DUAL STREAMING FOR HARDWARE-ACCELERATED RAY TRACING

by

Konstantin Shkurko

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

August 2019

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Konstantin Shkurko**
has been approved by the following supervisory committee members:

Erik Brunvand,	Chair(s)	**11/30/2018** Date Approved
Cem Yuksel,	Member	**11/30/2018** Date Approved
Rajeev Balasubramonian,	Member	**11/30/2018** Date Approved
Valerio Pascucci,	Member	**11/30/2018** Date Approved
Peter Shirley,	Member	**11/30/2018** Date Approved

by **Ross Whitaker**, Chair/Dean of
the Department/College/School of **Computing**
and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

Hardware acceleration for ray tracing has been a topic of great interest in computer graphics. However, even with proposed custom hardware, the inherent irregularity in the memory access pattern of ray tracing has limited its performance, compared with rasterization on commercial GPUs. We provide a different approach to hardware-accelerated ray tracing, beginning with modifying the order of rendering operations, inspired by the streaming character of rasterization. Our *dual streaming* approach organizes the memory access of ray tracing into two predictable data streams. The predictability of these streams allows perfect prefetching and makes the memory access pattern an excellent match for the behavior of DRAM memory systems. By reformulating ray tracing as fully predictable streams of rays and of geometry we alleviate many long-standing problems of high-performance ray tracing and expose new opportunities for future research. Therefore, we also include extensive discussions of potential avenues for future research aimed at improving the performance of hardware-accelerated ray tracing using dual streaming.

To my family

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Computer-generated imagery has become pervasive in the entertainment industry over the past couple of decades: from video games where each image is generated at interactive rates (16 milliseconds) to film and visual effects where each image may take hours to compute. The thirst for visual entertainment drives the development of sophisticated algorithms, software and hardware. While video games rely on specially designed graphics processing hardware, movie studios employ thousands of commodity machines throughout large data centers. Each application relies on a different image generation algorithm and corresponding hardware architectures. The film industry desires photo-realistic images and can tolerate high computation time. The realism is achieved by using an algorithm that can simulate light propagation, ray tracing. Although ray tracing can easily describe all desired lighting effects with a single computational operation, making derived algorithms performant is difficult. Interactive graphics relies on dedicated hardware designed specifically to accelerate a different image generation algorithm, rasterization. Although fast, rasterization can not simulate complex lighting effects without sophisticated approximation techniques.

The two applications are starting to merge: real-time interactive computer graphics strives for film-like visual quality, while the software that generates film images strives for interactivity to improve artists' productivity. Even though ray tracing can simulate complex lighting effects, making it a suitable choice to address the needs of both applications, the major challenge in making ray tracing fast enough for interactive image generation is improving how it accesses memory. Tree-like data structures, common for acceleration, access memory almost randomly, straining the memory subsystem which is designed for sequential access instead. Thus, enabling ray tracing to reach interactive rates requires innovative algorithmic improvements and custom hardware architectures.

1.1 The Memory Wall

Although the number of transistors in processor chips continues to grow, the computational performance of individual processing cores has not been improving as fast. As a result, the total computational performance per chip has been increasing through parallelism by including more cores per chip. Unfortunately, the bandwidth between the on-chip processing cores and the off-chip main memory has not been increasing at a pace similar to the growth in transistor counts. In fact, the available memory bandwidth per core has been decreasing, effectively starving each core of data. To combat this starvation, hardware and software designers must focus their efforts on creating systems that are frugal in terms of accessing and moving data.

Current architectures address the per-core data starvation through a hierarchical memory system. The main memory is at the top of the hierarchy and provides data to the components on lower levels. Individual on-chip components, called caches, store recently used data at different levels in the hierarchy. Caches provide shared access to data between multiple cores, thus reducing total off-chip memory bandwidth required to feed the chip. The data structures used to improve the performance of ray tracing rely on trees to organize the input data. Tree traversal results in accessing tree nodes scattered throughout the memory, which reduces the likelihood of reusing data fetched into the on-chip cache hierarchy and thus requires more data requests to the main memory. Such incoherent access patterns introduce additional latency and energy costs at the main memory level because of the design of the dynamic random access memory (DRAM) chips that implement the main memory. Data access is so important that the memory hierarchy, especially DRAM, requires the largest portion of energy used to generate a single image [76]. Because data movement can be the primary performance limitation and energy consumer in modern computing systems, the random data access has so far has prevented ray tracing from being used for real-time image generation.

An approach to reducing the negative effects related to random memory accesses organizes memory references as a *stream*: a set of contiguous references, or with a fixed stride, that bring in data as a continuous block rather than as individual data elements. Streamed memory accesses improve the performance and energy use for at least three reasons. First, data streaming relieves the processor of address calculation tasks and pointer chasing

when traversing tree-like data structures. Second, streaming helps hide memory latency via prefetching because the next set of data needed for computation is known perfectly in advance. Finally, the circuit-level architecture of the DRAM chips that make up the off-chip main memory is designed for streaming. Each access to DRAM streams contiguous data across the memory interface, typically at least a 64-byte chunk of data, called a *cache line*. Internally, DRAM chips provide fast and low-energy access to even larger blocks of contiguous data about 100 cache lines in size.

Designing problem-specific software along with the dedicated hardware offers a compelling methodology to address the problems introduced by the memory wall [51]. In this spirit, we focus on co-designing hardware and software for path tracing, an algorithm that helps generate photo-realistic images but requires a significant amount of computation and leads to incoherent memory access patterns.

1.2 Dissertation Statement

This dissertation explores approaches to reducing the amount of energy used to generate an image frame using dedicated ray tracing hardware architectures without sacrificing performance or scalability. Given that the primary source of energy consumption for ray tracing of large complex scenes is data movement, specifically to and from DRAM [76], this thesis modifies how ray tracing algorithms and hardware use DRAM to derive synergistic efficiencies. Specifically, this work restructures path tracing into a streaming algorithm that both optimizes the use of scene and ray data, and optimizes the delivery of that data to the processors through making accesses to it perfectly predictive ahead of computation. This work also develops a custom hardware architecture to support this streaming algorithm and evaluates the architecture using cycle-accurate simulation.

1.3 Dissertation Organization

Chapter 2 introduces the necessary concepts from various fields. Chapter 3 describes the previous work aimed at improving ray tracing efficiency in commodity hardware and dedicated ray tracing hardware architectures. Chapter 4 describes the new algorithmic formulation of ray tracing, and Chapter 5 proposes and evaluates the corresponding dedicated hardware architecture. Finally, we conclude in Chapter 6.

CHAPTER 2

BACKGROUND

This work combines concepts from several fields, mainly the simulation of light propagation, data structures, and hardware architectures. This chapter provides only a high-level introduction to the important concepts. Each presented topic can certainly be expanded in significant detail, and the discussion aims to provide plenty of references an interested reader can follow to gain a deeper understanding.

We start by introducing how light behavior is modeled, motivating the formulations and explaining the physical intuition behind them. Then, we discuss how the resulting equations can be solved mathematically and thus the behavior of light simulated computationally. Finally, we introduce basic concepts of dedicated hardware architectures.

2.1 Simulating Light Propagation

Generating images of objects and their compositions specified digitally is one of the core subfields of computer graphics called *rendering*. At a very high level, the process mimics how one might capture a photograph, except it uses computers to simulate how light travels towards the camera. Rendering requires algorithms, data structures, and methods that solve two problems. The first is a model of how light behaves so it can be simulated computationally. The second is a faithful digital representation of objects, including physical dimensions, appearance, movement, etc.

2.1.1 Physics of Light Transport

A high-level intuition behind light propagation can be built through a thought experiment that considers a single "photon" as it travels through a simple scene shown in Figure 2.1. Although in reality there are billions of photons interacting with objects at once, it is easier conceptually to consider a single particle carrying some energy as it travels throughout the scene. First, the photon is created (emitted) at a light source. The

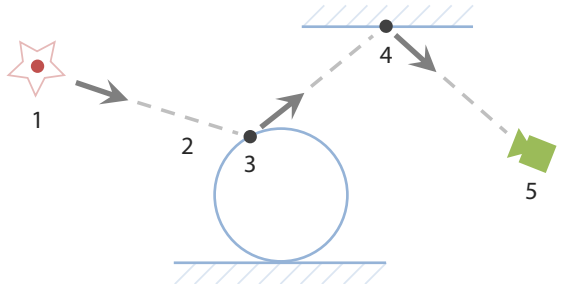


Figure 2.1: A thought experiment that considers a single particle carrying energy traveling throughout a scene, loosely referred to as a photon. The photon is emitted, 1. The photon travels through a scene (2) until it hits a sphere, 3. Photon reflects and proceeds until it hits a wall, 4. Finally, the photon reflects into the camera where it is absorbed by the sensor, 5.

photon then proceeds to travel through the scene until it hits the sphere. There, the photon interacts with the sphere's material and reflects in a new direction but a little dimmer.[1] This process repeats many times. Finally, the photon hits the camera sensor, which absorbs the photon's energy, making a small portion of the image a little brighter. Simulating many such photons eventually produces an image of the scene. Essentially, this process of particle transport connects a light source to the camera sensor through a number of interactions (bounces) with objects in the scene.

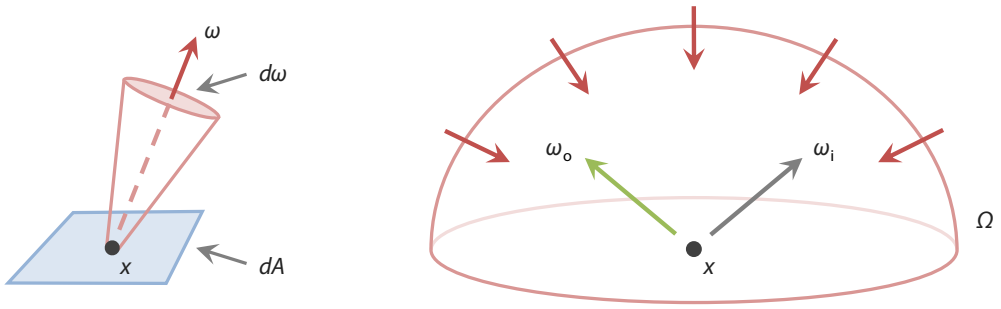
It is more computationally convenient to consider the process of light propagation from the localized perspective of a single surface interaction. Typically the mathematical formulation considers a specific surface location and an outgoing direction from the surface (towards the camera sensor, for example).

2.1.1.1 Radiometric Quantities

The total amount of light that shines onto something is measured as a count of the total number of photons that hit a detector. The physical unit of measurement of this energy Q is Joules (J). However, the energy is measured over time (exposure), which can be used to estimate instantaneous power called *radiant flux* $\Phi = {}^{dQ}/_{dt}$ $\left({}^{\mathrm{J}}/_{\mathrm{s}}\right)$. Other relevant radiometric quantities are derived from radiant flux.

The amount of light arriving from all directions at a surface measured at a point is

[1] Ignoring quantum mechanics, the photon would be absorbed by an outer electron of an atom making up the surface, exciting the electron to a higher energy level. This electron then decays to a lower energy level emitting another photon in some new direction. Many photons would undergo this process at the surface. Some would be absorbed, making the reflection appear dimmer than the original illumination.



(a) Radiance definition **(b)** Rendering equation

Figure 2.2: Local coordinate frames illustrating the parameters used in the definition of radiance (a) and the rendering equation (b).

referred to as *irradiance* $E = {}^{d\Phi}/_{dA}\ \left({}^{\mathrm{W}}/_{\mathrm{m}^2}\right)$, which is radiant flux per unit area. Flux leaving a surface is called *radiant exitance* or *radiosity* B. Considering all energy traveling along a specific direction is called *intensity* $I = {}^{d\Phi}/_{d\omega}\ \left({}^{\mathrm{W}}/_{\mathrm{sr}}\right)$. An angle in 3D is referred to as a *solid angle*, measured in steradians (sr).

Radiance is a measure of light near some location $\mathbf{x}$ traveling near some direction ω, $L(\mathbf{x}, \omega) = {}^{d^2\Phi}/_{dA\, d\omega}\ \left({}^{\mathrm{W}}/_{\mathrm{m}^2 sr}\right)$. It can be considered as a count of photons traveling within a solid angle $d\omega$ from direction ω near a point $\mathbf{x}$ landing on a surface dA perpendicular to the direction ω, Figure 2.2a. Note that one can consider radiance to be irradiance (or radiant exitance) per unit solid angle or an intensity per unit area. Algorithms simulate light propagation by tracking radiance throughout the scene.

2.1.1.2 The Rendering Equation

The *rendering equation* was introduced to the field of computer graphics by Immel et al. [55] and Kajiya [64] in 1986. The rendering equation computes outgoing radiance from a specific location along a specific direction based on all possible light interactions at a surface. The surface can emit light, or it can reflect some of the incident illumination. The reflection can be computed by a sum of light from all possible directions modulated by a reflectivity coefficient based on a specific outgoing direction.

Using the local coordinate frame shown in Figure 2.2b, the outgoing radiance $L_o(\mathbf{x}, \omega)$ at a surface location $\mathbf{x}$ along an outgoing direction ω_o can be computed by the following solid angle formulation:

$$\underbrace{L_o(\mathbf{x},\omega_o)}_{\text{outgoing radiance}} = \underbrace{L_e(\mathbf{x},\omega_o)}_{\text{emitted radiance}} + \underbrace{\int_\Omega \underbrace{f_r(\mathbf{x},\omega_o,\omega_i)}_{\text{reflection coefficient}} \underbrace{L_i(\mathbf{x},\omega_i)}_{\text{incident radiance}} (\omega_i \cdot \mathbf{n})\, d\omega_i}_{\text{reflected radiance: sum over all incident directions in hemisphere}} \tag{2.1}$$

The term $L_e(\mathbf{x},\omega_o)$ evaluates the radiance emitted by the surface at a location $\mathbf{x}$ along an outgoing direction ω_o. The total radiance reflected from the surface along the direction ω_o is computed by the integral over the hemisphere Ω of incident directions ω_i. The integrand contains three parts. The first, $f_r(\mathbf{x},\omega_o,\omega_i)$, is the bidirectional reflectance distribution function (BRDF) which specifies the material properties and computes how much of the radiance that arrives at the location $\mathbf{x}$ from the incident direction ω_i is reflected along the outgoing direction ω_o. The second term, $L_i(\mathbf{x},\omega_i)$, specifies the incident radiance arriving at the location $\mathbf{x}$ along the direction ω_i. Finally, the term $(\omega_i \cdot \mathbf{n})$ accounts for the reduction in the received energy because the surface may be oriented in a way other than perpendicular to the incident direction ω_i. As the surface becomes more angled, the incident flux is distributed across a larger surface area. The surface orientation is marked using the surface normal $\mathbf{n}$ which is perpendicular to the surface at $\mathbf{x}$. It is generally assumed that directions ω_i, ω_o and $\mathbf{n}$ are normalized. Note that the presented formulation ignores any dependency on time, light wavelength, or occlusion of light.

Efficient solutions of the radiance equation are still an active area of research. One of the issues is that the radiance term L appears within the integrand, requiring iterative solutions which propagate energy throughout the scene. Before introducing a family of solutions of the rendering equation, we must describe how scene data is represented digitally.

2.1.2 Describing Virtual Scenes

The entire rendering process is summarized in Figure 2.3. It takes the description of the virtual scene and produces an image depicting that scene. The *scene* input typically describes the virtual objects, their material properties, lights used to illuminate the scene, and camera(s) used to capture the scene. Scenes can also be dynamic where various parameters change over time. Once the visible objects are found, their appearance is simulated, producing colors that are accumulated into the image. Depending on the rendering algorithm, object appearance can account for light inter-reflections, thus making

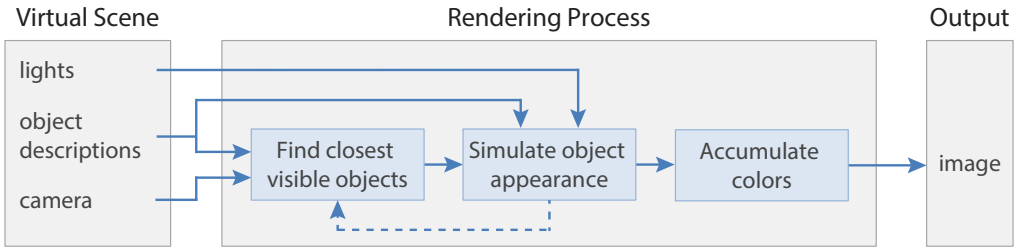
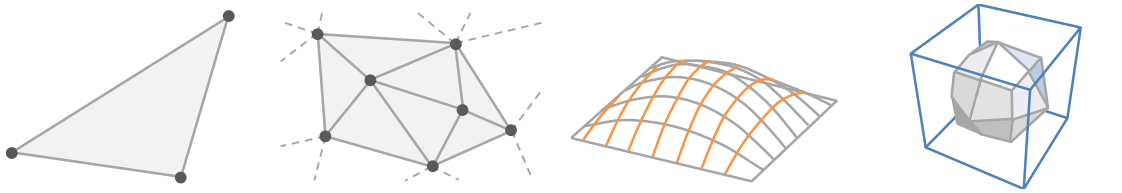


Figure 2.3: A high-level overview of the rendering process, which takes the description of the virtual scene (left) and produces an image depicting that scene (right).

computation iterative: another search for the closest visible objects and simulation of their appearance. Section 2.1.3 introduces the ray casting algorithm used to derive a family of popular rendering algorithms.

One of the properties virtual scene descriptions specify is the physical extent occupied by each object. Based on how light interacts with an object, it is convenient to classify its description as a *surface* or a *volume*.

Surfaces are encountered most commonly in daily life and often appear as solid with clear boundaries or edges. Surfaces can be described as two-dimensional interfaces between media, typically air and a material making up the object. One can choose a particular mathematical representation of the interface based on the desired properties of the representation. Some of the common surface representations are shown in Figure 2.4. An individual element representing an object or part thereof is called a *primitive*. A common planar primitive is a *triangle*, Figure 2.4a. A collection of triangles, referred to as a *mesh*, forms a piecewise linear representation of a complex surface, Figure 2.4b. A triangle mesh does not preserve continuity or smoothness in surface derivatives, therefore making it difficult to faithfully recreate curved objects. Applications like computer-aided design (CAD) that require continuity of derivatives define shapes using high-order patch primitives. Each patch is constructed as a tensor product of high-order polynomials interpolating a set of control points. Nonuniform rational basis spline (*NURB*) surfaces, Figure 2.4c, can be defined using high order Bézier curves [36]. *Subdivision surfaces*, Figure 2.4d, are defined by a recursive application of a surface refinement strategy, which, at each application, subdivides a polygonal input mesh to create new polygons. Although one can directly use the limit surface created by an infinite number of applications of the subdivision rule [22, 122], implementations commonly subdivide to a maximum level using triangles



(a) Triangle **(b)** Triangle Mesh **(c)** NURBs Patch **(d)** Subdivision Surface

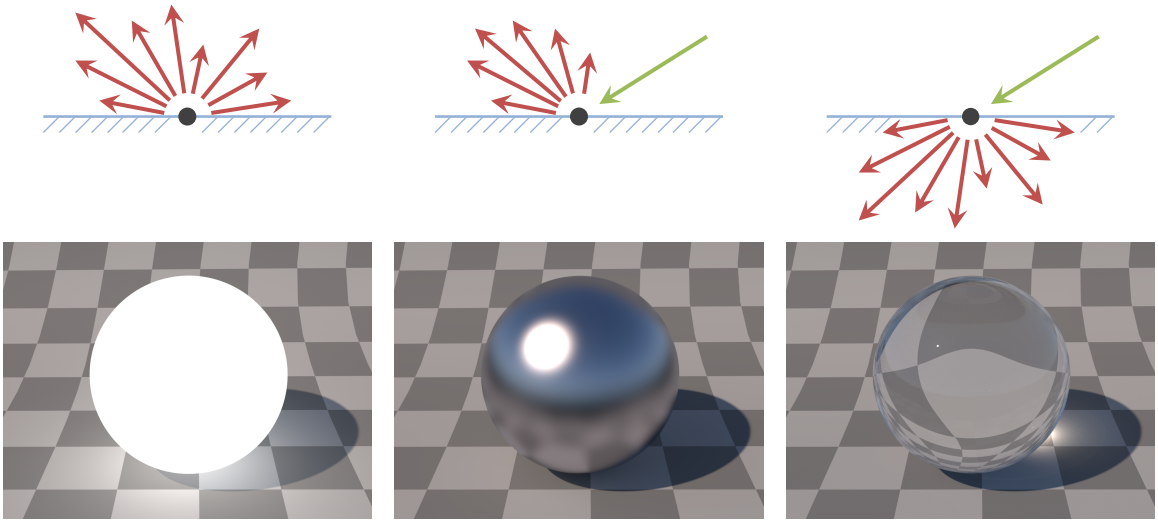
Figure 2.4: A few common surface representations. The subdivision surface (d) shows the result after a single subdivision of the blue cube. The limit surface is a sphere.

and treat the resulting surface as a mesh [11]. The collection of all surface representations in a scene is referred to as scene *geometry*.

