ignore when considering the ray tracing algorithm, so that it is an additional part of getting the rasterizer to work with some architectural modifications. One way that coherence might be maintained is by using a version of the SWEL protocol [128] to push the shared buffers to the shared level of memory and ensure that they stay there as long as that is beneficial for performance. Since there are currently many caches in the TRaX architecture, it might be necessary to implement some smart cache data placement techniques such as R-NUCA [129]. In combination with these additional hardware features, a rasterizer could be used for primary visibility with ray tracing performing a second pass on the image to increase the quality of illumination. In such a system with multiple applications it may be necessary to allow the caches to efficiently share capacity with a scheme like Dynamic Spill-Receive [130].

7.2 Future Work

As rays diverge and threads begin to process highly diverged rays, the efficiency of the cache-based memory system begins to degrade. A potential improvement to coherent ray processing in a parallel ray tracing system would involve some sort of ray reordering. Streamray [120] proposes the use of hardware ray streams in order to facilitate on-chip reordering of rays after every node intersection. Another approach [131] reduces the frequency with which rays need to be reordered through the use of treelets. A treelet is a group of nodes within a BVH (or other tree structure) that are nearby in the tree. In order to incorporate both of these ideas, TRaX would leverage treelets in order to find portions of the acceleration structure that can fit in the L1 cache of a given TRaX TM. Assuming enough rays could be kept in flight at once, the regions of memory associated with a given treelet can be brought in

once and held for a large amount of computation prior to evicting those data. This increased cache efficiency should also reduce the amount of off-chip bandwidth at the expense of on-chip data traffic. Additionally, once rays within a TM are coherent enough, the shared functional units can be chained together to reduce the energy consumed by instruction fetch and register accesses [51].

Noise and other procedural texture generation would be interesting in other applications to analyze on both SIMD and MIMD modes of the TRaX simulator. Noise textures would use a noise kernel that would be called many times per pixel to sample the noise at many different frequencies and composite the contributions to one pixel. For example, Perlin noise techniques [65, 66] increase FP ops by about 50% in the worst case, but have a negligible impact on memory bandwidth. The expectation would be that noise texture generation performs very well on SIMD or MIMD hardware since there are no points where the code can diverge. The impact of a custom functional unit [132] for noise computation on an entire rendering system would also be an interesting piece of data to gather.

It could be possible to allow the TRaX simulator to change between SIMD and MIMD on the fly. The simulator even includes basic support for switching SIMD mode on and off dynamically, or even varying SIMD width at run time, though no testing has yet been performed. For performance reasons, SIMD should only be switched on when each of the SIMD blocks is synchronized. SIMD could be switched on right before a SIMD synchronize to cause the block to synchronize soon after switching to SIMD mode. It is unlikely for performance to be good in both modes because the number of execution units that must be provisioned is likely to vary greatly based on which one is chosen. The experiment could be quite interesting because the shading phase of ray tracing is very easy to perform in SIMD when there are a small number of shaders being executed. The primary benefit of dynamic SIMD switching would be the reduction of width for the fetch and issue stages of the TRaX architecture.

While it may seem counter-intuitive that MIMD can work efficiently in comparison to the SIMD execution model, most other MIMD systems do not share expensive computational units. In addition, the programming effort required to exploit parallelism in a SIMT system is quite high. Parallelism can be further increased

by allowing the threads to occasionally execute vector instructions, increasing the burden on the programmer. While automated tools can be developed to ease the required programming effort, these tools are not yet widely available. In contrast, the programming effort required to write a ray tracer for a MIMD architecture, such as the one proposed in this dissertation, is greatly reduced. The programmer can rely on the hardware to exploit parallelism through the proper use of shared resources.