Volumes, on the other hand, describe objects whose appearance is driven by light bouncing within the object extent, like clouds or fog. Modeling volumes requires storing a notion of material density and other physical properties throughout the volume extent. Since this work focuses on surfaces, volumes are not discussed further. We refer interested readers to Pharr et al. [105] for details.

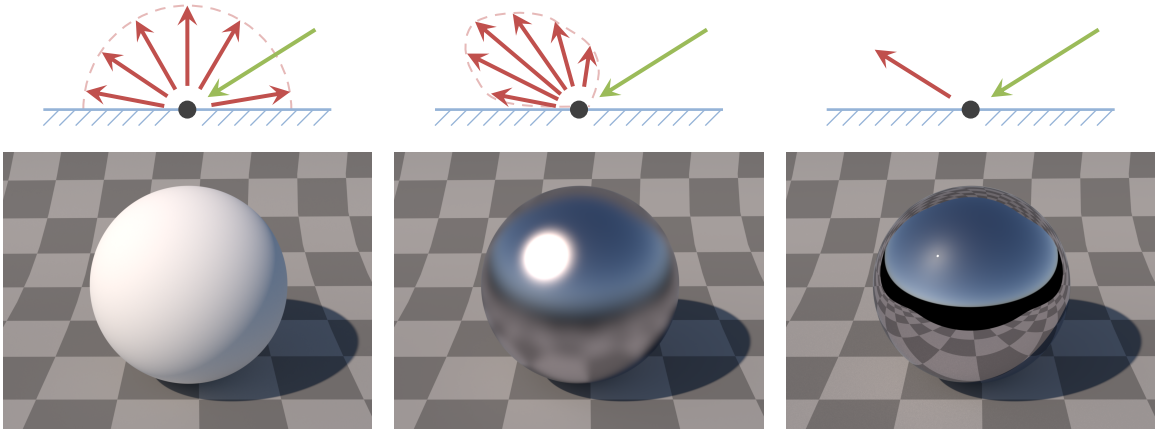
A *material* simulates the appearance of an object by modeling the physical behavior of light when it encounters the object based on what it is made of, like wood or glass. Material models are typically derived from first principles. Figure 2.5 shows some possibilities for how light interacts with a surface. A surface can *emit* light (Figure 2.5a), giving the surface a glowing appearance – a cell phone screen for example. Surfaces can also *reflect* light (Figure 2.5b) thus redirecting some of the incident energy away from the surface – a polished metallic ball bearing for example. Surfaces can also *transmit* light (Figure 2.5c) thus redirecting some of the incident energy through the surface interface – water for example. Each of these interactions can be classified further based on their directional concentration, Figure 2.6. *Diffuse* interactions spread the energy equally in all directions (Figure 2.6a), *specular* interactions concentrate all energy in one direction (Figure 2.6c), while *glossy* interactions lie somewhere in between (Figure 2.6b). Complex appearance can be generated by combining several material models together as layers [59]. This work uses only diffuse reflective materials because they produce a good test case of the proposed algorithms and architectures. Please see Pharr et al. [105] for a more in depth discussion of material models and their derivations.

In addition to the object descriptions, the rendering process also needs a description of



(a) Emission **(b)** Reflection **(c)** Transmission

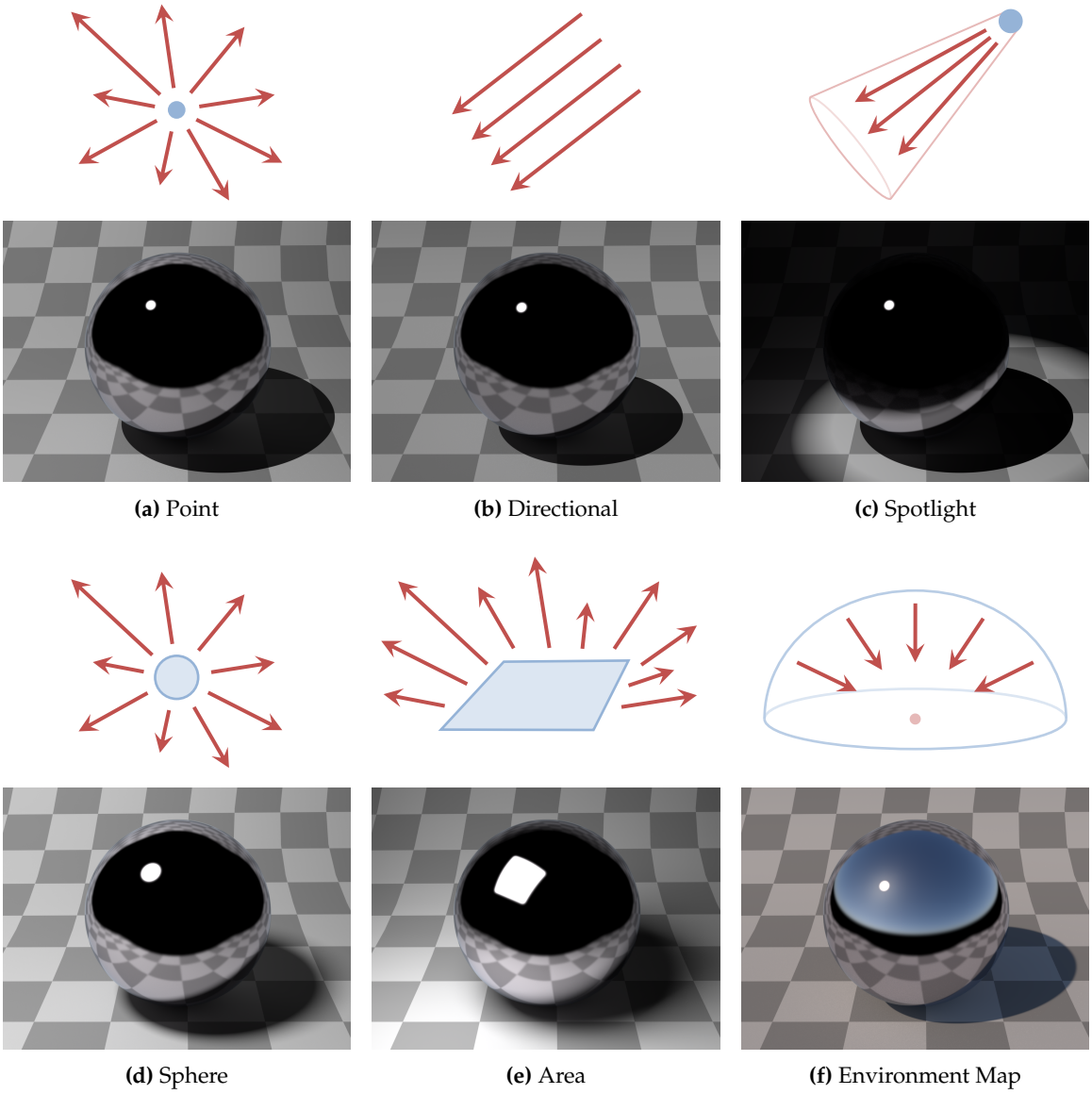
Figure 2.5: Possible ways light can interact with a surface. The top row shows a diagram of the interaction, while the bottom row shows an example of the resulting appearance.



(a) Diffuse Reflection **(b)** Glossy Reflection **(c)** Specular Reflection

Figure 2.6: Directional concentration classification for reflections. Emission and transmission are classified similarly. The top row shows a diagram of the interaction, while the bottom row shows an example of the resulting appearance.

the *light sources* that illuminate the scene. Common luminaires include the sun, the sky, and various light bulbs. Much like the objects in the scene, describing the lights requires both their physical representation (position, surface, etc.) and the illumination profile which models the positional and directional distributions of the emitted light. Figure 2.7 shows examples of common lights. Although not physically possible, a *point light*, Figure 2.7a, is the simplest: it has no surface and emits all light radially, typically distributed evenly. It



(a) Point **(b)** Directional **(c)** Spotlight

(d) Sphere **(e)** Area **(f)** Environment Map

Figure 2.7: Common light models used in rendering. Diagrams at the top of each subfigure depict the light source with arrows indicating a possible distribution of the emitted energy. Images at the bottom show an example of the resulting appearance when illuminating a simple scene.

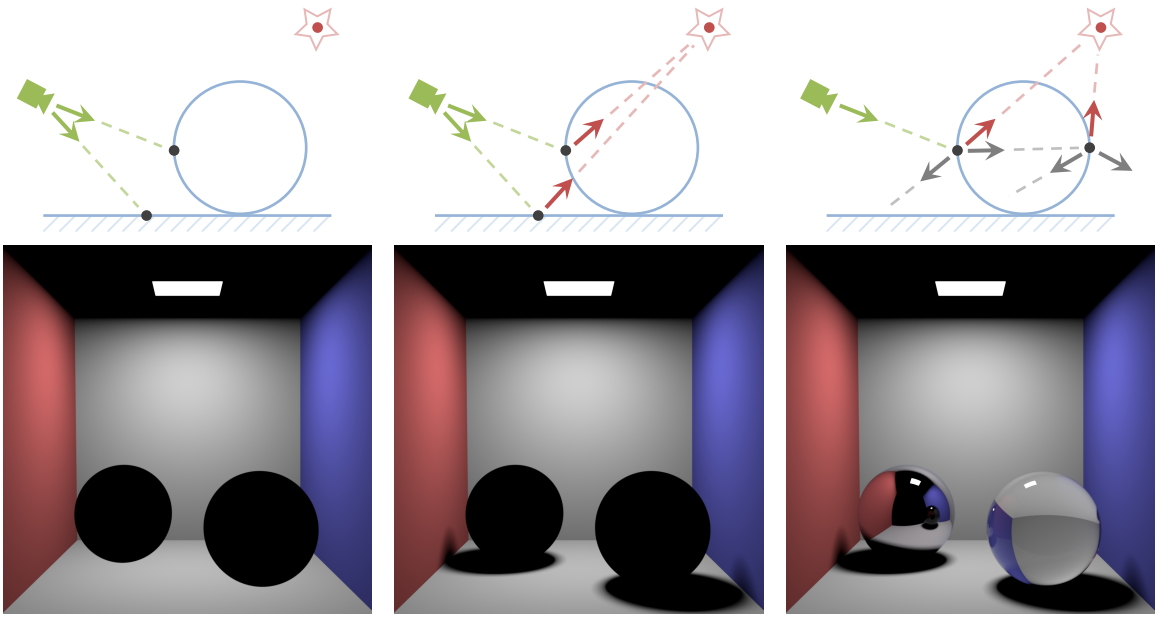
is specified by a position and an intensity. *Directional light sources*, Figure 2.7b, are another extreme. They simulate light that is substantially far away so that all radiance travels along a specific direction regardless of the spatial position throughout the scene. A *spotlight*, Figure 2.7c, mimics lights with a lampshade, where all illumination travels within a given angle away from a primary direction. Although a conical light is shown as an example, the cross-sectional shape of the light can be arbitrary. Spotlights can also define a radial falloff

function to smooth out the edge of the shadow. Most of the physical luminaires have an emissive surface. Such light sources are called *area lights*. One of the simplest is the *spherical light*, Figure 2.7d, which is defined by a position and a radius. Its simplicity stems from the fact that a sphere projects onto a circle from all directions, making it easy to use during rendering. Area lights can also have arbitrary shapes with the surface defined by geometric primitives. Figure 2.7e shows a simple rectangular area light. Finally, an *environment map* light defines the illumination arriving from far away from a (hemi)sphere of directions. One type of the environment map is a 360 degree image captured at a real-world location. Other environment maps use analytic models to imitate sky illumination (the sun can be modeled separately) and can incorporate parameters like the time of day, latitude, etc. This work relies only on point lights because of their simplicity. Please see Pharr et al. [105] for a more in depth discussion on light models.

Finally, the rendering process requires the details of the *camera* used to capture what the scene looks like. Camera models mimic the real-world equivalents, which can be parameterized using position, orientation, aperture, shutter speed, etc. Depending on the use-case, camera models can range from a simple pinhole to a fully realized optical system. Although this work is somewhat agnostic to the camera model, we rely on the pinhole camera model because of its simplicity. Please see Pharr et al. [105] for more details.

2.1.3 Ray Casting

We introduce a single family of rendering algorithms that relies on geometric optics to solve the rendering equation by iteratively computing how light bounces within a scene. Simulation using geometric optics ignores wave-like behavior of light including diffraction, interference, and polarization. Light propagation is modeled by infinitesimally thin semi-infinite geometric *rays* that represent the path taken by light. Each ray is represented by an origin (a point in 3D) and a direction (a vector in 3D). Modeling a bounce of light requires searching the scene for the object that is closest to the ray origin along the ray's direction. We refer to the position of the intersection between a ray and a primitive as the *hit point*, which can also contain a description of the surface material properties. We refer to the process of finding the closest primitive representing a surface as *ray casting* because it casts a ray into the scene to find a hit point. A simple image of the scene can cast a single



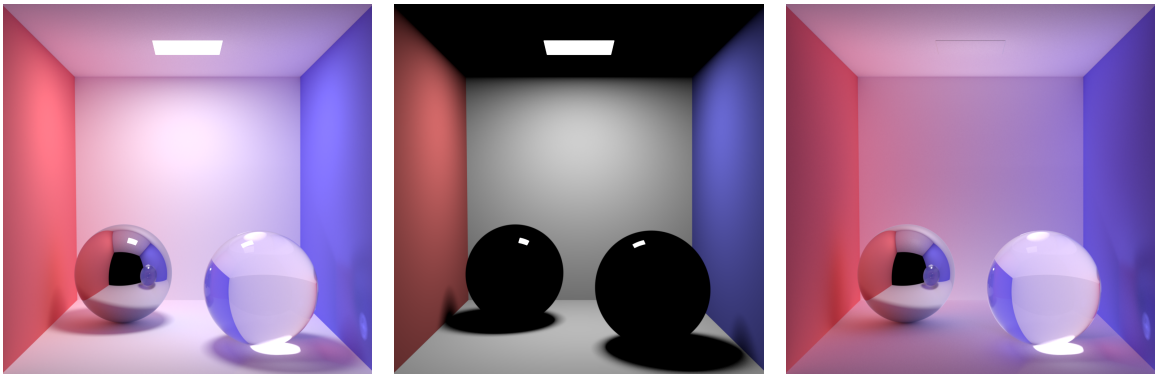
(a) Ray casting **(b)** Adds shadows **(c)** Adds reflections and refractions

Figure 2.8: Casting rays to generate different images of a simple virtual scene. The top row illustrates types of rays that are cast, and the bottom row shows the resulting image.

ray for every pixel to show the surface appearance, Figure 2.8a.

Such images lack important visual details like shadows or reflections and thus do not look realistic. One can simulate these effects by casting more rays, each generated at the hit points derived from the previous ray casting step. To simulate shadows, one can simply cast a *shadow ray* towards the light source. If the shadow ray hits any surface before reaching the light, then the hit point is considered in shadow and receives no contribution from the particular light. Figure 2.8b shows the result of casting shadow rays. Reflections and refractions rely on additional rays to query how much light would arrive at a hit point from the direction of the reflection or refraction. Computing this amount of light requires yet additional rays to be cast, Figure 2.8c.

What makes ray casting so powerful is the ability to derive a variety of sophisticated image generation methods from it. One can keep iterating between casting rays to compute the closest hit points and using the material properties of the hit surfaces to decide which set of rays to cast next. Each iteration models a single light bounce. The derived image generation algorithms can be separated based on when and how the secondary rays are generated and how their results are used.



(a) Global Illumination **(b)** Direct Illumination **(c)** Indirect Illumination

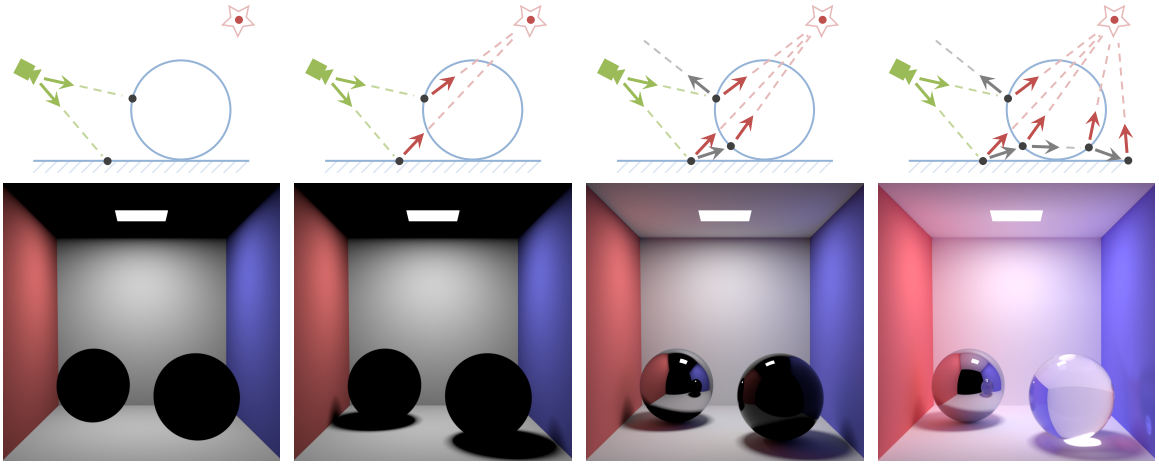
Figure 2.9: Global illumination (a) fully simulates light propagation in a simple scene. It can be separated into direct illumination (b), where all lights contribute illumination directly without interacting with any other objects in the scene, and indirect illumination (c), where light arriving at any surface must interact with other objects first.

2.1.4 Path Tracing

Introduced by Kajiya in 1986, the *path tracing* algorithm [64] can faithfully simulate light propagation and produce images indistinguishable from reality. The basic implementation can handle incredibly complex scenes and is fairly straightforward.

At any particular hit point, it helps to separate light into two components: *direct* and *indirect illumination*. The direct illumination component, shown from the point of view of the camera in Figure 2.9b, only considers the light that arrives onto the surface from all light sources directly, without interacting with any other surfaces. This includes shadows, but not reflections or refractions. The indirect illumination component, shown from the point of view of the camera in Figure 2.9c, considers all other light paths which include at least one bounce in the scene immediately after leaving the light source. Notice that the ceiling painted white receives red light reflected from a wall nearby, which acts as an indirect source of light. As a result, the white ceiling appears tinted red closer to the red wall and appears tinted blue closer to the blue wall. Color bleeding is one example of indirect illumination.

The combination of both direct and indirect illumination components computes *global illumination*, shown in Figure 2.9a. Global illumination encompasses simulating all possible interactions light can have between all surfaces over great distances. One example of global illumination is a room illuminated by an exterior light source through a slightly



(a) Primary Rays **(b)** Shadow Rays **(c)** 1 bounce GI **(d)** 10 bounce GI

Figure 2.10: Using a path tracing algorithm to simulate global illumination (GI). The top row illustrates the behavior of light and the bottom row illustrates the effect on a sample scene. Note that n^{th} bounce produces light paths that are at most $n + 1$ in length.

open door. An important strength of the path tracing algorithm is that it can simulate global illumination by solving the rendering equation.

Although not difficult conceptually, simulating global illumination requires an extensive amount of computation. In essence, one needs to simulate a subset of all *light paths* that connect the camera to all lights in the scene. Each light path represents several bounces a single light particle has taken between objects in the scene before arriving at the camera. Each light path is built from rays that connect hit points throughout a scene. In fact, the true number of paths and bounces is unbounded. The path tracing algorithm samples this set of possibilities by iteratively casting rays into the scene starting from the camera. Figure 2.10 illustrates the process. First, based on the camera, the algorithm generates a *primary ray* through each pixel in the image. Then the algorithm casts each primary ray into the scene to find which objects are the closest (Figure 2.10a). All of these objects are visible to the camera and appear in the image. For every hit point, the algorithm computes how much direct light is reflected from the hit surface by tracing shadow rays towards lights in the scene and applying the material properties of the hit surface (Figure 2.10b). To simulate the indirect illumination arriving at the hit point, the algorithm picks a random direction where the light could arrive from, casts another (*secondary*) ray in that direction, and shades the corresponding hit point (Figure 2.10c). Repeating this process iteratively

computes light paths of increasing length that connect the camera to the light sources in the scene. The algorithm stops iterating once the maximum light path length is reached.[2] Figure 2.10d shows the final image after ten iterations (light bounces). The described algorithm is called a Kajiya-style path tracer [27, 64], pseudo-code for which is shown in Algorithm 2.1.

Please note that reproducing the exact light paths would require simulating an infinite number of them. Path tracing relies on random sampling techniques to generate only a subset of all possible paths [27]. As more ray paths are traced, the image slowly converges to the ground truth. An image indistinguishable from the ground truth image requires many rays and a lot of computation time. Thus, this description is meant only for a high-level understanding of the process, and omits optimizations, sampling, and integration techniques. We refer the interested readers to Pharr et al. [105] for details on this and other sophisticated rendering algorithms.

2.1.5 Ray Traversal and Intersection

This work focuses on accelerating the ray intersection phase of the path tracing algorithm, referred to by the function call `scene.Intersect(ray);` on lines 7 and 14 in Algorithm 2.1. Given a ray, this phase searches through the scene primitives to find the closest one to the ray origin by computing ray-primitive intersections.

The simplest way to test whether any object intersects with a ray is by iterating over all primitives making up all of the objects in the scene. However, scenes used in film production contain billions of triangles, making any algorithm that tests all of the primitives for every ray prohibitively expensive. A film quality image at a 4K resolution (3840×2160 pixels) could require up to $2 \cdot 10^{19}$ intersection tests, which is astronomical. As a result, scene geometry is placed into an *acceleration structure*, which is an additional data structure designed to significantly reduce the number of ray-primitive intersection tests and thus the computational cost of the intersection phase. The process of searching through the acceleration structure is referred to as *ray traversal*.

[2] If the maximum path length is set too low, rendering can miss important inter-reflections between specular objects and produce an incorrect image. The refractive sphere in Figure 2.10c appears black because the paths are not long enough to allow for the second refraction on the other side of the sphere. An alternative technique, called Russian Roulette [7], can be used to statistically stop iterating based on the probability that the illumination simulated by a ray is absorbed at a particular hit point.

```
foreach pixel in image do
    // generate a primary ray
    rayDepth = 0;
    color = black;
    thruput = white;
    ray = camera.GenRay(pixel);
    // loop until ray reaches max path length
    for rayDepth < max_depth do
        // find closest visible object for ray.  Figure 2.10a
        hitInfo = scene.Intersect(ray);
        // ray missed everything
        if not hitInfo.didHit then
            color = color + thruput * background_color;
            break;
        end
        // ray hit an object - compute direct illumination by casting
           shadow rays.  Figure 2.10b
        foreach light in scene do
            shadowRay = light.GenShadowRay(hitInfo.pos);
            shadowHitInfo = scene.Intersect(shadowRay);
            // shadow ray missed - hit point is illuminated.  Apply
               material properties
            if not shadowHitInfo.didHit then
                brdf = hitInfo.EvalMatl(hitInfo.pos, ray.dir, shadowRay.dir);
                color = color + thruput * light.color * brdf * cos;
            end
        end
        // compute indirect illumination by casting ray for next bounce.
           Figure 2.10c
        oldRayDir = ray.dir;
        ray = hitInfo.GenBounceRay();
        rayDepth = rayDepth + 1;
        thruput = thruput * hitInfo.EvalMatl(hitInfo.pos, oldRayDir, ray.dir);
    end
    // write output pixel color
    image.SetPixelColor(pixel, color);
end
```

Algorithm 2.1: Pseudocode for the Kajiya-style path tracing algorithm.

The path tracing algorithm depends on two types of ray traversal. The *closest-hit* traversal searches for the primitive in the scene that is the closest to the ray origin along the ray direction. The closest-hit traversal keeps only the closest ray-primitive intersection hit point from many that may be found and sorted. *Any-hit* traversal can terminate once a hit is found, avoiding unnecessary intersection tests. Any-hit traversal applies to *visibility queries* which test whether two points in the scene are visible to each other with no primitive occluding one from the other. For example, shadow rays use any-hit traversal to test if any primitive blocks illumination onto the hit point that generated the shadow ray. The same acceleration structure can accommodate both types of traversal. When the acceleration structure deems no further hits need to be found, the traversal can be stopped resulting in *early ray termination*.

2.1.6 Acceleration Structures

Acceleration structures significantly reduce the number of intersection tests during rendering because they help avoid testing primitives that are guaranteed to miss the ray. Traversing any acceleration structure comes at an additional cost because it can require intersecting the ray against other special primitives. Therefore, to be beneficial computationally, the acceleration structure design must balance the savings from reducing the number of ray-primitive tests with the additional costs incurred by traversal.

All acceleration structures can be designated based on what they partition: space or objects.

2.1.6.1 Space Partitioning

Space partitioning schemes segment the volume the scene occupies into nonoverlapping regions. Primitives are then sorted and filtered into corresponding individual regions of the acceleration structure. A ray then marches from region to region and tests only the primitives referenced within visited regions. If there are no primitives within a region, the ray simply continues onto the next one, potentially skipping large empty volumes. The traversal can stop once a hit is found, as long as it is contained within the specific region. The traversal can also stop early if the distance along the ray to the hit point is closer than the bounds of the next region to be visited.

A downside of all space partitioning schemes is that primitives may span several re-

(a) Grid **(b)** Octree **(c)** Kd-Tree **(d)** BVH **(e)** BIH

Figure 2.11: Two-dimensional representations of common acceleration structures. *BVH* stands for bounding volume hierarchy, and *BIH* stands for bounding interval hierarchy. Different colors represent different levels in the hierarchical structures: black is top, blue is one level down, and red is two levels down.

gions. As a result, the primitives need to be either split at region boundaries, generating new primitives [33], or the primitives need to be referenced multiple times within the acceleration structure, once for every region that contains the primitive. Both of these solutions result in higher memory use. Another problem with duplicate primitive references is that a ray may intersect with the same primitive multiple times, which can be mitigated by using a mailbox [73]. For each multiply referenced primitive, it stores a reference to the primitive and an index for the ray that intersected it last.

Because space partitioning schemes not based on trees may not adapt well to variations in primitive density, a subset of their regions may reference a significantly larger number of primitives than other regions, making that subset much more expensive to test against. As a result, ray traversal and intersection performance is not as efficient as it could be.

Grid A uniform grid [26] simply subdivides the scene bounding box into a regular grid of volumetric cells, called *voxels*. Figure 2.11a shows an example of a grid acceleration structure. Each voxel contains a list of all primitives it overlaps with. A ray traverses through this structure using discrete differential analyzer, originally developed to draw lines on pixel displays [17]. The method relies on small constant increments to step from one voxel directly into the next one. Grids can be nested recursively to achieve better performance than regular grids [63], but the maximum number of nested subdivisions is typically set before the structure is built. For a complete performance analysis, see Ize et al. [56]. Havran briefly describes two other types of grids, hierarchy of uniform grids and adaptive grids [49].

Octree An octree [41] is a tree-based space partitioning scheme, where each level can be split into eight subregions of equal size along the center of the level's region,

Figure 2.11b. Each region is stored as a node in the tree. Octrees can adapt to nonuniform distributions of primitives throughout the scene much better than uniform grids can. Octrees can be considered a special case of a recursive grid. Traversing this data structure requires maintaining a stack of nodes that a ray will visit next.

Recent work embeds two planes, called contours, into each octree node [78, 79]. The contours approximate and constrain the surface within the node. Contours can be used as a proxy surface geometry in some cases and intersected directly, improving ray casting efficiency without a loss in visual fidelity.

Kd-Tree One of the most popular acceleration structures used in ray tracing is a kd-tree [12, 49, 124]. Each node in the tree is split into two by a single axis-aligned plane, Figure 2.11c. Allowing the tree to become deep enough, this acceleration structure can adapt to nonuniform distributions of primitives. Kd-trees were popular in the early 2000s because each node is compact to store and bounds are tight, thus leading to fast traversal. However, building kd-trees has proven to be not as efficient as more modern acceleration structures. Kd-trees can be considered a special case of a binary space partitioning tree [98] which also splits space into two parts per node but using an arbitrarily oriented plane.

Kd-trees are typically built in the top-down fashion using a heuristic to approximate ray traversal performance [54, 108, 134]. At each step, all primitives are sorted and binned along each axis, and then one bin boundary is selected as the location for the splitting plane based on the greedy optimization of the heuristic cost. The splitting plane could also be used to carve out empty space producing a node with no primitives, which can considerably reduce unnecessary traversal and intersection tests [49].

Traversing a kd-tree involves marching the ray through each node's children visiting the one closest to the ray origin first. If a hit is found within a node, it is guaranteed to be the closest and traversal can stop because nodes do not overlap. Traversing from one node to another revisits nodes higher up the tree, which requires for each ray to maintain a traversal stack of nodes to be visited. The overhead of maintaining the traversal stack can be reduced by linking nodes to their siblings with ropes [50]. The traversal stack requires memory for storage, which may prohibit tracing many rays in parallel. It is possible to avoid storing the stack explicitly, which is well suited for architectures that do not provide much memory per thread like modern graphics processors [109].

Because of the build times, kd-trees are no longer the popular acceleration structure for ray tracing. They are still used for rendering algorithms that rely on the nearest-neighbor search queries for points [47, 61, 62].

2.1.6.2 Object Partitioning

Object partitioning schemes build a tree separating primitives into groups rather than the volume they occupy. Each node in the tree bounds the volume occupied by its children nodes, and typically each primitive is referenced once. These schemes easily adapt to variations in primitive density.

Bounding Volume Hierarchy A bounding volume hierarchy (BVH) [41, 65, 133] bounds all primitives within the scene by an n-wide tree of nodes, Figure 2.11d. Each node in the tree stores a *bounding volume* which encloses the volumetric extent of all of its children and thus primitives within. Although trees are typically binary, several levels of the tree can be combined together into a single n-wide node. Wider trees can utilize the vector processing hardware architectures more efficiently.

During traversal, if the ray misses the bounding volume of a node, there is no need to test that ray against any of the node's children, avoiding unnecessary primitive intersection tests. For efficiency, intersecting the bounding volume representation must be much cheaper than intersecting a small number of primitives. Axis-aligned bounding boxes (AABBs) are a common choice, but other bounding volumes like spheres or polyhedra have been considered in the past [65]. The intersection test between a ray and an AABB requires six ray-plane intersections[3] and a few comparisons [139]. AABBs work best for axis-aligned primitives and fail to create tight bounds when primitives are oriented off axis – a rectangle oriented at a 45 degree angle for example. Many rays that intersect with the large bounding box will likely miss the rotated rectangle inside, leading to unnecessary node visits. One benefit of BVHs is that primitives stored within are referenced exactly once, unless the BVH derives tighter bounds by using split planes, duplicating some primitive references [123].

Popular BVH builders work in a top-down fashion relying upon the similar heuristic

[3]Each dimension requires two plane intersections: one for the minimum and another for the maximum bound of the AABB. Each intersection between a ray and an axis-aligned plane can be expressed as a single one-dimensional fused multiply-add operation.

as the top-down builders for kd-trees. Once the partitioning plane is selected, primitives are separated into two sets based on which side of the axis-aligned plane each primitive falls into. Then the two child nodes are created, one per partition, each with the bounding volume encompassing the primitives within. The sibling bounding volumes can overlap and the child volumes do not necessarily fill the parent's volume, leaving some empty space carved out.

Approaches that build the tree from the bottom up can produce higher quality trees. Each primitive is first bounded directly, forming a single cluster. Recursively, pairs of nearest clusters are combined together into individual clusters until the final single cluster is created [138]. Each cluster then forms a node in the BVH. Agglomerative build approaches require nearest-neighbor searches, which have high computational overhead. As a result, more recent approaches approximate the nearest neighbors [31, 46, 91].

Tree traversal starts from the top node. Each node's bounding volume is tested against a ray, and if intersected, the ray continues to traverse the children nodes. A common optimization is to traverse the node closest to the ray origin first. Also ray traversal can stop if a hit is found closer than a node's bounding volume. However, because bounding volumes can overlap, a ray may have to visit the node's sibling even if a hit was already found.

BVH trees with AABBs are such a popular acceleration structure that they are commonly used by the major film rendering software, like Arnold [40], RenderMan [25] or Hyperion [20], as well as the real-time ray tracing APIs, like OptiX [102] or Embree [137, 141]. As a result of this popularity, there are many variations of the BVH acceleration structure. Please see Section 3.1.2 for recent work on BVHs.

Bounding Interval Hierarchy A bounding interval hierarchy (BIH) [131] is a binary tree where each node partitions space using two rather than six axis-aligned planes, Figure 2.11e. Both of the planes lie along the same axis and form two bounding intervals in one dimension. One interval bounds the left child node, where the plane stores the maximum bounds of the child node and the parent node provides the minimum bound. The other interval bounds the right child node, where the plane stores the minimum bound of the child node and the parent node provides the maximum bound. When the bounding intervals overlap, they represent overlapping bounding volumes of children

nodes. Nonoverlapping bounding intervals represent empty space in between children nodes along a given axis.

The acceleration structure traversal exhibits behaviors similar to both a BVH and a kd-tree. Determining which child node is closer to the ray origin, making early ray termination effective, relies on the axis encoded in each node directly. When the sign of the ray direction along the appropriate axis is positive, the left child is closer to the ray origin and should be traversed first. The negative sign of the ray direction signifies that the right child is closer. While kd-trees are traversed this way, enabling similar behavior in the BVH requires additional storage per node. Similar to a BVH, one may need to traverse both child nodes of the BIH after a hit is found because child node intervals can overlap.

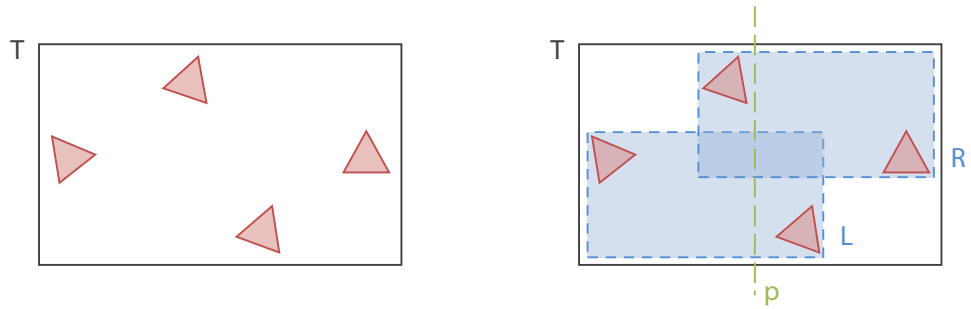
BIHs are not used in the rendering software because they suffer from issues similar to kd-trees. The acceleration structure requires deep trees to adapt to the scene data. Many rays traversed in parallel can diverge, loading data from different acceleration structure nodes scattered throughout memory, which strains the memory system. As discussed later, certain types of memory accesses are costly and effecient memory use is paramount for performance.

2.1.6.3 Estimating Acceleration Structure Quality

Acceleration structures can significantly reduce the number of primitives each ray must intersect during traversal. Ideally, the acceleration structure partitions all primitives optimally to minimize the number of intersection tests per ray on average. In practice, we must rely on quality heuristics to generate acceleration structures that reduce computation time spent on traversal and intersection.

One popular heuristic used to estimate quality of each acceleration structure is the surface area heuristic (SAH) [43, 49, 88]. SAH combines the probability that a ray hits a particular node of the acceleration structure with the costs to intersect against primitives inside. SAH assumes that a ray originates outside the scene volume and travels along a random direction without stopping anywhere within the scene volume. Lower SAH cost values are considered better because they should lead to lower time spent on traversal and intersection during rendering.

Given a volume bounding the primitives in a scene, we can create an acceleration



(a) Leaf node configuration **(b)** Internal node configuration

Figure 2.12: Using the SAH cost to decide whether to make a node T into (a) a leaf or into (b) an internal node with two child sub-nodes L and R. The node contains four triangles, shown in red. The black line outlines the AABB of the node T. A dashed line on the right represents one possible splitting plane, p.

structure comprised of *internal* and *leaf nodes*. Internal nodes reference other nodes as children, and leaf nodes contain references to primitives. The quality of an entire acceleration structure can be approximated using the following cost metric, which is a sum of traversal and intersection costs weighted by the probability that a ray can enter a particular node:

$$C_T = \frac{1}{\mathsf{SA}(scene)} \left[C_{TI} \sum_{i=1}^{N_I} \mathsf{SA}(i) + C_{TL} \sum_{l=1}^{N_L} \mathsf{SA}(l) + C_I \sum_{l=1}^{N_L} \mathsf{SA}(l)\mathsf{N}(l) \right], \tag{2.2}$$

where $\mathsf{SA}(n)$ computes the surface area of a node n, i is i^{th} interior node and l is l^{th} leaf node. Counters N_I and N_L count the total number of interior and leaf nodes respectively. The costs C_{TI} and C_{TL} correspond to the run-time costs to traverse an internal and a leaf node respectively. The cost C_I corresponds to the cost to intersect against a primitive, like a triangle. Finally, the number of primitives in a leaf node l is given by $\mathsf{N}(l)$.