The TRaX programming model is among the most simple parallel programming models in existence. It is certainly much simpler than GPGPU models such as CUDA or OpenCL, which have additional complications and difficulties added in due to their origin and development process. With TRaX, we set out to keep things as simple as possible and to only include the features that were necessary for the simple thread communication model. While the model as it exists for the work presented here is quite useful, capable and simple to use, there are a couple improvements that would greatly increase its usability for general purpose applications. In particular, it is more difficult at present to access objects and data structures in global memory since they are only loaded and stored through explicit intrinsic operations. This means that the TRaX C code must encapsulate the access intrinsics with functions to read and write. Additionally, none of the data structure features of high-level languages like C++ are directly exposed for global memory in TRaX C.

A simple solution that has not been implemented is to include an option in the language for "global" or "local" on any of the variable declarations with global variables requiring explicit pointers for the predefined structures, and perhaps explicit allocation for those dynamically allocated items. This simple extension to the current programming model would allow arbitrary parallel code to be recompiled for TRaX, allowing for simulation on many thousands of simple, throughput-optimized cores.

Power is an important consideration for any integrated circuit design, especially since all processors hit a wall in power consumption where performance ends up throttled because of thermal and energy limits on the designs. While GPUs typically are allowed to consume more power than other circuit designs, especially at the high end, they are still heavily constrained by power. The results presented in this dissertation represent a significant energy savings per ray over implementing a ray

tracer on either a CPU or a GPU since each consumes power for functionality that is not necessary in a dedicated ray tracing processor. As detailed in Section 5.2.3, a rough power analysis shows a high-powered TRaX architecture consumes 83 W, however, the power number alone does not tell the whole story. To truly compare any ray tracing implementation against the competition, the real metric of comparison should be the energy per pixel at some relatively fixed or even level of quality for each image. In other words, an image should be generated by a rasterizer within some decibels (dB) of the full path traced solution and profiled for energy consumption. Then a similar image should be generated with a ray tracer at that same dB from the path traced solution and profiled for energy. These two numbers would then give about as fair a comparison as possible for the two techniques, though a wide variety of scenes of interest should also be included. Unfortunately, this detailed comparison is outside the scope of this work, but I believe a good hardware design for ray tracing would be within a 2-5x energy per pixel when given an acceptable closeness to the path tracing result for future video game scenes. As the chosen comparison point moves farther from the path traced solution, rasterization is likely to be more efficient, while ray tracing is likely to be better close to the high-quality end-point.

Hardware companies that look at implementing a chip specifically for ray tracing may be tempted to add SIMD due to the low cost of including SIMD on top of a MIMD execution pipeline. An example where SIMD was given even when many of the target applications may not have desired it is the Larrabee architecture, now called Xeon Phi [45]. While I believe strongly that MIMD is the best execution model for ray tracing, and particularly for the traversal stage of the ray tracer, there are well explored ways of exploiting additional parallelism available if SIMD hardware appears anyway. In particular, the Embree ray tracing kernels have shown how to effectively use SIMD to both traverse wide BVHs (where there are more than two children for each node) and a bundle of relatively coherent rays at a 4-wide parallelism for each on the Intel Xeon Phi architecture [14, 13, 133]. The inclusion of 16-wide SIMD in Xeon Phi is likely due to the architecture being designed for a wider variety of applications than just ray tracing in contrast to the TRaX architecture. The usefulness of SIMD is a function of the overhead of adding SIMD as compared to the performance increase

achievable by adding SIMD-capable software capabilities to the system. Ray tracing traversal is well documented to have much smaller increases in performance from including SIMD capabilities than many other graphics and multimedia applications, so it depends on a low-overhead SIMD implementation to be practical. Since any commercial architecture designed for ray tracing is likely to also be designed for other applications that can extract more benefit from SIMD, it is likely that any real ray tracing application with need to use SIMD in some fashion to optimize performance, even if it is not the most power-efficient implementation.

REFERENCES

[1] Lux Render. http://www.luxrender.net/en_GB/index.

[2] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proc. High Performance Graphics*, (New York, NY, USA), pp. 145–149, ACM, 2009.

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

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

[5] A. Glassner, ed., *An Introduction to Ray Tracing*. London: Academic Press, 1989.

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

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

[8] J. T. Kajiya, "The rendering equation," in *Proceedings of SIGGRAPH*, pp. 143–150, 1986.