The builder of tree-based acceleration structures can rely on the SAH metric to estimate whether it is best to make a node an internal node, splitting it into sub-nodes, or to make it into a leaf node. Illustrated in Figure 2.12, the cost of each decision is evaluated via the following formulation:

$$C_{leaf}(T) = C_I \mathsf{N}(T) \tag{2.3}$$

$$C_{interior}(T, p) = C_{TI} + \frac{C_I}{\mathsf{SA}(T)} \left[\mathsf{SA}(L)\mathsf{N}(L) + \mathsf{SA}(R)\mathsf{N}(R) \right], \tag{2.4}$$

where a plane p splits the current interior node T into a left L and a right R child sub-nodes. The costs C_I and C_{TI} correspond to the the cost of intersecting a primitive and traversing a node respectively. The surface area of a given node n is evaluated by $SA(n)$. The counts

$N(n)$ correspond to the number of primitives in a node n. Because some primitives can be referenced in both child sub-nodes, note the following relationship between primitive counts per node: $N(T) \leq N(L) + N(R)$.

Unfortunately, because the assumptions SAH makes about rays are unrealistic, the SAH cost values do not perfectly correlate with the quality of an acceleration structure (how fast the ray traversal is) [2]. Not all ray types result in purely random distributions. Primary rays are highly correlated, but, after a few bounces, the secondary rays become more random. The SAH metric also assumes all ray origins are located outside of the scene bounding box, however, in practice, majority of rays start from within the scene.

2.2 Computer Architecture

This work relies on the fundamentals of computer architecture to design dedicated hardware architectures aimed to accelerate the path tracing algorithm. This section only introduces the concepts used throughout the thesis; please see Hennessy et al. [52] for more details. We discuss only the following two components of architecture design: computation and memory. We assume that the necessary data has been loaded into the machine ahead of the computation and ignore the input/output portion of computing systems.

There is a wide range of programmability in chips used for data processing. At the one extreme are processors designed for specific use case and nothing else, called *application specific integrated circuits* (ASICs). These processors encode the required processing functions directly in digital logic on chip and thus are extremely energy efficient and fast at their specific task. At the other extreme are fully programmable general-purpose processors, which can read and execute a sequence of provided operations of programmable architectures. They trade off flexibility for effeciency because flexibility requires certain architectural components that add energy and latency when compared to dedicated ASICs. Programmable architectures can also exploit dedicated hardware components to effeciently accelerate specific tasks that are common like video encoding and decoding for example. Microprocessors, central processing units (CPUs), and graphics processing units (GPUs) are examples. *Field programmable gate arrays* (FPGAs) fall in between the two extremes because they can emulate digital logic elements that an ASIC would implement in silicon directly but can be reconfigured for any specific application. The performance and energy

efficiency of FPGAs lies in between ASICs and general-purpose processors. Although programmable and efficient, configuring FPGAs is more difficult than programming general-purpose architectures.

Programmable processors execute a program, which is a stream of commands each represented as individual instructions. Instructions can include simple arithmetic, like addition, or can be specifically tailored for an application, like intersecting a given ray with a given triangle. The *instruction set architecture* (ISA) defines all the instructions that a particular machine can execute, and how to encode each instruction along with its inputs and outputs in a machine-readable (binary) format.

We refer to the smallest single processing unit as a *hardware thread* or a *processing core*. A machine with a single thread is called a *uniprocessor*, while machines with multiple threads are called *multi-core processors*. We refer to the physical realization a processor architecture as a single *chip*.

2.2.1 Types of Parallelism

Applications can enable concurrent processing through two types of parallelism. One type is data-level parallelism, where many data items can be processed at the same time. The other type is task-level parallelism, where work can be decomposed into parts that can operate independently of others and at the same time. At a coarse level, hardware can exploit these two types of parallelism in several ways [37, 52]. *Single instruction stream, single data stream* classifies a uniprocessor which executes a single sequential program on a single sequence of data. *Single instruction stream, multiple data streams* (SIMD) classifies architectures that execute the same instruction stream using several individual processor threads each operating on a different data stream. All threads in a SIMD architecture execute the same instruction at the same time. *Multiple instruction streams, multiple data streams* (MIMD) is the most general application of parallelism. MIMD architectures can be thought of as a collection of uniprocessors or threads, each executing its own program on its own data stream.

Recent architectures exploit parallelism by combining SIMD and MIMD processing. *Single instruction stream, multiple threads* (SIMT), common in recent GPUs, operates on a block of processing threads called a *tile*. All threads in a tile operate in a SIMD fashion,

however each tile within the GPU can execute its own program called a *kernel*. Moreover, GPUs can process more data, gaining efficiency, by dynamically switching which kernel of several is executed by each single tile. In a *single program, multiple data streams* (SPMD) type of architectures all threads execute the same program but on different data streams. Unlike SIMD, each thread can execute a different instruction within the single instruction stream. Note that, although similar to MIMD, SPMD executes only a single program.

2.2.2 On-Chip Computation

The architectures presented in this work rely on a reduced instruction set computer (RISC) instruction set [24, 103, 104]. RISC reduces the number of cycles spent per instruction by significantly simplifying the ISA. The key properties of the RISC ISA are three-fold. First, data operations work on data stored in registers and each operation changes the entire register. Secondly, the only memory operations allowed are load and store operations that fetch data between the memory system and thread registers. Finally, there are only a few binary formats for instructions and all instructions are of the same size, 32 bits for example. The instruction *latency* is the number of processor clock cycles it takes to execute an instruction to completion. The complexity of the digital logic implementation of the instruction functionality dictates this latency.

The average time it takes to execute an instruction can be reduced by using a *pipeline*, which is composed from several stages. The pipeline with n number of stages should reduce the time per instruction by a factor of n because each stage can execute simultaneously and with a separate instruction. The longest time between different stages is typically set to the processor clock cycle time so that each stage can complete execution within one clock cycle of the processor. Thus pipelines reduce the processor clock time compared to nonpipelined architectures. Note that an instruction with the latency of 30 cycles can be fully pipelined, thus computing a new result on every cycle.

One simple pipeline involves five stages, shown in Figure 2.13a. The first stage is the instruction fetch (IF) stage which fetches the current instruction from memory and advances the *program counter* which tracks which instruction in the program to execute next. The next stage is the instruction decode (ID) stage, which decodes the binary representation of the instruction and reads the inputs from the appropriate registers. The third

(a) Pipeline stages arranged in order. Ignores pipeline optimizations or connections to on-chip memory units.

Instruction Stream	Clock Cycle 1	2	3	4	5	6	7	8	9
instruction i	IF	ID	EX	MEM	WB				
instruction $i+1$		IF	ID	EX	MEM	WB			
instruction $i+2$			IF	ID	EX	MEM	WB		
instruction $i+3$				IF	ID	EX	MEM	WB	
instruction $i+4$					IF	ID	EX	MEM	WB

(b) Instructions flowing through the pipeline. A new instruction is fetched at every row (from top to bottom). Every instruction takes 5 cycles to complete execution (from left to right). Each column shows which stage of the pipeline is executing each instruction.

Figure 2.13: An example five-stage pipeline for a RISC processor. Stages are abbreviated by: *IF* is instruction fetch, *ID* is instruction decode, *EX* is execution, *MEM* is memory access, and *WB* is write-back.

stage is the execution (EX) stage, which performs the operation dictated by the instruction. Memory instructions compute the data address, while arithmetic instructions operate on the registers provided by the previous ID stage. The fourth stage is the memory access (MEM) stage, which either reads data from or writes data into the memory system. The final stage of the pipeline is the write-back (WB) stage, which writes the result of the instruction into the output register.

Figure 2.13b shows the concurrent execution of several instructions as they pass through the example five-stage pipeline. Each instruction is fetched at every sequential cycle. Assuming each pipeline stage takes a single cycle to execute for all instructions in the ISA, an instruction that enters the pipeline on cycle c completes its execution on cycle $c+5$. However, the effective instruction latency can be higher if the pipeline stalls between different stages. A *stall* pauses the execution of a stage because it cannot pass its data forward to the next stage in the pipeline. All instructions in the preceding stages would also incur an additional cycle of latency for every cycle the pipeline stalls. The simplest example of a pipeline stall is a stall in the IF stage, which delays the instruction being fetched by a cycle. A more complex example is a data stall where the next instruction

needs to use the output of the current instruction. As a result, the pipeline has to stall in the ID stage until the WB stage for the current instruction completes, so that the next instruction can read the data it needs. Data stalls can be addressed by connecting stage outputs to the inputs of the prior stages to avoid waiting for the WB stage to complete. Pipeline hazards like these and solutions to them are outside the scope of this introduction. Please see Hennessy et al. [52] for more details.

The example five-stage pipeline describes a simple *in-order* execution that sequentially executes each instruction in the stream. One can improve the number of instructions executed per cycle by executing instructions *out of order*. Out of order processors consider a small sliding window of instructions in the stream. Once an instruction within the window has all of its inputs available, it can be executed. Because memory access instructions can change the data currently stored in registers, they must be handled carefully. For example, the processing thread can avoid stalling the pipeline while waiting for a memory load instruction to return and instead execute arithmetic instructions that appear later in the program stream and that do not rely on data to be loaded. The improvement in performance comes at the cost of increased architectural complexity and used chip area. Because this thesis relies on an in-order pipelined SPMD architecture, we do not discuss the out-of-order processor architectures. Please see Hennessy et al. [52] for details.

2.2.3 Memory Subsystem

The memory subsystem of any architecture feeds the computational units with data so they can perform useful work. The thread pipeline sends data directly to computational units after it is fetched from a *register file*, which is a small collection of registers. Each register stores a small piece of data between 32 and 256 bits in size and provides access within a single cycle latency. Because the on-chip area is a precious resource, register files are limited to store only a few dozen of registers.[4] At the other extreme lies the main memory, which offers a large data storage located off chip. While it is significantly slower per access than the register file (100s of cycles), it is much faster than other sources of data like disk storage. The main memory is implemented using dynamic random access

[4]CPU and GPU architectures that can execute several kernels at once require a larger number of registers per thread to be able to save and restore the kernel state.

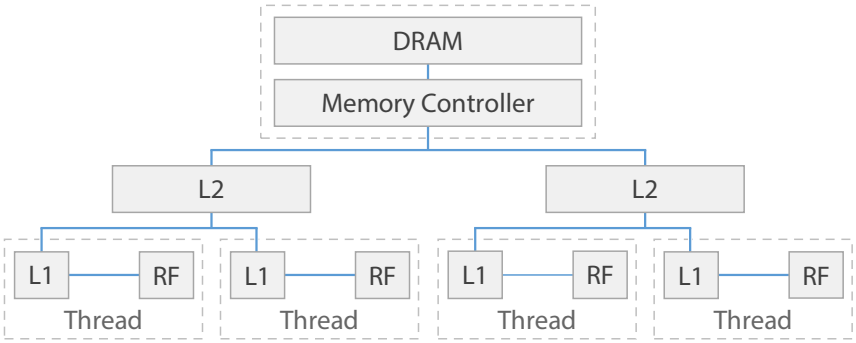


Figure 2.14: An example of a memory hierarchy with two levels of caches below the main memory. Blue lines indicate bidirectional data connections. *DRAM* implements the main memory located off-chip. We consider the *Memory Controller* a part of the DRAM system, although it is located on chip. *L2* and *L1* represent level-2 and level-1 caches. *RF* is the register file for every hardware *Thread* in the machine.

memory (DRAM) which is built from dedicated memory chips and is located near the processor chip. As a result, the rest of the memory system is designed to facilitate moving data between the large slow off-chip memory and small fast on-chip registers. Traditional memory systems are designed hierarchically where each level holds an increasing amount of data with higher access latencies and shares access between more computational cores, Figure 2.14. Data requests flow from each thread up the memory hierarchy until they reach the main memory.

2.2.3.1 Caches

Every level in the hierarchy between the register file and DRAM is called a *cache* because it stores a portion of recently accessed data. In a perfect world, main memory would provide a spacious data store that is accessible within a single cycle of latency. Unfortunately, data storage is either fast and small (on-chip memory) or a slow and large (DRAM). Thus on-chip caches which respond with much lower latency than DRAM can only store a small subset of the data stored in main memory. The limitation of storing only a subset of data results in architectural complexity: in both the design of caches and their operation as well as the entire memory hierarchy itself.

Caches are designed to fetch, store, and provide the requested data based on its memory address. When a data request is received, the cache first checks if the data is contained within. If contained in the cache, a *cache hit*, the piece of data is returned from the cache.

Otherwise, if not stored within the cache, a *cache miss*, the data is fetched from the cache or the main memory located at the next level in the hierarchy. Caches operate on blocks of data typically 64 bytes in size, called *cache lines*. Because the lowest level cache serves the register file, it can support word-sized accesses, typically 32 or 64 bits in size.

The efficiency of a cache is measured by its *hit rate*, computed as the percentage of all data requests that result in a cache hit. The higher the hit rate the faster data is returned because fewer requests reach a higher level cache which adds extra access latency. Small improvements in the cache hit rate can result in sizable reductions in latency per data access, which is linearly correlated with the *miss rate* computed as $(1 - \textit{hit rate})$. Increasing the hit rate from 98% to 99% reduces the miss rate from 2% to 1%, thus halving the effective access latency and potentially doubling the chip performance. As a result, caches are tuned to have the highest possible hit rates (lowest possible miss rates) across a variety of applications, the behavior of which is typically approximated using comprehensive software performance benchmarks like SPEC [53].

There are several approaches that improve memory system performance. The first approach increases the cache line size in the architecture. Larger cache lines take advantage of the spatial locality of data, where accessed data is located nearby in memory address space. An array of numbers exhibits high spatial locality, while a linked list of objects may not. However, a cache of a fixed size would contain fewer cache lines, which could lead to an increase in the *miss penalty*. The miss penalty is defined as the latency cost incurred upon a cache miss, including the costs at all levels of the hierarchy above the current cache level. Another approach to increase the hit rate increases the cache size to fit more data, thus increasing the probability that the requested data is located within. Unfortunately, this approach increases the access latency and power used by the cache because of how caches are built in silicon. The third and popular approach relies on a multi-level hierarchy of caches to reduce the miss penalty. Because of the relationship between capacity and speed, where the larger caches are slower, and the fastest cache not being large enough, architects can make use of a number of cache levels each with increasing capacity and hence lower performance. The lowest-level cache, called the level-1 (L1) cache, is the closest to the hardware threads and can provide data to the register file with a single cycle latency. However, the speed and the placement of L1 caches limits their size, 32KB is typical for

CPUs for example. Thus a slower level-2 (L2) cache with more capacity is used to capture all of the L1 cache misses. Without the L2 cache, the L1 cache misses would reach the main memory directly adding significant additional latency and increasing the miss penalty. Recent CPU architectures rely on three to four levels in the cache hierarchy below the main memory.

The cache miss rate can also be reduced by increasing the cache *associativity*, which dictates where the cache can internally store a cache line. A *direct-mapped cache* has no associativity and statically maps cache line addresses to fixed locations in its internal storage. One can use the least significant bits of the cache line address to compute the target location in the cache: $(cache\_line\_address \mod cache\_capacity)$, where $cache\_capacity$ is measured in the number of cache lines. Note that this function maps every cache line separated by a stride to the same internal storage location. Although very simple to implement in hardware, the performance of direct-mapped caches suffers when the data requests can alternate between two addresses spaced apart perfectly. Every data request would result in a cache miss because it replaces data stored previously, incurring unnecessary access latency. A *fully associative cache* can place each requested cache line anywhere within, thus such caches do not suffer from the alternating data access pattern. After the first two cache misses that pull both addresses into the cache, both pieces of data would be available immediately. Such caches require more hardware resources to implement. A *set-associative cache* allows a fixed number of locations where a requested cache line can be stored. A *set* is a group of cache lines internal to the cache that acts in a fully associative manner, so that a requested cache line that maps to a specific set can be placed anywhere within that set. A cache with n cache lines in each set is called *n-way set-associative*. It applies the following mapping from a cache line address to a set: $(cache\_line\_address \mod number\_of\_sets)$. After calculating the set where the cache line maps to, data requests check if the set contains the requested address. Keeping the on-chip area the same as a direct-mapped cache, the n-way set-associative cache has an addressable size that is reduced by a factor of n because each set can store n cache lines. Typical values for the set size are 2, 4, and 8, selected based on the tuning of the memory system for each specific architecture.

Associative caches can rely on a cache line *replacement policy* to improve the hit rates

further. The replacement policy dictates which cache line in a set is evicted from the cache and is replaced by the new cache line. The *random* replacement policy selects a random cache line to replace so that evictions are spread out uniformly throughout the set. The *least recently used* (LRU) replacement policy attempts to keep the most recently used data in the cache. LRU attempts to exploit the temporal locality of data: data that was accessed recently is likely to be accessed again. The policy counts accesses to each cache line and replaces the cache line that was accessed least recently. Faithful hardware implementations of the LRU are complex, so architects rely on the *first-in first-out* (FIFO) approach to approximate the LRU behavior. FIFO replacement policy simply replaces the oldest cache line in the set rather than the oldest used. The age of a cache line can be approximated by incrementing a small (3-bit for example) counter at every memory clock cycle. Replacement policies can have a significant effect on the cache behavior and continue to be an active area of research.

One can also rely on a *victim buffer* [5, 147] to provide associativity for a cache dynamically. A victim buffer is a small buffer of cache lines that were evicted most recently. When a data request arrives, the victim buffer is checked concurrently with its corresponding cache. The data found in the victim buffer is returned, replacing what is stored in the cache. The victim buffer would then contain the recently evicted cache line. In the example of the alternating data accesses, the victim buffer would provide the recently evicted data at every cycle without incurring extra latency. The benefit of a small victim buffer is the increase in effective cache hit rates without incurring latency, power and area costs as large as what is required by associative caches.

Eventually, all cache line requests reach the main memory, which fundamentally behaves differently from the on-chip caches. Caches are built from static random access memory (SRAM) which represents each bit using four to ten transistors. As a result, SRAM provides fast access but is expensive in terms of run-time energy and on-chip area, limiting the cache sizes to a few dozen megabytes at most. The main memory is made from DRAM chips which define a single bit using a capacitor and a transistor, ignoring sense amplifiers and other circuitry required to read or write stored data. Using capacitors for storage requires more time and energy to read and write data, thus DRAM includes a small SRAM cache. As a result, DRAM requires to be utilized differently to extract maximum

performance.

2.2.3.2 DRAM Behavior

Although DRAM provides high capacity (tens of gigabytes) memory at affordable prices, it is notoriously complex in terms of access behavior and characteristics, and is relatively slow in terms of access latency [9, 18, 23, 57, 76, 143]. The cost reductions per bit forced DRAM chips to rely on capacitors for storage, which leak their charge over time. As a result, DRAM storage must be periodically *refreshed*, reading and rewriting data in DRAM storage. The refresh operation costs both latency and energy. From an external point of view, DRAM chips support accessing data sized in cache line chunks. However, unlike an on-chip cache, internally to DRAM, every access fetches an entire *row* of data (up to 8KB) from one of the low-level memory circuit arrays into the *row buffer*.

The row buffer is implemented as fast static memory that provides cache-line-sized access. In a sense, the row buffer acts like an additional hidden cache that is located across DRAM chips. Fetching data into the row buffer from the internal DRAM storage is called *opening a row*. Because reads are destructive, this requires writing the data already in the row buffer back into DRAM storage, *closing the row*. The process of opening or closing the row requires a significant amount of energy to either sense discharging capacitors or charge them up with correct data. Accessing the data within the row, called an *open row access*, is dramatically faster and more energy-efficient than if the access requires opening a new row.

The memory controller is another critical piece of the DRAM memory system because it interfaces between the memory requests from the processor and the complex DRAM memory. Aside from managing DRAM storage to make sure the data does not degrade, the memory controller also accumulates data requests and reorders them based on which rows they map to. This improves the row buffer hit rates but introduces variability in access latency. A ray tracer carefully restructured to improve memory access patterns during traversal can help the memory controller increase the row buffer hit rate, thus reducing both DRAM latency and energy [76]. Looking ahead, accessing a contiguous stream of data comparable in size to a row buffer can represent a best-case use of DRAM in terms of achievable latency and power consumption.

2.2.4 Architecture Simulation

When designing application-specific accelerator architectures, cycle-accurate simulators are indispensable tools for rapid exploration of the potential design space. They provide the necessary details to capture the precise inner-workings of an entire system during the entire execution of the target application to completion. This thesis relies on the SimTRaX infrastructure [116, 117] to explore and tune architecture designs with thousands of hardware threads and complex memory subsystems without introducing high-level approximations. Other simulators are designed for different types of hardware architectures, like gem5 [15], Simics [89], SimpleScalar [19], GPGPU-Sim [8], and others. Please see the discussion in the original work [116, 117] on how the SimTRaX simulator compares to these.

Cycle-accurate architecture simulators help iterate on designs optimizing a particular application much faster than designing an ASIC. Simulators enable estimating performance benefits of proposed custom hardware units dedicated to accelerating a particular portion of an algorithm.

2.3 Discussion

This thesis combines ideas discussed so far to identify and to propose solutions to inefficiencies in the traversal and intersection phase of the path tracing algorithm. With the continued growth in the number of scene primitives, acceleration structures are essential for ray tracing performance because they improve algorithmic complexity of the search through all primitives per ray. Unfortunately, the reduction in computation and the reduction in the amount of scene data transferred from main memory comes at the cost of inefficient memory access patterns inherent in traversing acceleration structures. The random memory accesses significantly degrade the memory system performance, leading to high miss rates and miss penalties throughout the entire memory hierarchy. Moreover, the DRAM row buffer hit rates are affected and require opening new rows, costing additional latency and energy.

As the gap between available on-chip computational power and the available memory bandwidth per core widens, the computation savings offered by acceleration structures become less effective compared to the degrading performance of the memory system.

As a result, this thesis reorganizes the BVH acceleration structure and how it is used to make memory accesses resemble streaming rather than random access. In addition to other improvements, this reorganization makes the ray traversal phase of ray tracing use memory bandwidth more efficiently, improving the overall path tracing performance.

CHAPTER 3

PREVIOUS WORK

In this chapter, we provide a summary of the recent work on improving the ray tracing performance through both algorithmic modifications and custom hardware architecture designs.

3.1 Software Approaches

There is a large body of work evaluating and enhancing the performance of ray tracing on commodity hardware that can not be modified. Software approaches modify the ray tracing algorithm and its corresponding data structures to increase the efficiency of the available hardware. Approaches tend to modify their treatment of either rays or scene data in the process of ray tracing.

3.1.1 Focusing on Ray Data

The ray tracing algorithm described in the previous chapter considers processing individual rays one at a time. This description does not explicitly exploit *ray coherence* to gain performance. Coherent rays tend to travel through the scene near each other, visiting many of the same acceleration structure nodes and intersecting the same primitives. As a result, one could reuse scene data between many rays, thus improving the efficiency of the memory hierarchy and increasing the ray tracing performance. Methods focusing on ray data tend to target finding and exploiting ray coherence.

Ray Packets One of the most popular techniques in achieving the real-time ray tracing performance is tracing small collections of rays, *ray packets*, at once. The technique was popularized by the increase in the adoption of the SIMD instructions in commodity CPUs [13, 30, 34, 38, 111, 132, 135, 136]. Each ray packet typically contains a few dozen rays. Ray packets benefit ray tracing performance in two ways.

The first benefit of ray packets is that they improve scene data reuse across many

rays. As each ray traverses through the scene, it fetches scene data. Treating coherent rays independently results in reloading the same data many times. Collecting coherent rays into a packet to be traversed together lets the rays reuse the data across the entire packet, amortizing the cost of cache misses and reducing the total amount of scene data transferred from main memory. As a result, using ray packets enables to perform a lot more computation per byte of data fetched, improving ray tracing efficiency [137].

The second benefit of ray packets is that they map nicely to SIMD vector instructions within hardware architectures. One example of a vector instruction is component-wise multiplication of two four-wide data vectors at once. Alternatively, this operation would require four multiply instructions, one for each individual component. Data in ray packets is arranged so that the traversal and intersection computations can use the full width of the SIMD vector instructions across many rays in the packet. For example, one can intersect four rays with a single bounding box at the same time. Other combinations of intersecting many rays against many AABBs or triangles are possible [137].

The performance of ray packet approaches suffers when the number of active rays in a packet drops, as rays finish traversal after finding their hits, or as rays diverge from each other and traverse different portions of the scene. Primary rays can be quite coherent, but coherency quickly breaks down for secondary rays which rarely follow similar paths throughout the scene. Typically systems maintain a list of which rays are active in the packet and continue processing ever smaller portions of the packet until no active rays remain. As the ray coherence decreases, the benefits of ray packets decrease [3, 111, 135].

3.1.2 Focusing on Scene Data

Improving the acceleration structure design can lead to significant increases in the ray tracing performance by reducing the amount of computation required to find the closest object a ray hits. In that spirit, some recent research focuses on building BVHs with tighter bounds and lower SAH values. Other recent work focuses on modifying the acceleration structures to reduce how much data is fetched to generate an image frame either through increasing scene data reuse, reducing the amount of memory bandwidth used, or improving how the memory is accessed.

Popular BVH Flavors Split BVH (SBVH) behaves similar to kd-trees and allows splitting triangles between child nodes, duplicating their references, when it significantly reduces the SAH cost of the tree [123]. SBVH benefits scenes with long triangles or triangles rotated off axis because without splits their AABBs would contain a lot of empty space.

Linear BVH (LBVH) is designed to be built on massively parallel hardware architectures by leveraging Morton codes [92] to order all input primitives [81]. At each level of the tree b, the primitives are sorted using the radix sort according to the b most significant bits in their Morton code. Work queues are used to test many potential splits using SAH beyond certain LBVH depth.

Hierarchical LBVH (HLBVH) extends the LBVH approach and uses two phases to build the tree by relying on compress-then-decompress strategy to exploit spatial and temporal coherence in the mesh [100]. First phase uses the Morton codes to pre-sort all primitives and build a tree using a number of most significant bits in the Morton code. Each leaf node of this coarse tree can contain many primitives. In the second phase, each leaf is subdivided into subtrees by considering the next few bits in the Morton code as possible split positions. The authors handle dynamic scenes by rebuilding the BVH for every frame instead of refitting the BVH from the previous frame. HLBVH can be built very quickly on commodity GPU architectures, thus making it feasable to support dynamic scenes without a loss in ray traversal performance. The quality in terms of the SAH value can be improved through the use of tree rotations [67].

Scene Subdivision Scene data can be reordered in memory to improve the locality of the memory access patterns during ray traversal. Rearranging data can help increase cache hit rates because of spatial data locality and can help reduce the amount of memory traffic needed to render the scene by processing more rays through each portion of the scene.

The scene acceleration structure is subdivided into subsections based on either spatial extents like a grid [14, 97] or parts of subtrees called *treelets* [1]. The entire BVH tree can be subdivided into a set of treelets, each of which is a collection of connected BVH tree nodes that form a subtree. An example is illustrated in Figure 3.1. The methods associate a queue of rays for each subsection and process each ray in a given queue only through the associated scene subsection. As a result, the ray tracing algorithm can fetch a scene subsection into the on-chip caches and traverse a large number of rays through the

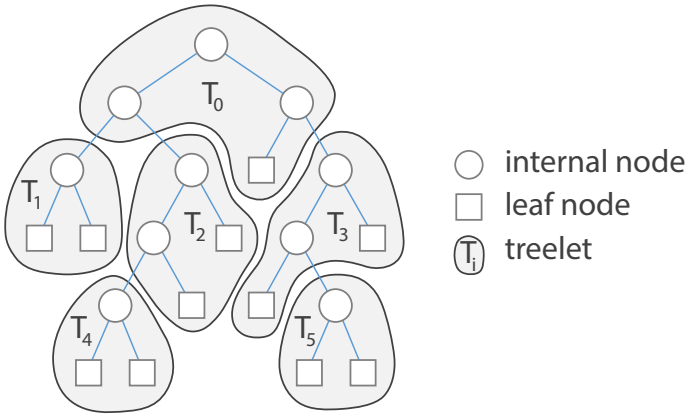


Figure 3.1: Illustration of a BVH tree subdivided into a collection of treelets. Each treelet is shown with gray background. Recreation from [1].

subsection leading to high cache hit rates. Rays are enqueued into different queues as they enter the next scene subsections. The size of each scene subsection can be chosen based on the desired cache behavior, for example the size of an L1 data cache. Rays can traverse different scene subsections either in front-to-back order, visiting each subsection once, or like a tree, reentering scene subsections from their children subsections. Revisiting scene subsections may require reloading the scene data, potentially wasting memory bandwidth.

One problem with such methods is the increased cost of maintaining ray queues: creating, grouping, sorting, and storing them. Rays must either be kept on chip where memory size is small, limiting the total number of rays that can be processed at the once. Alternatively, rays must be written into main memory, adding pressure on the memory subsystem and using memory bandwidth that could be instead allocated for scene or shading data.

3.2 Ray Tracing Hardware

In addition to the pure software approaches, one can design hardware architectures to accelerate parts of the ray tracing algorithm. Typically the hardware is taylored to a specific ray tracing configuration, like a particular acceleration structure. This section introduces the dedicated hardware architectures designed to accelerate ray tracing. Although not covered in this section, there are hardware accelerators dedicated to constructing the acceleration structure prior to rendering [127, 128].

Starting in the 1990s, several companies built commercially available dedicated hardware to accelerate ray tracing algorithms for the film industry. Unfortunately, the details

are scarce and only the AR350 architecture that drove the Advanced Rendering Technology's RenderDrive rendering appliance has been discussed recently [21, 39, 48]. Introduced in 2014, RayChip from SiliconArts [101] is a dedicated ray tracing processor for embedded and low power applications that was built from the T&I architecture which is discussed below. Instead of an ASIC designed only for ray tracing, recent commercial GPU architectures (Imagination PowerVR [129, 130] and NVIDIA Turing GPU [99]) include dedicated ray traversal logic accessible to shader programs. The architectural details are scarce at the time of writing.

Hardware acceleration for ray tracing can be very broadly categorized into SIMD and MIMD approaches [29]. SIMD accelerates applying a single computation to a lot of data, like intersecting a collection of rays against a bounding box or a triangle. Typically larger work sizes lead to better SIMD utilization defined by the proportion of the individual elements that are used during the computation. SIMD approaches suffer when there is data or control divergence which reduce the SIMD utilization: for example, some rays in a collection completing traversal earlier than others. Although MIMD techniques accrue costs from decoding the same instruction between many threads, they handle control divergence more gracefully because MIMD cores execute its own instructions independently.

3.2.1 SIMD Approaches

The SaarCOR (Saarbrücken's Coherence Optimized Ray Tracer) architecture [112, 113] implements a fixed-function pipeline for all phases of the ray tracing algorithm. It traces packets of 64 rays through a kd-tree, which is built on a CPU and is loaded into the accelerator memory ahead of time. The global scheduler unit controls and assigns work between the four ray tracing processors, each connected to DRAM through its own dedicated memory interface. Each ray tracing processor contains two additional units: a unit for ray generation and shading and a ray tracing core. The ray generation unit generates rays and dispatches them to the compute resources in the ray tracing core. Only Phong-like simple shading [106] is implemented, which can generate additional rays and feed them back into the system for traversal. The ray tracing core contains a ray collection unit which is simply an on-chip buffer of rays to be processed. Ray packets from the collection unit are fed into the fixed-function pipelines to traverse the kd-tree and intersect triangles. The

fixed-function units use the specialized memory controller to load tree node or triangle data from the corresponding caches backed by the main memory.

The RPU (Ray-Processing Unit) architecture [140, 142] extends the SaarCOR machine by making the pipelines for intersection and shading programmable, while keeping the kd-tree traversal units fixed-function. RPU also supports dynamic scenes because it can build the kd-tree on chip. The shading processor operates on four-wide vectors in a SIMD fashion with several threads executing simultaneously as a chunk. Each ray is assigned to its own thread and the shading processor can switch between processing multiple rays in a SIMT fashion when memory requests stall. This exploits data parallelism and hides memory access latency.

The MRTP (Mobile Ray-Tracing Processor) architecture [68–70] optimizes ray tracing for mobile processors by focusing on single ray performance through the use of reconfigurable stream multiprocessors. Rays are provided by the ray generation unit and processed sequentially by three reconfigurable stream multiprocessors: one is configured for traversal, another for intersection, and the third for shading operations. The main memory stores the acceleration structure and ray tracing kernel instructions. Each reconfigurable stream multiprocessor contains 12 scalar functional units, each of which implements several types of floating point operations. These scalar units can be dynamically reconfigured to operate in 12-wide or $3\times$ four-wide SIMD fashion.

The StreamRay architecture [45, 110] combines rays into a stream of data that is passed from one ray tracing hardware stage onto the next in a pipeline fashion. Each stage can be executed in parallel. Each stream of rays is filtered to identify active rays to be processed before each stage. Stages include traversal and intersection and process many rays in parallel. The filtering operation helps maintain coherent traversal for rays and reuse scene data. However, once rays become incoherent enough, the architecture suffers from reduced SIMD efficiency.

Modern GPU architectures are much more general purpose than the fixed-pipeline graphics architectures of the past. Current high-end GPUs support both arbitrary memory accesses and branching in the instruction set, and can thus handle operations required for acceleration structure traversal. However, an NVIDIA GPU for example assumes that every set of 32 threads (a *warp*) essentially executes the same instruction, and that instruc-

tions can thus be executed in a SIMD manner. Operations like branching, frequent in ray traversal kernels, are realized by transparent thread masking, which could lead to divergent threads, very low SIMD utilization, and poor performance. NVIDIA provides a ray tracing application programming interface (API) called OptiX [102] that can realize impressive ray tracing performance on existing GPUs, but the SIMD execution model can still be a limiting factor. AMD has a similar API and products in their GPUs called Radeon Rays [6]. Until recent architectural advances in GPUs, ray tracing implementations have relied on software techniques discussed previously [3, 4]. Recent research also proposes enabling ray traversal on existing GPU architectures by adding small dedicated units to keep overhead minimal [66].

In 2018, NVIDIA released the Turing GPU architecture with dedicated ray tracing hardware support [99]. Although offering staggering performance per GPU in terms of rays cast per second, the hardware is intended for hybrid rendering techniques. Such techniques generate primary visibility using rasterization [107] like all modern GPU architectures but allow casting rays into the scene for high quality secondary effects like reflections and shadows. The ray tracing cores in the Turing architecture accelerate both triangle intersection and traversal through an unknown BVH type. Unfortunately, the available architectural details do not describe the structure of the ray tracing cores or how they connect and utilize the memory subsystem.

3.2.2 MIMD Approaches

MIMD approaches to accelerate ray tracing typically fall into the SPMD form rather than true MIMD, where each core could be running a completely different program.

The Copernicus architecture [44] is based on an Intel Core2-series cores. The chips consist of 16 tiles, each with a large L2 cache shared by eight out-of-order processing cores each capable of supporting eight threads simultaneously. The architecture traces four-wide ray packets of rays through a kd-tree to utilize the four-wide vector SIMD functional units per core. The architecture is very flexible, relying on more general-purpose execution units, although it provides support for a dedicated accelerator block that is left unexplored.

The T&I (Traversal and Intersection engine) architecture [96] relies on dedicated ray traversal and intersection units to accelerate ray tracing, targeting single rays rather than

packets. The ray scheduler assigns active rays to execution units through intermediate buffers. The full execution pipeline consists of four stages: node traversal, triangle fetch, and two stages for the triangle intersection. The number of individual units in each stage is balanced based on optimal performance of the entire architecture. Stages are connected together by intermediate buffers. The first intersection stage performs ray-plane intersection, while the second intersection stage performs the division necessary to find the distance to the triangle from the ray origin. It is possible to avoid executing the second stage if the ray misses the triangle.

The SGRT (Samsung reconfigurable GPU based on Ray Tracing) architecture [82, 84, 85, 115] modifies the T&I engine in several ways. First, it replaces a single deep BVH traversal pipeline by three fixed-function pipelines that separate traversal into individual components: one fetches a node and checks if it is a leaf, another intersects a ray with an AABB, and the third performs traversal stack operations. The other change is the addition of the Samsung reconfigurable processor which generates and shades rays. It contains a coarse-grained reconfigurable array to help execute computationally expensive kernels more efficiently. The array is a collection of floating point execution units connected by a reconfigurable fabric made up of multiplexers.

One pitfal of the T&I and SGRT architectures is how they handle cache misses for the necessary intersection and traversal data when processing a ray through the pipelines. Because stalling the pipelines until data arrives is detrimental for performance, the authors keep retrying to process the ray through the pipeline until the data arrives, which results in unnecessary energy expendature. To save energy, one can add a small ray reorder buffer [83], which tracks which rays within the ray buffer are waiting for data. The ray reorder buffer helps prioritize scheduling rays that have all the data necessary for processing before feeding them into the intersection pipelines.

The RayCore architecture [95] builds on the T&I engine by unifying the implementation of each pipeline through common hardware that operates in different modes, by adding hardware for kd-tree construction and by adding support for shading. The T&I unit has an L1 cache, with four units connected to a unified L2 cache. When an L1 data access results in a cache miss, the pipeline simply retries to execute for the same ray until the requested data is returned from a higher level cache. The shading unit supports textures and simple

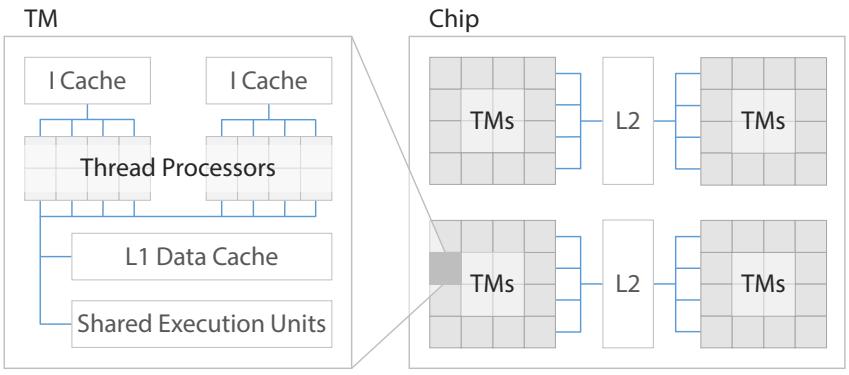


Figure 3.2: Basic TRaX architecture. A thread multiprocessor, *TM*, is shown on the left. Instruction caches, *I Cache*, are shared by thread processors within a TM. TMs can be tiled to create a single chip, where they share the *L2* data caches. DRAM and its memory controller which service the data requests from L2 caches are not shown on the right.

Phong shading [106].

The HART (Hybrid Architecture for Ray Tracing) architecture [94] expands the T&I engine to add support for dynamic scenes through dedicated hardware for BVH updates. The BVH itself is infrequently rebuilt on the CPU. The architecture also enables programmable shaders similar to a commodity GPU. The HART system uses shallow BVH trees where leaf nodes can store more than two triangles each. Each triangle is stored packed with its own bounding box to reduce the average number of triangle intersection tests per ray. Because this thesis does not consider dynamic scenes, the dedicated BVH update hardware is not discussed.