[9] Khronos Group, "OpenGL - The Industry's Foundation for High Performance Graphics." http://www.opengl.org/.

[10] Microsoft, "DirectX: Advanced Graphics on Windows." http://msdn.microsoft.com/en-us/directx/default.aspx.

[11] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[12] G. Riguer and B. Bennett, "Mantle: Empowering 3d graphics innovation," 2013. http://amd-dev.wpengine.netdna-cdn.com/apu/wp-content/uploads/sites/3/2013/11/GS-4112_Guennadi_Riguer-final.pdf.

[13] S. Woop, L. Feng, I. Wald, and C. Benthin, "Embree ray tracing kernels for cpus and the xeon phi architecture," in *ACM SIGGRAPH 2013 Talks*, p. 44, ACM, 2013.

[14] M. Ernst and S. Woop, "Embree: Photo-realistic ray tracing kernels," tech. rep., Intel Corporation, 2011.

[15] A. Kensler and P. Shirley, "Optimizing ray-triangle intersection via automated search," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 33–38, September 2006.

[16] T. Möller and B. Trumbore, "Fast, minimum storage ray triangle intersection," *Journal of Graphics Tools*, vol. 2, pp. 21–28, October 1997.

[17] T. Ize, *Efficient Acceleration Structures for Ray Tracing Static and Dynamic Scenes*. PhD thesis, The University of Utah, August 2009.

[18] T. Ize, I. Wald, and S. G. Parker, "Grid creation strategies for efficient ray tracing," in *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, pp. 27–32, May 2007.

[19] C. Lauterbach, S.-E. Yoon, D. Manocha, and D. Tuft, "RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs," in *Symposium on Interactive Ray Tracing (IRT06)*, pp. 39–46, 2006.

[20] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," *Computer Graphics Forum*, 2007.

[21] I. Wald, "On fast construction of SAH based bounding volume hierarchies," in *Symposium on Interactive Ray Tracing (IRT07)*, 2007.

[22] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Transactions on Graphics (SIGGRAPH '05)*, vol. 24, no. 3, pp. 1176–1185, 2005.

[23] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. Parker, "Ray tracing animated scenes using coherent grid traversal," in *Proceedings of the ACM SIGGRAPH 2006 Conference*, vol. 25, (New York, NY, USA), pp. 485–493, ACM Press, 2006.

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

[25] T. Ize, I. Wald, C. Robertson, and S. G. Parker, "An evaluation of parallel grid construction for ray tracing dynamic scenes," in *Proceedings of the 2006 IEEE/Eurographics Symposium on Interactive Ray Tracing*, pp. 47–55, 2006.

[26] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," in *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, (New York, NY, USA), pp. 66:1–66:13, ACM, 2010.

[27] NVIDIA, "NVIDIA's next generation cuda compute architecture: Fermi," tech. rep., NVIDIA Corporation, 2009. http://www.nvidia.com/object/fermi_architecture.html.

[28] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on gpus," in *Proc. High Performance Graphics*, 2013.

[29] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, "Ray tracing on the CELL processor," in *Interactive Ray Tracing IRT06*, Sept. 2006.

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

[31] E. Reinhard, C. Hansen, and S. Parker, "Interactive ray tracing of time varying data," in *Eurographics Workshop on Parallel Graphics and Visualization*, pp. 77–82, 2002.

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

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

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

[35] H. Gouraud, "Continuous shading of curved surfaces," *IEEE Transactions on Computers*, pp. 623–629, 1972.

[36] B. T. Phong, "Illumination for computer generated pictures," *Communications of ACM*, pp. 311–317, 1975.

[37] J. F. Blinn, "Simulation of wrinkled surfaces," *SIGGRAPH Comput. Graph.*, pp. 286–292, August 1978.

[38] 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*, pp. 477–494, 1997.

[39] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime ray tracing of dynamic scenes on an FPGA chip," in *Graphics Hardware Conference*, pp. 95–106, August 2004.

[40] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics (SIGGRAPH '05)*, vol. 24, July 2005.

[41] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *Symposium on Interactive Ray Tracing (IRT07)*, pp. 113–118, 2007.

[42] M. L. Anido, N. Tabrizi, H. Du, M. Sanchez-Elez, N. Bagherzadeh, *et al.*, "Interactive ray tracing using a simd reconfigurable architecture," in *Computer Architecture and High Performance Computing, 2002. Proceedings. 14th Symposium on*, pp. 20–28, IEEE, 2002.

[43] H. Du, A. Sanchez-Elez, N. Tabrizi, N. Bagherzadeh, M. Anido, and M. Fernandez, "Interactive ray tracing on reconfigurable simd morphosys," *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pp. 471–476, 21-24 Jan. 2003.

[44] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a multicore architecture for real-time ray-tracing," in *IEEE/ACM Micro '08*, October 2008.

[45] 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 Transactions on Graphics*, vol. 27, August 2008.

[46] D. Kopta, J. Spjut, E. Brunvand, and S. Parker, "Comparing incoherent ray performance of TRaX vs. Manta," in *Interactve Ray Tracing IRT08*, August 2008.

[47] J. Spjut, D. Kopta, S. Boulos, S. Kellis, and E. Brunvand, "TRaX: A multithreaded architecture for real-time ray tracing," in *6th IEEE Symposium on Application Specific Processors (SASP)*, June 2008.

[48] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A multicore hardware architecture for real-time ray tracing," *IEEE Transactions on Computer-Aided Design*, vol. 28, no. 12, pp. 1802 – 1815, 2009.

[49] D. Kopta, J. Spjut, E. Brunvand, and A. Davis, "Efficient MIMD architectures for high-performance ray tracing," in *IEEE International Conference on Computer Design (ICCD)*, 2010.

[50] J. Spjut, D. Kopta, E. Brunvand, and A. Davis, "A mobile accelerator architecture for ray tracing," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, 2012.

[51] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "An energy and bandwidth efficient ray tracing architecture," in *High-Performance Graphics (HPG 2013)*, 2013.

[52] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "Memory considerations for low-energy ray tracing," *Computer Graphics Forum*, 2014. To Appear.

[53] S. Inc., "Synopsys design ware and design compiler." http://www.synopsys.com.

[54] "Artisan cmos standard cells." ARM Ltd. http://www.arm.com/products/physicalip/standardcell.html.

[55] D. Burger and T. Austin, "The Simplescalar toolset, Version 2.0," tech. rep., University of Wisconsin-Madison, June 1997.

[56] 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.

[57] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs," in *Symposium on Interactive Ray Tracing (IRT08)*, pp. 49–57, 2008.

[58] M. Ernst and G. Greiner, "Multi bounding volume hierarchies," in *Symposium on Interactive Ray Tracing (IRT08)*, pp. 35–40, 2008.

[59] Chris Lattner and Vikram Adve, "The LLVM instruction set and compilation strategy," tech. report, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[60] I. Xilinx, *Microblaze processor reference guide*. 2006.

[61] The RenderMan Interface. http://renderman.pixar.com/products/rispec/rispec_pdf/RISpec3_2.pdf.

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

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

[64] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *ACM SIGGRAPH Computer Graphics*, vol. 11, pp. 192–198, ACM, 1977.

[65] K. Perlin, "An image synthesizer," *ACM SIGGRAPH Computer Graphics*, vol. 19, pp. 287–296, July 1985.

[66] J. C. Hart, "Perlin noise pixel shaders," in *HWWS '01: Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, (New York, NY, USA), pp. 87–94, ACM Press, 2001.

[67] K. Perlin, "Improving noise," *ACM Transactions on Graphics (SIGGRAPH '02)*, vol. 21, pp. 681–682, July 2002.

[68] ATI, "ATI products from AMD." http://ati.amd.com/products/index.html.

[69] NVIDIA Corporation, "Nvidia website." http://www.nvidia.com.

[70] 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.