3.2.2.1 TRaX: Threaded Ray Execution Architecture

The TRaX (Threaded Ray Execution) architecture [77, 120, 121] explores a parallel tiled architecture with many small processing cores that share access to computation and cache resources. The smallest building block is a thread processor (TP) which is a simple, in-order, single-issue core with general purpose registers and a small local scratchpad memory managed by the programmer. TPs are combined to form a thread multiprocessor (TM), sharing access to other hardware units within the TM. Such units contain a multi-banked L1 data and instruction caches and execution units that are expensive in terms of area like a floating point divide, Figure 3.2. Each TP has its own program counter and can execute instructions that are different from other TPs on chip. Because each TP is a very simple

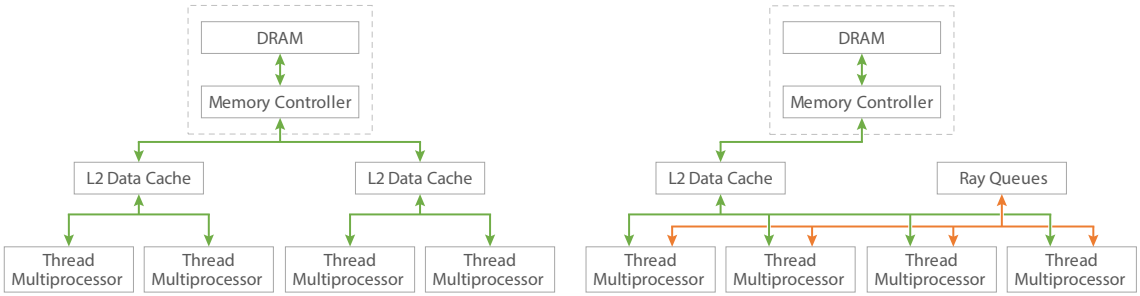
core requiring small area on chip, parallelism is achieved by increasing the number of TPs per chip rather than through architectural complexities like out-of-order execution or branch prediction. TRaX architectures rely on a simple memory hierarchy for its memory subsystem. The chip tiles several TMs which share access to L2 data caches, which in turn are backed by DRAM, Figure 3.3a.

The TRaX architecture is designed for general purpose computation and does not contain any hardware units specific to ray tracing like traversal or intersection pipelines. The unpredictable and embarrassingly parallel nature of ray tracing lets each TP, processing an individual ray, diverge in terms of the ray tracing program and use the shared on-chip resources more efficiently than commodity SIMD-style GPU architectures. The lack of synchrony between ray threads reduces resource sharing conflicts between TPs and reduces the area and complexity of each core. Given the appropriate mix of shared resources and low-latency L1 data cache accesses, TRaX can sustain a high instruction issue rate without relying on latency hiding via thread context switching.

Although scene data is reused across many TPs in the architecture, TRaX does not attempt to include any coalescing techniques for either ray or scene data. However, TRaX achieves more favorable performance per Watt and performance per unit area when compared to commodity GPUs of the time. The throughput of TRaX is primarily limited by the power and bandwidth consumption rather than the lack of computational resources.

3.2.2.2 STRaTA: Streaming Treelet Ray Tracing Architecture

STRaTA (Streaming Treelet Ray Tracing Architecture) [75, 76] extends the TRaX architecture to address the incoherent DRAM access patterns. STRaTA reorganizes the scene data using BVH treelets that enable the discovery of rays that all access the same cache-sized portion of the scene. STRaTA stores rays in on-chip buffers, thus restricting the total number of rays and limiting shared complexity. STRaTA is motivated by the observation that the DRAM uses about 60% of all energy used to render a frame, and that the entire memory subsystem can use up to 90% [76]. The primary opportunity lies in improving the memory system utilization by restructuring data access patterns to increase cache hit rates and reduce off-chip memory access energy. From an energy and delay perspective, this is a compelling target because fetching an operand from main memory is both slower and



(a) TRaX **(b)** STRaTA

Figure 3.3: Overview of the memory hierarchies for TRaX and STRaTA architectures. Arrows show the data flow. Green indicates scene data, while orange indicates ray data.

three orders of magnitude more energy expensive than performing a sigle floating point arithmetic operation [28].

Overall, the memory hierarchy for STRaTA is shown in Figure 3.3b. One difference from the TRaX architecture is the use of a single small L2 data cache shared by all TMs. Another difference is a large global ray queue unit to manage all ray requests from TMs, which lets different TPs contribute to processing a single ray. In the TRaX architecture, each TP stores its own ray until the ray processing is complete. Note that the on-chip ray queue and a small L2 data cache in STRaTA replace the large L2 data cache used by the TRaX architecture.

STRaTA relies on two mechanisms to reduce the total energy consumption per frame. The first mechanism subdivides the scene acceleration structure by using treelets [1, 97], which enable to access scene data by streaming it from DRAM. Rays are kept on chip and are accessed through dedicated hardware. Each TM processes many rays through its own treelet, sized to fit in an L1 data cache, increasing the L1 cache hit rates significantly.

The second mechanism uses special-purpose computation pipelines that can be reconfigured dynamically. The pipelines significantly reduce the register and L1 instruction cache accesses and the instruction decode overhead. The pipelines consist of execution units and multiplexers which can reconfigure the data flow to compute a ray intersection with either an AABB or a triangle. The pipelines replace a large number of conventional instructions with a single large intersection instruction. STRaTA does not investigate using reconfigurable pipelines for shading.

As a result, STRaTA significantly improves the power consumption for traversal and

intersection when compared to the TRaX architecture. Most of the savings originate from the improved DRAM access patterns, which have been designed to align with how the DRAM row buffer operates. Another benefit of STRaTA is that it scales much better with the number of TPs compared to the TRaX architecture. As the number of TPs per chip grows, the memory bandwidth becomes a bottleneck. Because STRaTA reuses the scene data more aggressively than the TRaX architecture, it becomes bandwidth limited with a larger number of TPs.

CHAPTER 4

DUAL STREAMING ALGORITHM

This thesis introduces the *dual streaming* approach for ray traversal, which reorders the traditional ray tracing algorithm to make it more suitable for hardware acceleration, considering DRAM behavior [118]. Our dual streaming approach organizes the memory access pattern of ray tracing into two predictable and prefetch-friendly data streams: one for scene data and one for ray data. Therefore, we pose ray tracing in a fully streamed formulation, reminiscent of rasterization [107]. The *scene stream* consists of the scene geometry data (including the acceleration structure) that is split into multiple *scene segments*. The *ray stream* consists of all rays in flight collected as a queue per scene segment they intersect. Our scheduler *prefetches* a scene segment and its corresponding ray queue from the main memory into on-chip buffers before both data are needed for traversal (i.e., perfect prefetching). Hence, the compute units no longer access the main memory directly. Rays at the same depth are traced together as a wavefront, so simulating each additional light bounce requires an additional computational pass. A predictable scene traversal order ensures that each scene segment is streamed *at most once* per ray wavefront. Thus, we regularize the memory traffic for scene data and reduce it to its absolute minimum.

The dual streaming algorithm provides a new ray traversal order that resolves some of the decades-old problems of high-performance ray tracing:

- *Scene data traffic from main memory is minimized.* Rays do not revisit scene segments during traversal. Since each scene segment is streamed at most once per ray wavefront, memory bandwidth is not wasted by fetching the same scene data several times. This is particularly important for large scenes and incoherent rays (such as secondary rays).

- *Random access to main memory during traversal is avoided.* All necessary scene and ray data are streamed on chip before they are needed for computation.

- *Memory latency is hidden by perfect prefetching.* A traditional solution hides memory latency by adding more threads, which is effective only when the memory system can process their requests fast enough. Instead, the dual streaming algorithm hides latency by perfectly predicting the next workload, which is dictated by the order in which scene segments are processed.

- *The memory access pattern for each stream fits how DRAM operates.* Both the ray and the scene streams are stored in memory as a collection of contiguous blocks of data that fit nicely to the preferred DRAM operation. When data is fetched, the entire block is streamed at once across the memory bus. This results in extremely high row buffer hit rates, minimizing DRAM operations needed to access the requested data. Thus, DRAM chips process requests faster and at lower energy, further resulting in better bandwidth utilization.

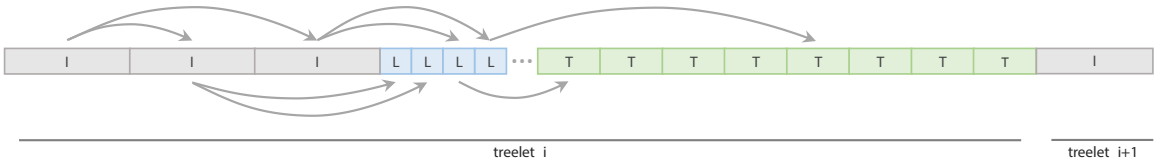
Notice that all of these improvements relate to the memory system, since traditional ray tracing, especially for large scenes, can be bound by the memory accesses rather than the amount of computation. Data movement is also the main culprit for energy use. Therefore, all of these outcomes are critical to addressing the traditional problems with high-performance ray tracing in terms of both rendering speed and energy use.

Overall the dual streaming algorithm implements path tracing as a sequence of processing steps executed fully for every wavefront of rays. First, all rays in the wavefront are generated, which includes either all primary rays created by the camera or all secondary and shadow rays created by evaluating materials. Then, all rays in the wavefront are traversed through the scene. The algorithm generates the next light bounce only once all rays in the previous wavefront have been processed. This thesis focuses on accelerating only the traversal and intersection phase of path trhacing, and thus does not consider the shading phase that evaluates materials.

4.1 The Two Streams

The main goal of the dual streaming algorithm is to eliminate the irregular accesses to the main memory and to minimize the scene data transfer by reformulating ray tracing as processing of two separate data streams: a *scene stream* and a *ray stream*.

(a) BVH Nodes. Interior nodes use 16 words (64 bytes), while leaf nodes use 2 words (8 bytes).



(b) BVH Treelets. Each node is stored according to (a). Boxes are colored based on data type: gray for interior nodes (I), blue for leaf nodes (L), and green for triangles (T). Arrows indicate pointers from a node to its first child node or to the first triangle.

Figure 4.1: In-memory layout for the scene stream that consists of BVH treelets.

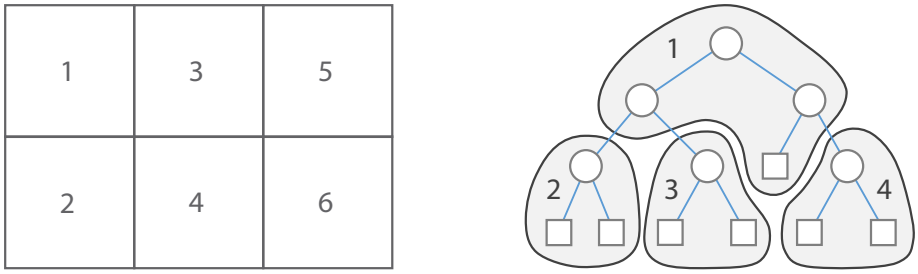
The scene stream consists of multiple *scene segments* that collectively represent the entire scene geometry data and that can be processed independently. The memory footprint of each scene segment is sized as a multiple of the DRAM row buffer capacity to enable efficient streaming. Although most acceleration structures could be used to generate scene segments, our implementation splits a BVH into treelets [1], each containing both internal and leaf nodes, as well as the scene geometry (e.g., triangles). Each scene segment is organized in memory so that all of its nodes and geometry can be accessed sequentially to improve the spatial locality of the data. Each scene segment stores all of its acceleration structure nodes first followed by the primitive data, interleaving both data types in memory. Note that this memory layout is different from the traditional acceleration structure implementations that separate the block of acceleration structure node data from the block of geometry data, even if the primitives themselves are reordered.

Figure 4.1 shows the memory layout for the scene segments made up of BVH treelets. The acceleration structure node format is described in Figure 4.1a. The interior nodes use 64 bytes, while the leaf nodes use 8 bytes. The interior nodes are larger because they use 48 bytes extra to store the AABBs for their children. Note that the first word in each node representation uses the sign bit to distinguish between node types. Optionally for internal nodes, the first word can also pack the two-bit representation (buts 30 and 31) of the axis which was used to separate the children nodes. The leaf node uses the remainder of the bits in the first word to specify the number of primitives it stores. Words that store near

address locations point at either a child node (for interior nodes) or the first primitive (for leaf nodes). In a *subtree ids* word, intior nodes store indexes of the treelets that the chidlren nodes belong to, each represented as a 16-bit integer. As shown in Figure 4.1b, each treelet is stored in its entirety in memory, including both the acceleration structure node data and the triangle data. Node data is stored first, following a typical BVH approach where sibling nodes are placed next to each other. Triangles are stored after, such that triangles belonging to the same leaf node are stored together. Each triangle is stored simply as 9 words (36 bytes): a 3D position for each vertex. Other triangle attributes can be placed immediately after. Note that storing a set of indexes into the vertex position array to define triangles can potentially significantly reduce the overall scene storage. Treelets are stored one after another in memory without any padding.

The ray stream is the collection of all rays that are in flight, split into multiple queues, one per scene segment. The ray stream contains basic ray information: origin, direction, and a ray index. The index specifies which image sample any given ray corresponds to. Since the scene segments are traversed independently, there is no need to store a global traversal stack for each ray, significantly reducing the storage overhead for the ray state. Each ray queue is stored as a linked list of *ray buckets*, where each ray bucket contains a number of rays. The ray bucket header stores the number of rays contained within and a pointer (a memory address) to the next ray bucket in the list. Because each ray uses the same number of words for storage, we can treat each ray bucket as an array rather than a linked list, and thus storing the number of rays per ray bucket is sufficient. Although rays within a single ray bucket are spatially coherent, different ray buckets within a single ray queue are unlikely to be. Ray buckets located sequentially in main memory can belong to different ray queues and can be filled out of order, leading to fragmentation.

Throughout traversal, rays are added to the ray queues of scene segments they need to visit, and are removed from the ray queues as the associated scene segments are processed. Because our scene segmentation is hierarchical, the ray queue for a given scene segment is filled as the algorithm processes the rays through its parent scene segment. The ray queue is drained only after the parent scene segment has been fully processed, since no more rays can enter into the ray queue.



(a) Grid **(b)** BVH Treelets

Figure 4.2: Illustration of the scene segment traversal order for (a) a 2D grid and (b) BVH treelets. Each number shows when a particular scene segment can be processed.

4.2 Predictive Traversal Order

The construction of scene segments dictates the fixed traversal order based on how rays can flow from one segment into the next. Because the dual streaming algorithm aims to minimize the amount of scene data transferred, the algorithm maximizes the reuse of every scene segment by processing all of its rays to completion. As a result, the algorithm aims to never reload scene segment data unnecessarily. Making sure each scene segment is fetched only once forces a specific order in which scene segments are processed, the *segment traversal order*. Tree-based segmentations (e.g., BVH treelets) impose a hierarchical relationship between scene segments in which rays flow from a parent to its children, but never from children back to their parent. This eliminates the need to reload scene segments, since rays are not allowed to revisit them.

The order in which scene segments are loaded is determined by the scene segmentation used. The segment traversal order may follow strict breadth- or depth-first ordering, or may leverage run-time statistics, like the distribution of rays among scene segments. For example, we can consider the traversal order in the case where scene segments are grid cells, Figure 4.2a. Each cell would load one after another from one grid corner towards the other. Our particular implementation which uses BVH treelets as scene segments focuses on the depth-first traversal order, Figure 4.2b. Once all rays finish traversing through a particular scene segment, its children become available to be processed. If a scene segment's ray queue is empty when it is time to be processed, the scene segment (and all of its descendants) can be safely skipped, since no rays visit that part of the scene.

The dual streaming algorithm maintains a list of all scene segments that are currently

being processed, which we refer to as the *working set*. All scene segments in the working set have nonempty ray queues and are processed in parallel. The algorithm also maintains a list of scene segments that can be scheduled into the working set for future processing. As scene segments finish processing, their children are added into this list. The process of moving a scene segment into the working set is called *scheduling*. Several approaches can be used to schedule scene segments into the working set. Moreover, rays from the same ray queue, which traverse the same scene segment, can be distributed between different processing threads. Traversal of the current ray wavefront ends when the ray queues for all scene segments have been emptied. Due to the predefined scene segment traversal order, each scene segment is processed (and therefore loaded) at most once per ray wavefront (i.e., once per ray bounce).

4.2.1 Scheduling Scene Segments

We consider four approaches to select which scene segment is added into the working set. Each approach scans through the queue of scene segment indexes to be processed and can use the number of enqueued rays to choose whether to schedule a specific scene segment. Once a scene segment is inserted into the working set, its data is prefetched from the main memory. As a result, each approach chooses if and when scene segments are prefetched from memory, resulting in different memory traffic and access patterns.

The *conservative* scheduler guarantees to fetch only the scene segments necessary to process the current ray wavefront. Scene segments are added into the working set in the same order generated by the fixed scene segment traversal order. The conservative scheduler considers only the very first scene segment in the segment queue and schedules that scene segment only if it contains rays. If the scene segment has no rays and its parent is evicted from the working set, then the very first scene segment is popped from the segment queue without being inserted into the working set. This behavior ensures that both the given scene segment and its children are skipped during processing without being prefetched unnecessarily.

The *aggressive* scheduler aims to fill the working set with scene segments aggressively by scheduling the first scene segment in the segment queue regardless of whether that scene segment has any rays. Once the scheduler determines that no rays would visit a

particular scene segment that is already in the working set, it is evicted, canceling any data prefetch instructions yet to be issued.

The *opportunistic* scheduler balances the two previous extremes. It aims to prefetch scene segments more aggressively than the conservative scheduler, but, unlike the aggressive scheduler, still aims to fetch only the necessary scene data. We explore two flavors of this scheduler. The *opportunistic 1st* scheduler scans through the segment queue from front to back and selects the first scene segment that contains rays, potentially skipping some scene segments in the queue. The *opportunistic max* scheduler selects the scene segment with the most rays. A scene segment is removed from the segment queue without insertion into the working set if the scene segment contains no rays and its parent scene segment has been evicted from the working set. The opportunistic schedulers can load and process scene segments in an order that is different from the fixed segment traversal order.

4.2.2 Ray Duplication

Within a scene segment, each ray follows a typical BVH traversal implementation using a local stack for the nodes to visit. However, when a ray finds an exit point from the current scene segment into one of its child scene segments, the ray does not immediately follow the path. Instead, the ray is duplicated into the ray queue for the child scene segment and continues traversal *within* the current scene segment until all exit points are found. A ray can be copied into many child scene segments after it traverses through a given scene segment. We apply early ray termination locally: if a ray hits a primitive within the current scene segment, it avoids traversing any nodes and enqueuing into any child scene segments farther than its hit point. Ray duplication avoids reloading scene segments per ray wavefront, because rays do not revisit the parent scene segment to reach a sibling scene segment.

We maintain a shared hit record for each set of duplicate rays, which must be updated each time an intersection is found. Since the number of rays is very large, the hit records are stored in main memory rather than sent along with each ray. Updating the shared hit records – and ray duplication in general – presents special challenges. Several scene segments that are in the working set can contain duplicates of a given ray. Because all scene segments in the working set are processed in parallel, updating the shared hit record

must be atomic.

4.2.3 Early Ray Termination

Because rays can be duplicated, implementing the early ray termination optimization becomes nontrivial. When a ray-primitive intersection is found, there is no easy way to check whether this distance is the global minimum, or if a duplicate ray being traced simultaneously in another scene segment has found a closer hit.

Because the shared hit records are kept in main memory, early ray termination tests must access the memory during traversal, potentially incoherently. Although these accesses should get coalesced by the memory controller, threads could stall waiting for the current hit distance to return from the main memory. The performance impact of testing the shared hit record atomically before every ray-node intersection during traversal may be prohibitive. As such, we do not consider this approach for the early ray termination. Instead, we consider two other mechanisms: *pre-test* and *post-test*.