[71] F. E. Nicodemus, "Directional reflectance and emissivity of an opaque surface," *Appl. Opt.*, vol. 4, no. 7, p. 767, 1965.

[72] G. J. Ward, "Measuring and modeling anisotropic reflection," in *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 265–272, ACM, 1992.

[73] E. Lafortune and Y. D. Willems, "Bi-directional path-tracing," in *Proceedings of Compugraphics*, (Portugal), pp. 145–153, December 1993.

[74] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, "Fast, effective BVH updates for animated scenes," in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2012)*, 2012.

[75] J. Pantaleoni and D. Luebke, "HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry," in *High Performance Graphics'10*, pp. 87–95, 2010.

[76] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster hlbvh with work queues," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 59–64, ACM, 2011.

[77] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," *Proc. High-Performance Graphics*, 2013.

[78] S. Woop, E. Brunvand, and P. Slusallak, "Estimating performance of a ray tracing ASIC design," in *IRT06*, September 2006.

[79] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," in *Computer Graphics Forum (Proc. Eurographics 2001)*, vol. 20, pp. 153–164, 2001.

[80] J. Bigler, A. Stephens, and S. G. Parker, "Design for parallel interactive ray tracing systems," in *Symposium on Interactive Ray Tracing (IRT06)*, 2006.

[81] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 3–14, IEEE Computer Society, 2007.

[82] S. Bhagwat, "Samsung exynos 5250 begins sampling - mass production in q2 2012." http://www.anandtech.com/show/5467/.

[83] H.-Y. Kim, Y.-J. Kim, and L.-S. Kim, "MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization," *IEEE Journal of Solid-State Circuits*, vol. 47, pp. 518–535, feb. 2012.

[84] H.-Y. Kim, Y.-J. Kim, and L.-S. Kim, "Reconfigurable mobile stream processor for ray tracing," in *Custom Integrated Circuits Conference (CICC), 2010 IEEE*, September 2010.

[85] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "Sgrt: A mobile GPU architecture for real-time ray tracing," in *Proceedings of the 5th High-Performance Graphics Conference*, pp. 109–119, ACM, 2013.

[86] W.-J. Lee, Y. Shin, J. Lee, S. Lee, S. Ryu, and J. Kim, "Real-time ray tracing on future mobile computing platform," in *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications*, p. 56, ACM, 2013.

[87] Y. Shin, W.-J. Lee, J. Lee, S.-H. Lee, S. Ryu, and J. Kim, "Energy efficient data transmission for ray tracing on mobile computing platform," in *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications*, p. 64, ACM, 2013.

[88] J. H. Clark, "The geometry engine: A VLSI geometry system for graphics," in *SIGGRAPH '82: Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 127–133, ACM Press, 1982.

[89] 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 *SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 21–30, ACM Press, 1988.

[90] 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 imgage enhancements in pixel-planes," in *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 111–120, ACM Press, 1985.

[91] 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 *Chapel Hill Conference on VLSI*, 1985.

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

[93] D. Balciunas, L. Dulley, and M. Zuffo, "Gpu-assisted ray casting acceleration for visualization of large scene data sets," in *Interactive Ray Tracing IRT06*, September 2006.

[94] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, pp. 39–55, March-April 2008.

[95] NVIDIA CUDA Documentation. http://developer.nvidia.com/object/cuda.html.

[96] NVIDIA SIGGRAPH Ray Tracing Demo, August 2008. http://developer.nvidia.com/object/nvision08-IRT.html.

[97] NVIDIA, "NVIDIA GF100: World's fastest GPU delivering great gaming performance with true geometric realism," tech. rep., NVIDIA Corporation, 2009. http://www.nvidia.com/object/GTX_400_architecture.html.

[98] NVIDIA G80 Architecture Documentation. http://www.nvidia.com/page/geforce8.html.

[99] NVIDIA, "GeForce GTX 200 GPU architectural overview," tech. rep., NVIDIA Corporation, May 2008.