The pre-test checks the ray hit record before traversing a ray through the current scene segment immediately after fetching the ray from a ray queue. Although this approach avoids unnecessary ray traversal through the current scene segment, the ray still needs to be enqueued into this segment for processing.

The post-test checks the ray hit record after traversal through the current scene segment and before enqueuing the ray into another scene segment. This approach does not save the cost of traversal through the current scene segment but avoids enqueuing rays into child scene segments unnecessarily.

4.3 Algorithm Pseudocode

This section describes the pseudocode for the dual streaming algorithm combining all of the concepts presented so far. The algorithm assumes the acceleration structure is built and all of the scene data is already stored in main memory. The algorithm that renders the entire image is shown in Algorithm 4.1. After setting up several helpful color buffers and hit records (lines 1-6), the algorithm iteratively processes all rays one wavefront at a time until the maximum path length is reached (line 7). Each ray wavefront requires three processing phases.

```
   // initialize buffers
1  parallel foreach pixel in image do
2      image.SetPixelColor (pixel, black);
3      thruput.SetPixelColor (pixel, white);
4      shadowColor.SetPixelColor (pixel, black);
5      DS::ResetHitRecord (pixel);
6  end
   // iterate over ray wavefronts.  Extra pass for shadow rays from last
      bounce
7  foreach depth ≤ max_depth do
       // 1.  generate rays from camera or from shading a hit.  Save into
          queue for root scene segment
8      parallel foreach pixel in image do
9          if depth == 0 then
10             rays = camera.GenRay (pixel);
11         else
12             rays = GenSecondaryRays (thruput, shadowColor, pixel, depth);
13         end
14         DS::SaveRay (rays, 0);
15     end
16     barrier;
       // 2.  trace rays:  fetch a ray from stream, traverse through
          corresponding scene segment, add into next scene segments
17     parallel while ray = DS::ReadRay () do
18         TraverseRay (ray);
19     end
20     barrier;
       // 3.  accumulate colors from shadow rays into image
21     parallel foreach pixel in image do
22         color = ColorFromHit (pixel, shadowColor);
23         image.Accumulate (pixel, color);
24     end
25     barrier;
26 end
```

Algorithm 4.1: Pseudocode for the dual streaming algorithm. Execution of each parallel loop is distributed between a collection of threads based on thread index. The barriers that synchronize threads in between wavefront processing phases ensure correctness.

The first phase generates the rays at a given ray depth for the current ray wavefront (lines 8-16). These rays can be either primary (line 10) or secondary (line 12) which include shadow rays. The generation of the secondary rays also shades hit points by evaluating the material properties of the intersected surfaces. Shading also modifies the path throughput for the current image sample. Further details are discussed below.

The second phase traverses all of the rays in the current wavefront (lines 17-20). Note that fetching a new ray to proces (line 17) maintains the scene segment currently being processed. The abstraction of tracking which scene segment a thread is processing simplifies the implementation. As a result, fetching a new ray can schedule a new scene segment into the working set. Details are discussed further below.

The third phase of processing a ray wavefront simply accumulates sample colors into the image (lines 21-25). The shadow rays are used to transmit how much illumination is transferred from the light onto the image (line 22).

Each phase of the dual streaming algorithm is written assuming it is executed in parallel and is depicted using **parallel** loops. It is assumed that the work is distributed between processing threads automatically, and the work for each thread is treated in a fully sequential manner. For simplicity, all functions specific to the dual streaming algorithm are labeled using the **DS::** prefix. Other nontrivial functions are described in detail separately below.

The function that generates the secondary rays, **GenSecondaryRays(...)**, is shown in Algorithm 4.2. Rays are generated only if the current ray hits an object, and thus a nonempty hit record is available (lines 2-5). If the path has already terminated for this pixel, then the pixel will store a miss in its hit record. First, the method generates a shadow ray by sampling the lights in the scene to select one (line 7). Because shadow rays are used to account for how much light is transmitted onto the image, the contribution of the chosen light to the image is stored assuming the light is not occluded (line 10). If the maximum path length has not yet been reached, light can bounce from the hit surface. In this case the function also generates a secondary ray (lines 11-18). A new ray is generated by sampling the material properties of the hit surface (line 12). The path throughput is modulated by the material properties (line 14) and is recorded in the path throughput buffer, `thruput` (line 15). The method can return either no rays, a shadow ray, or a shadow and a secondary

```
1  rays = function GenSecondaryRays(thruput, shadowColor, pixel, depth):
       // get hit record for pixel's ray.  Misses and terminated paths are
          identified the same - hit record stores a miss
2      hitInfo = DS::FetchHitRecord (pixel);
3      if not hitInfo.didHit then
           // background color or environment map can be handled here
4          return null;
5      end
       // generate shadow ray.  Store contribution color assuming light not
          occluded based on path throughput so far
6      pathThruput = thruput.GetPixelColor (pixel);
7      light = scene.SampleLight ();
8      shadowRay = light.GenShadowRay ();
9      shadowRay.depth = depth + 1;
10     shadowColor.SetPixelColor (pixel, pathThruput * light.color);
       // generate secondary ray, modulating path throughput by material
          properties
11     if depth < max_depth then
12         { secRay, brdf } = hitInfo.brdf .Sample ();
13         secRay.depth = depth + 1;
14         pathThruput = pathThruput * brdf * cos;
15         thruput.SetPixelColor (pixel, pathThruput);
16     else
17         secRay = null;
18     end
       // return 1 or 2 generated rays
19     return { shadowRay, secRay };
20 end
```

Algorithm 4.2: Pseudocode for the dual streaming algorithm function that generates secondary rays based on a given hit point. The function **GenSecondaryRays** is used on line 12 of Algorithm 4.1.

rays. An extension to generate several shadow rays, which can sample multiple lights, or several secondary rays is straight-forward and is omitted for brevity.

The function that fetches color contributions to the image from individual pixel samples, **ColorFromHit(...)**, is shown in Algorithm 4.3. It simply checks whether a shadow ray is occluded (line 4). If not, then the function returns the color stored in the shadow transmission buffer (line 5). Otherwise, if the shadow ray is occluded, it hits an object and thus the light is occluded. Therefore, since no illumination can reach the camera along this light path, the function returns the black color (line 2). Prior to returning any color, the

```
1 color = function ColorFromHit(pixel, shadowColor):
      // black returned by default
2     color = black;
      // if shadow ray misses, light is visible.  Return its contribution
         to pixel
3     shadowHitInfo = DS::FetchHitRecord (pixel);
4     if not shadowHitInfo.didHit then
5         color = shadowColor.GetPixelColor (pixel);
6     end
      // reset hit records for secondary and shadow rays at given pixel
7     DS::ResetHitRecord (pixel);
8     return color;
9 end
```

Algorithm 4.3: Pseudocode for the dual streaming algorithm function that computes the contribution to the pixel color from the current ray wavefront. The function **ColorFromHit** is used on line 22 of Algorithm 4.1.

function resets the hit records for both the secondary and the shadow rays (line 7). The output of this function is sent into the image color accumulation function which computes the image color per pixel, (line 23) in Algorithm 4.1.

The function that fetches a ray to be processed by a single thread, **DS::ReadRay()**, is shown in Algorithm 4.4. It tries to prioritize the ray queue for the scene segment that the thread is currently processing. If the current scene segment has some rays assigned to it, the function will return the next ray in the ray queue (lines 3-6). If there are no rays left in the current scene segment, the function schedules the new scene segment into the working set (line 7) and fetches a ray from the corresponding ray queue (line 10). If no ray can be fetched, all rays in the wavefront have been processed. Then the function finally returns **null** signifying the stopping condition for traversing rays in a given wavefront (line 12). Although the given description treats each ray queue data structure as a queue, the conversion to a more-realistic implementation using a linked list of blocks of rays is straight-forward and is omitted for brevity.

Finally, the function that traverses a given ray through the scene segment and intersects it with all appropriate primitives within, **TraverseRay(...)**, is shown in Algorithm 4.5. The pseudocode includes the details for both pre- and post-test early ray termination approaches discussed in Section 4.2.3. The implementation follows the traditional BVH

```
1  ray = function DS::ReadRay():
       // get id of the current thread
2      threadId = GetThreadId ();
       // if current scene segment has rays for processing, return next ray
3      curSegmentId = DS::GetCurrentSceneSegment (threadId);
4      if DS::GetNumRays (curSegmentId) > 0 then
5          return DS::GetRayFromQueue (curSegmentId);
6      end
       // schedule new scene segment, return next ray for processing
7      nextSegmentId = DS::ScheduleSceneSegment (threadId, schedulerType);
8      if DS::GetNumRays (nextSegmentId) > 0 then
9          DS::SetCurrentSceneSegment (threadId, nextSegmentId);
10         return DS::GetRayFromQueue (nextSegmentId);
11     end
       // all rays in wavefront processed.  Return end condition
12     return null;
13 end
```

Algorithm 4.4: Pseudocode for the dual streaming algorithm function that fetches a ray to be processed by the current thread. The function **DS::ReadRay** is used on line 17 in Algorithm 4.1. The pseudocode relies on several dual streaming algorithm helper functions identified with **DS::** prefix, whose functionality is described by their name.

traversal which relies on a traversal stack that stores which BVH node is to be traversed next, although in our case we store the node memory address instead of its index (lines 11-13). The algorithm proceeds until the traversal stack is empty, which signifies that all nodes in the scene segment that a ray can traverse through have been processed. After loading a node on top of the stack from memory, the algorithm intersects its AABB with the ray (lines 16-17). If the ray misses the AABB, and thus the node, the algorithm grabs the next node address from the stack (lines 26-27). If the stack is empty, the ray updates its hit record, recording a closer hit that may have been found in this scene segment, and the traversal stops (lines 20-25). If the ray intersects the AABB, and thus the node, it continues traversal (lines 18-19), which is described in Algorithm 4.6. If the node is a leaf, the ray intersects with all referenced primitives (lines 2-16). If the node is interior, we must test which scene segment it belongs to (line 18). If the node is part of the same scene segment currently being traversed, the traversal algorithm pushes the address of the farther child node onto the traversal stack (lines 19), and continues traversal through the child node that is closer (lines 20). If the node belongs to a scene segment different than the current

one, the ray is added into the corresponding ray queue (lines 29-32), before continuing to traverse the current scene segment by getting the next node from the traversal stack (lines 26-27 of Algorithm 4.6).

The pre-test for the early ray termination is shown on lines 2-10 in Algorithm 4.5. It fetches the global hit depth for the given image sample (lines 3-4) and compares against the current ray maximum depth, set when the ray was enqueued into this scene segment. Traversal continues with the shorter of the two distances (line 14). Shadow rays skip traversal if the global distance is closer than the ray distance because a hit was already found elsewhere marking this shadow ray occluded (lines 5-7). The post-test, shown in lines 23-28 of Algorithm 4.6, checks the global ray hit depth against the hit depth of the currently hit node before enqueuing the ray into another scene segment. Note that for simplicity, the pseudocode enqueues the ray immediately rather than aggregating all of the scene segment indexes into an array. Such an approach reduces the number of accesses to the global hit record but introduces complexity. An implementation that relies on such a method must carefully consider the size of the array because any scene segment can have an unbounded number of child scene segments. As a result, an implementation must occasionally empty the array by enqueuing the ray into the appropriate scene segments.

4.4 Discussion

Although the dual streaming algorithm reformulates ray tracing to use the streaming memory access pattern, it exposes new challenges and restrictions. First, to maximize scene data reuse, all rays in flight are stored in and streamed from the main memory, instead of just a small number of rays being stored (and processed) on chip. Although streaming rays from DRAM is highly efficient, fetching rays introduces additional load on main memory, which we found to be offset by the reduction in scene traffic for most scenes. Secondly, our implementation of the predictable segment traversal order requires some rays to be duplicated. Even though the duplication eliminates the need to store a traversal stack per ray, it still puts extra pressure on the memory system, in terms of both storage and bandwidth, and requires atomic hit record updates. Finally, unlike traditional ray tracing, implementing efficient early ray termination and optimizing the ray traversal order with the dual streaming algorithm is nontrivial and is left for future work.

```
1  function TraverseRay(ray):
       // perform pre-test early ray termination
2      if pre_test then
3          globalHitInfo = DS::FetchHitRecord (ray.pixel);
4          globalHitT = globalHitInfo.rayDist;
5          if globalHitT ≤ ray.t and ray.isShadow then
6              return;
7          end
8      else
9          globalHitT = ∞;
10     end
       // initialize traversal stack, etc.
11     stack [max_stack_size];
12     sp = 0;
13     nodeAddr = ray.nodeAddr;
14     ray.t = min (ray.t, globalHitT);
       // continue tracing until stack is empty
15     while true do
           // load current node from memory, and intersect ray with its AABB
16         node = LoadNodeByAddr (nodeAddr);
17         nodeHitInfo = node.IntersectRay (ray);
18         if nodeHitInfo.didHit then
               // ...  process the node.  See Algorithm 4.6 ...
19         end
           // traversal stack is empty:  scene segment processing finished.
              Update ray hit record automically
20         if sp == 0 then
21             if not ray.isShadow then
22                 DS::UpdateHitRecord (ray.pixel, ray.hitInfo);
23             end
24             return;
25         end
           // get next node address, continue scene segment traversal
26         sp = sp - 1;
27         nodeAddr = stack [ sp ];
28     end
29 end
```

Algorithm 4.5: Pseudocode for the dual streaming algorithm function that traverses a ray through its current scene segment. The function **TraverseRay** is used on line 18 in Algorithm 4.1. Processing the BVH nodes (lines 18-19) is shown in Algorithm 4.6. The pre-test early ray termination is shown on lines 2-10. The post-test early ray termination is shown in Algorithm 4.6.

```
   // ...  continues from line 18 in Algorithm 4.5
1  if node.isLeaf then
       // leaf node:  intersect against triangles within
2      for trild in node.triangles do
3          triangle = LoadTriangle (trild);
4          hitInfo = triangle.IntersectRay (ray);
5          if hitInfo.didHit then
6              if ray.isShadow then
                   // stop traversing shadow rays once hit is found
7                  DS::UpdateHitRecord (ray.pixel, hitInfo);
8                  return;
9              else
                   // non-shadow ray:  save closest hit for current segment
10                 if hitInfo.rayDist < ray.t then
11                     ray.t = hitInfo.rayDist;
12                     ray.hitInfo = hitInfo;
13                 end
14             end
15         end
16     end
17 else
       // interior node:  continue traversal.  May enqueue into new segment
18     if node.segmentId == ray.segmentId then
19         stack [sp ++] = node.FarChildAddr (ray);
20         nodeAddr = node.CloseChildAddr (ray);
21         continue;
22     else
           // post-test eary ray termination updates ray hit distance
23         if post_test then
24             globalHitInfo = DS::FetchHitRecord (ray.pixel);
25             globalHitT = globalHitInfo.rayDist;
26         else
27             globalHitT = ray.t;
28         end
           // enqueue if node hit closer than current ray distance
29         if nodeHitInfo.rayDist < globalHitT then
30             DS::SaveRay (ray, node.segmentId);
31             return;
32         end
33     end
34 end
```

Algorithm 4.6: Pseudocode for the dual streaming algorithm traversal of a ray through a BVH node, used on line 18 in Algorithm 4.5. The post-test early ray termination is shown on lines 23-28.

CHAPTER 5

DEDICATED DUAL STREAMING HARDWARE ARCHITECTURE

The dedicated hardware architecture designed to accelerate the dual streaming algorithm follows closely the algorithm described in the previous chapter. The architecture includes specialized hardware units to handle the ray and scene streams, which service the functions identified with the **DS::** prefix as listed in the algorithms in the previous chapter.

Our hardware implementation of the dual streaming algorithm is shown in Figure 5.1. The design follows the single program multiple data (SPMD) paradigm, with independent control flow for each processing thread. The architecture partitions a large number of thread processors (TPs) into a number of thread multiprocessors (TMs) to allow TPs to share units that are expensive in terms of area, like fixed-function intersection units, ray staging buffers, and L1 caches. Each TP is a simple in-order hardware processor with its own program counter and a small local scratchpad memory.

The complete streaming processor is built from many TMs which share access to several global units: the *stream scheduler*, the *scene buffer*, and the *hit record updater*. These units connect to the memory controller, which interfaces with the off-chip DRAM. Figure 5.1 shows details for these global units and Table 5.1 describes their on-chip areas.

Unlike traditional CPU or GPU architectures, our dual streaming implementation features no large L2 data caches. Instead, the chip area is used for dedicated scene and ray buffers, which are essentially large static random access memory (SRAM) scratchpads. Compared to typical caches of similar capacity, such scratchpads are simpler to implement, faster to access, and consume less energy per access.

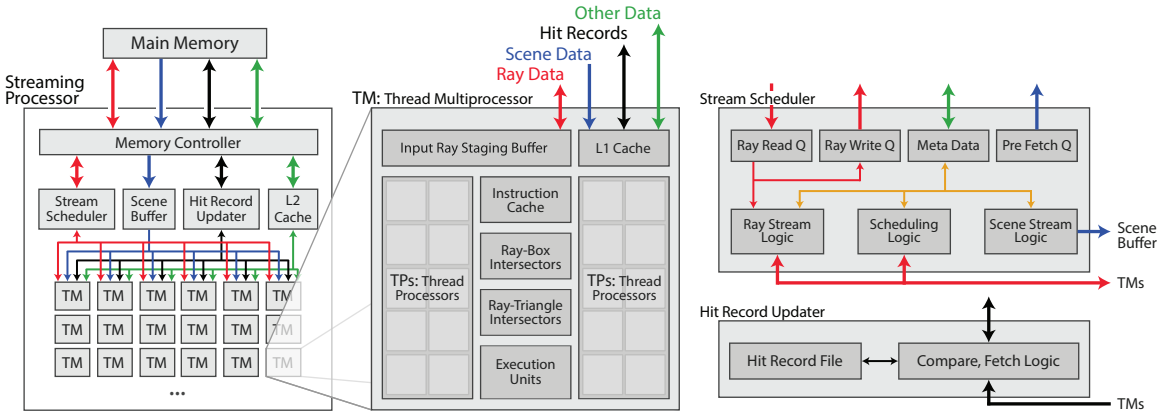


Figure 5.1: Overview of our dual streaming hardware architecture. Lines connecting hardware modules indicate the flow of data colored by its type: (red) ray data, (blue) scene data, (black) hit records, and (green) other data. The streaming processor uses many thread multiprocessors (TMs) for computation, which share chip-wide stream units. A TM combines many lightweight hardware thread processors (TPs) that share instruction cache and computation units.

	Dual Streaming	**STRaTA**
Common System Parameters		
Technology Node	65nm CMOS	
Clock Rate	1GHz	
DRAM Memory	4GB GDDR5	
Total Threads	2048	
On-Chip Memory		
L2 Cache	512KB, 32 banks	
On-Chip Ray Queues	*N.A.*	4MB
Scene Buffer	4MB	*N.A.*
TM Configuration		
TPs / TM	16	16
L1 Cache	16KB, 8 banks	32KB, 8 banks
Ray Staging Buffer	2×2KB	*N.A.*
Area (mm^2)		
Memory Controller	13.1	13.1
Scheduler	0.53	negligible
Caches / Buffers	190.4	159.7
Compute	57.1	57.1
Total	261.1	229.9

Table 5.1: Hardware configurations used for architectural performance evaluation.

5.1 Specialized Hardware Units

The dedicated hardware architecture implementing the dual streaming algorithm relies on several special-purpose hardware units for acceleration. Most of the complexity is in the memory system hierarchy. Each TP acts as a simple data processor with access to dedicated ray intersection pipelines. All of the complex control logic is located within the stream scheduler.