[100] NVIDIA, "NVIDIA geforce gtx 680," tech. rep., NVIDIA Corporation, 2012. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[101] NVIDIA, "NVIDIAs next generation CUDA compute architecture: Kepler tm gk110," tech. rep., NVIDIA Corporation, 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[102] M. Fried, "GPGPU architecture comparison of ATI and NVIDIA GPUs," tech. rep., Microway, June 2010. http://microway.com/gpu.html.

[103] R. Smith, "AMD's radeon HD 6970 and radeon HD 6950: Paving the future for AMD." http://www.anandtech.com/show/4061/.

[104] N. Brookwood, "AMD fusion family of APUs: Enabling a superior, immersive pc experience," tech. rep., AMD Corporation, 2010. http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf.

[105] C. Brandrick, "iPhone 4's retina display explained," June 2010. PC World, http://www.pcworld.com/article/198201/iphone_4s_retina_display_explained.html.

[106] Apple Computer, "iPad technical specifications," 2012. http://www.apple.com/ipad/specs/.

[107] Samsung, "Galaxy tablet technical specifications," 2012. http://www.samsung.com/global/microsite/galaxytab/10.1/spec.html.

[108] NVIDIA, "Bringing high-end graphics to handheld devices," tech. rep., NVIDIA Corporation, 2011.

[109] POWERVR, "POWERVR MBX technology overview," tech. rep., Imagination Technologies Ltd., May 2009.

[110] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23(2), pp. 56–65, March-April 2003.

[111] R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 chip: A dual-core multithreaded processor," *IEEE Micro*, vol. 24(2), pp. 40–47, March-April 2004.

[112] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25(2), pp. 21–29, March-April 2005.

[113] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

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

[115] 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.

[116] K. Skaugen, "Petascale to exascale: Extending intel's hpc commitment," 2010. http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf.

[117] 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.

[118] D. Hall, "The AR350: Today's ray trace rendering processor," in *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics Hardware - Hot 3D Session*, 2001.

[119] J. Schmittler, I. Wald, and P. Slusallek, "SaarCOR – A hardware architecture for realtime ray-tracing," in *Proceedings of Eurographics Workshop on Graphics Hardware*, 2002. http://graphics.cs.uni-sb.de/Publications.

[120] K. Ramani and C. Gribble, "StreamRay: A stream filtering architecture for coherent ray tracing," in *ASPLOS '09*, (Washington, D.C.), 2009.

[121] C. Gribble and K. Ramani, "Coherent ray tracing via stream filtering," in *Symposium on Interactive Ray Tracing (IRT08)*, 2008.

[122] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *ISCA '09*, 2009.

[123] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: A hybrid memory model for accelerators," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[124] P. J. Lohrmann, "Energy-Efficient Interactive Ray Tracing of Static Scenes on Programmable Mobile GPUs," Master's thesis, Worcester Polytechnic Institute, February 2007.

[125] C.-H. Chang, P. J. Lohrmann, E. O. Agu, and R. W. Lindeman, "Encore: Energy-conscious rendering for mobile device," *Proceedings of GPGPU*, 2007.

[126] M. Anido, N. Tabrizi, H. Du, M. Sanchez-Elez M, and N. Bagherzadeh, "Interactive ray tracing using a simd reconfigurable architecture," in *Computer Architecture and High Performance Computing, 2002. Proceedings. 14th Symposium on*, pp. 20 – 28, 2002.

[127] Nicolas Capens, "Advanced rasterization," September 2004. http://www.devmaster.net/forums/showthread.php?t=1884.

[128] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware cache coherence protocols to map shared data onto shared caches," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 465–476, ACM, 2010.

[129] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proceedings of ISCA*, 2009.

[130] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *Proceedings of HPCA*, 2009.

[131] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proc. High Performance Graphics*, 2010.

[132] J. Spjut, A. Kensler, and E. Brunvand, "Hardware-accelerated gradient noise for graphics," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2009.

[133] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A kernel framework for efficient cpu ray tracing," *ACM Transactions on Graphics*, 2014.