5.1.1 Stream Scheduler

One of the key units in our implementation is the stream scheduler, shown in Figure 5.1. The stream scheduler marshals the data required for ray traversal to prevent TPs from accessing main memory directly and randomly for both scene and ray stream data. The stream scheduler also tracks the current state of traversal, including the working set of active scene segments, the mapping of TMs to scene segments, and the status of the scene and ray streams.

As discussed in the previous chapter, the scene is partitioned into a number of scene segments each of which can be traversed independently. With the exception of the first scene segment (e.g., the root treelet, when using BVH treelets as scene segments), scene segments become eligible for traversal only after their parent has been traversed. When the traversal of a scene segment is completed (i.e., its ray queue is depleted), the stream scheduler replaces the scene segment with another. After adding a scene segment to the working set, the stream scheduler transfers the corresponding data from DRAM to the scene buffer.

Tracking the scene segment stream incurs little overhead per segment: starting memory address and the number of cache lines transferred so far. We have found that streaming eight scene segments simultaneously performs well, and requires modest area within the stream scheduler: 40 bytes of storage and some counters.

Rays are partitioned into a number of queues, with one queue per scene segment. While only a small subset of all active rays can fit on chip, the rest are stored in DRAM until they can be processed. Each ray queue is stored as a linked list of ray buckets. Within its header, each ray bucket stores the next bucket's memory address and a ray counter. Although some ray buckets may be filled only partially, buckets maintain constant size and

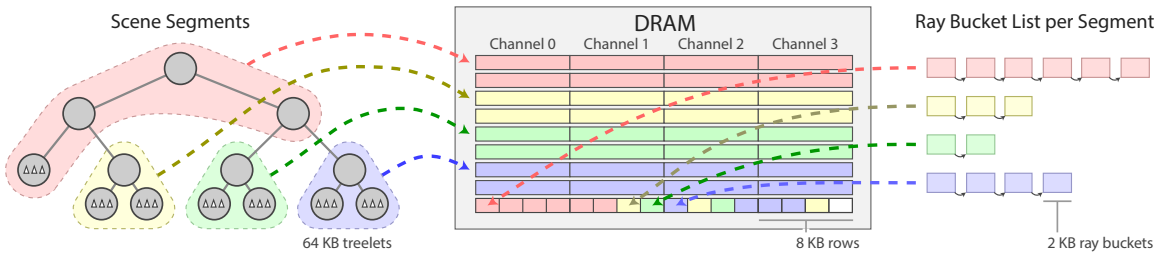


Figure 5.2: Layout of both ray and scene streams in DRAM. Each scene segment (left) is placed contiguously in DRAM, including triangles. Ray stream (right) consists of a linked list of buckets, which may be stored in a fragmented fashion. Note that the stream scheduler stores pointers to scene segments and the first (head) ray bucket, depicted by dashed arrows.

row buffer alignment in DRAM. In our implementation, four 2KB ray buckets perfectly fit within an 8KB DRAM row buffer to leverage the streaming behavior of DRAM.

Both scene segments and ray buckets are sized cognizant of the DRAM row buffer, to make sure each data stream is stored in main memory as a continuous block, as shown in Figure 5.2.

TPs do not read rays directly from the ray queues. Instead, the stream scheduler fetches entire ray buckets from DRAM and forwards them to the appropriate TM's *ray staging buffer*. Similarly, TPs write rays into ray queues via the stream scheduler, which maintains a small queue of such requests. The stream scheduler drains the queue by writing each ray into the appropriate ray buckets stored in DRAM.

Since rays are written into the ray queues of child scene segments as they exit their parent, the total number of potential write destinations equals the total number of children of all scene segments currently in the working set. Since scene segments can have many children, the number of destinations can be large, and for some of our test scenes it reached about a thousand. Maintaining pointers to the ray queues for such a large number of scene segments – along with the metadata capturing the parent-child and sibling relationships required for queue processing and scheduling – requires approximately 16KB of SRAM.

Each TP fetches individual rays to be processed from one of the ray staging buffers within its TM. There is no deterministic mapping between TPs in a TM and rays within the staging buffer. The staging buffer is sized to store exactly two ray buckets and is split into two halves: while one is being drained by TPs, the stream scheduler fills the other with another ray bucket from DRAM. Each ray stores the address of the node it is traversing.

The TPs use this address to load scene data from the shared L1 cache, which accesses the scene buffer on cache misses. The scene data is fed into TM-wide intersection pipelines for traversal.

After one half of a ray staging buffer runs out of rays, it is swapped for the other half. The stream scheduler polls which ray staging buffers are empty, and attempts to find another ray bucket from the same scene segment to improve the reuse of the L1 data cache which stores the scene data. If there are no more ray buckets for the scene segment being processed currently, the scheduler attempts to find a ray bucket for another scene segment already in the working set. If there are no more ray buckets left for any of the scene segments in the working set, the scheduler must wait for the traversal of a scene segment to complete, before evicting it from the working set and replacing it with another scene segment. To select the next scene segment for inclusion in the working set, the stream scheduler maintains a queue of scene segment indexes to be processed, ordered by the depth-first scene segment traversal.

5.1.2 Scene Buffer

The scene buffer is a global, on-chip memory that holds the scene data for all scene segments currently in the working set. Each scene segment is located contiguously in the buffer's internal storage. The scene buffer hides memory access latency much like the last-level caches in more traditional architectures, but operates in a simpler manner. Unlike a traditional cache, the scene buffer is read-only and is managed entirely by the stream scheduler, rather than responding to individual memory access requests. When a new scene segment is added into the working set, the scene scheduler replaces the data for the evicted scene segment by the new data in the scene buffer. TPs access scene segments only through the L1 data cache backed by the scene buffer, which eliminates random access to DRAM for scene data. The L1 data requests can stall if the requested cache line is yet to arrive into the scene buffer. Recently accessed scene data is retained in the L1 data cache of each TM, which reduces the contention for the global scene buffer.

5.1.3 Hit Record Updater

The hit record updater is a unit that atomically updates the ray hit records stored in DRAM. Duplicated rays share a common hit record, requiring atomic updates whenever

an intersection is found. When a TP finds a primitive hit, it sends an update request to the hit record updater, which maintains a small on-chip queue of requests. If there is an update pending for the given ray index (i.e., a duplicate ray found another hit), the closer of the two hits replaces the update; otherwise, the request is added into the queue. The hit record updater compares the hit distances to the values recorded in the DRAM and updates hit information in DRAM only if the pending hit is closer. This requires a read-modify-write operation, which must wait for the hit record read requests to return from DRAM. As long as the hit record updater queue is not full, TP execution is not blocked. Despite our initial apprehension, we have not found access to the hit record updater to be a bottleneck.

5.2 Performance Evaluation

We use a cycle-accurate simulator SimTRaX [116, 117] to evaluate our dual streaming architecture and we compare our results to STRaTA [75, 76], a state-of-the-art ray tracing specific architecture. The choice of STRaTA for direct comparison is motivated by the fact that it also aims to optimize DRAM accesses (although using a traditional ray tracing paradigm) and thus we can design fair comparisons by simulating similar hardware parameters. We also provide limited comparisons against NVIDIA's OptiX GPU ray tracer [102], Microsoft's DXR ray tracer [144], and Intel's Embree CPU ray tracer [137], running on actual hardware.

In our comparisons, we use no early ray termination for our dual streaming hardware, but we do use early ray termination for STRaTA, OptiX, DXR, and Embree. Therefore, our dual streaming hardware performs substantially more work without the benefits of early ray termination. For an additional comparison, we provide results with STRaTA without early ray termination. We present our test results with the two early ray termination approaches using the dual streaming hardware separately.

5.2.1 Cycle-Accurate Simulation

The SimTRaX simulation infrastructure [116, 117] provides several components that are important for quick exploration of possible architecture designs concurrently with targeted software modifications: combined cycle-accurate and functional simulation capability, flexibility in how functional units are connected, a highly accurate DRAM model, and

integration of the LLVM toolchain [80] for easy ISA extensions and compiling applications written in a high-level language like C++.

The simulation loop iteratively generates clock rise and fall signals across all functional units in the hardware architecture. For every clock cycle, SimTRaX first simulates the computation units, then the units in the memory hierarchy, and finally the main memory. The simulator runs until all hardware threads finish execution, then reports execution statistics and lets the application post-process memory to generate its output which can be used to write an output image. SimTRaX can use source-level debugging symbols provided by LLVM to debug and profile simulated architectures and applications using a wide variety of metrics (e.g., time spent per program source code line, or energy per function call). When simulating memory-bound applications like ray tracing, particularly with thousands of threads, an accurate memory model is of key importance and can have a drastic impact on the results. Furthermore, the energy used to render each frame using the ray tracing depends significantly on how the algorithm uses the memory system, requiring accurate simulation and profiling of the memory system behavior, including DRAM [126].

5.2.2 Hardware Specification

Table 5.1 lists the hardware configurations for both the dual streaming and STRaTA architectures. On-chip cache and SRAM buffer areas are estimated using Cacti 6.5 [93]. Compute resource areas are estimated with synthesized versions of the circuits using Synopsys DesignWare / Design Compiler. We did not fully synthesize the logic circuitry for the memory controllers or the dual streaming scheduler. Instead, the area consumed by the memory controllers is mostly dominated by their buffers and other SRAM components [16]. The dual streaming scheduler is similar to a memory controller in terms of logic circuitry. Therefore, to make conservative area estimates for the memory controller and the stream scheduler, we assume the area of these units is $2\times$ the size of the SRAM components, which we model with Cacti. STRaTA's scheduler is reported as roughly zero area because its scheduling metadata is contained entirely within the ray queue, which is already accounted for. Since STRaTA has a much simpler scheduler, the additional logic circuitry would be negligible in area. We do not include area comparisons for Embree, OptiX or DXR, since STRaTA and the dual streaming hardware are imagined as accelerators,

not a full-system CPU/GPU, and because the 65nm process technology we can simulate is much larger than the current commercial technologies.

In all simulated test results we present, the processor runs at 1GHz and has 4MB of on-chip memory (used differently by STRaTA and dual streaming architectures), 128 thread multiprocessors each with 16 hardware threads (2048 threads total), each with 32 registers and 512B of local scratchpad memory. Note that this is a relatively moderate configuration compared to currently available discrete GPU hardware. Beyond the common parameters, hardware-specific parameters are specified as a result of numerous tests to find an optimal setup for each hardware architecture.

The global scene buffer for the dual streaming architecture is 4MB in size and can store at most 64 BVH treelets, each 64KB in size. Ray buckets are 2KB in size and store up to 63 rays.[1] The sizes of these components are chosen based on our experiments with different configurations. Our earlier tests revealed that we can achieve slightly higher performance for almost all scenes when the scene buffer size is 4MB, as compared to 2MB. However, the optimal scene buffer size depends on the number of TMs. Our tests with different ray buffer sizes provided only slightly elevated performance for most scenes with 2KB, as compared to 1KB.

For the STRaTA results we use BVH treelets of size 32KB, which produced the best performance in our tests. The on-chip memory for STRaTA is split into a 512KB L2 cache and a 4MB ray buffer. The execution units of the original STRaTA multiprocessors dynamically reconfigure into either a ray-box or two ray-triangle intersection pipelines. For a fair comparison to the dual streaming architecture, we generated the STRaTA results using fixed-function pipelines, which have slightly elevated performance.

The pipeline intersecting rays against boxes relies on inverted ray directions [139]. Each TP uses the TM-wide shared division unit to compute this inverse immediately after fetching a ray from the ray staging buffer. The inverse is reused when traversing an individual scene segment. The ray-triangle intersection pipeline relies on Plücker coordinates [114] to delay the division until the ray-triangle intersection is confirmed. Both architectures include a single ray-box (1 cycle initiation interval, 8 cycle latency) and two ray-triangle

[1] Each ray bucket stores a small header, which reduces the total number of rays in a bucket by one - from 64 to 63 rays for a 2KB ray bucket.

(18 cycle initiation interval, 31 cycle latency) pipelines shared by all TPs in each TM.

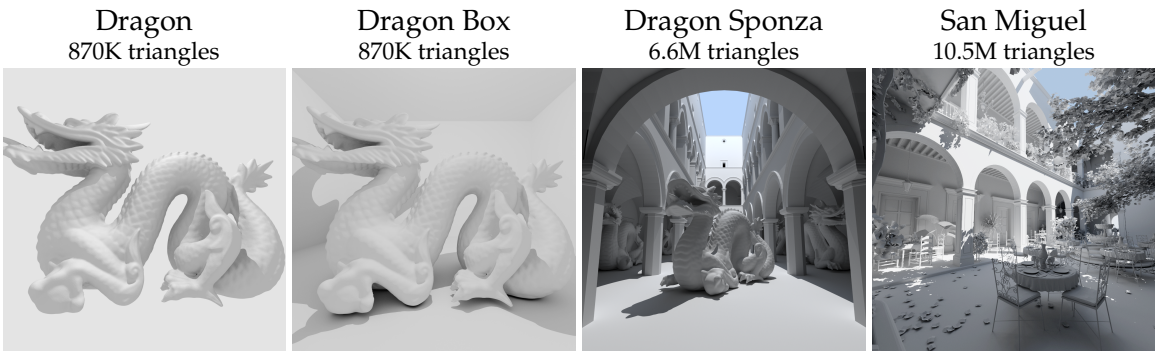
Our evaluation setup includes a GDDR5 DRAM subsystem with 16 32-bit channels, running at an effective clock rate of 8GHz for a total of 512 GB/s maximum bandwidth. The DRAM row buffer is 8KB wide. We rely on a sophisticated memory system simulator, USIMM [23], to accurately model DRAM accesses. Note that using a full memory system simulator is essential for producing reliable results, since the ray tracing performance is tightly coupled with the highly complex behavior of DRAM.

The OptiX (v3.9) results are obtained on an NVIDIA GTX TITAN GPU with 2688 cores running at 876 MHz and 6144 MB GDDR5 memory with 288.4 GB/s peak bandwidth. The Microsoft DXR results are obtained using the NVIDIA path tracing sample [144] running on NVIDIA RTX 2080 GPU with 2688 cores running at 1.8 GHz and 8192 MB GDDR6 memory with 448 GB/s peak bandwidth. The Embree (v2.10) results are obtained with its example path tracer (v2.3.2) running on an Intel Core i7-5960X processor with 20 MB L3 cache and 8 cores (16 threads) over-clocked to 4.6GHz.

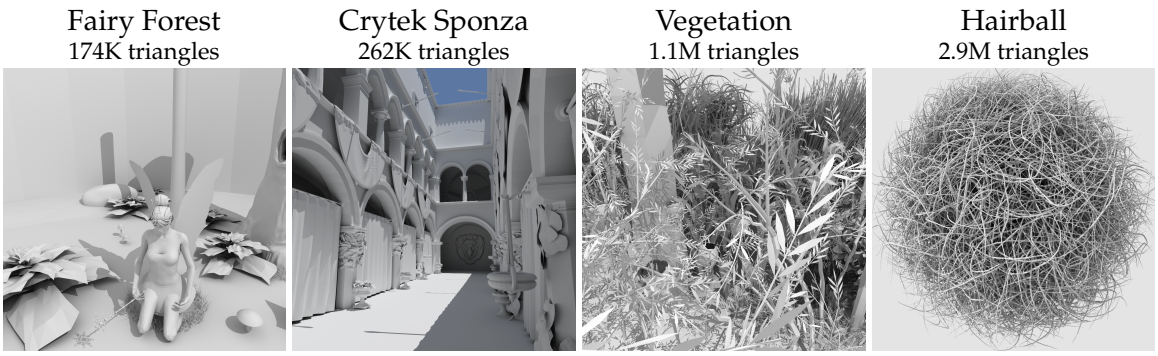
5.2.3 Test Scenes

We use eight test scenes, shown in Figure 5.3, to represent a range of complexities and scene sizes. They are rendered using path tracing [64] with five bounces, producing a highly incoherent collection of secondary rays, which is both challenging for high-performance ray tracing and typical for realistic rendering. Each image is rendered at the resolution of 1024×1024 pixels, resulting in at most 10.5 million total rays, including both primary and secondary. Dual streaming architecture traces at most two million rays (and their duplicates) per wavefront, while STRaTA traces 80,000 rays, many potentially at different depths. We use a simple Lambertian diffuse material on all scenes, so that the results are not skewed by expensive shading operations.

Some scenes are chosen to present the performance of our dual streaming hardware in atypical cases that it is not designed to optimize. The first two, Fairy Forest and Crytek Sponza, are small scenes that can mostly fit in the on-chip memory. Therefore, the improvements that the dual streaming architecture introduces for better DRAM accesses provide no benefits. The other two scenes, Vegetation and Hairball, are not as small, but have extreme depth complexity, where early ray termination, which is disabled for the



(a) Benchmark scenes



(b) Small scenes **(c)** Scenes with high depth complexity

Figure 5.3: Scenes used for all performance tests and comparisons.

dual streaming architecture, can provide tremendous savings in terms of ray duplication and traversal.

STRaTA is an architecture that assumes a slightly enhanced physical memory architecture by adding on-chip ray queues to the memory hierarchy. Because STRaTA stores rays just in on-chip buffers, it can process only a limited number of rays simultaneously, restricting shader complexity and treelet effectiveness. Nonetheless, because of its focus on optimizing the DRAM accesses for at least the scene data, we choose STRaTA as our primary comparison.

5.2.4 Overall Performance

Table 5.2 provides detailed test results. The results for our dual streaming hardware and STRaTA are obtained from hardware simulations, so they include detailed information. For the dual streaming architecture, we also report a breakdown of the memory traffic and average ray duplication rates, which measure the ratio of the total number of rays enqueued into any scene segment to the number of unique rays generated. The OptiX,

		Benchmark				Small		High Depth	
		Dragon	Dragon Box	Dragon Sponza	San Miguel	Fairy Forest	Crytek Sponza	Vegeta-tion	Hairball
OptiX	Render Time (ms/frame)	24.50	92.97	151.83	397.47	84.05	140.11	266.92	229.79
	Rays Traced per sec (M)	135.0	112.6	66.0	24.4	78.9	72.0	26.1	27.5
DXR	Render Time (ms/frame)	3.57	16.89	24.90	37.98	11.13	16.92	22.51	11.21
	Rays Traced per sec (M)	584.1	744.6	477.6	288.2	545.8	698.5	289.0	274.2
Embree	Render Time (ms/frame)	38.13	103.81	118.05	143.64	83.6	150.63	178.99	113.32
	Rays Traced per sec (M)	99.5	89.1	70.6	50.5	96.1	62.0	41.7	45.3
STRaTA	Render Time (ms/frame)	*23.12*	*91.27*	*70.98*	*125.51*	**16.1**	**39.0**	**48.23**	**36.2**
	Rays Traced per sec (M)	*89.7*	*128.9*	*135.4*	*72.6*	**365.6**	**233.3**	**121.4**	**111.2**
	DRAM Energy (J)	*2.34* (55%)	*10.17* (56%)	*5.32* (46%)	*15.08* (60%)	**0.87** (32%)	**2.26** (32%)	*5.38* (52%)	*4.61* (59%)
	On-Chip Memory Energy (J)	*1.84* (43%)	*7.67* (42%)	*6.03* (52%)	*9.76* (39%)	*1.76* (65%)	*4.55* (65%)	**4.70** (46%)	**3.02** (39%)
	Compute Energy (J)	*0.08* (2%)	*0.29* (2%)	*0.21* (2%)	*0.33* (1%)	*0.09* (3%)	*0.24* (3%)	*0.23* (2%)	*0.14* (2%)
	Avg. Bandwidth (GB/s)	219.33	266.65	137.48	219.34	99.01	101.95	229.59	254.53
	$ Lines Transferred (M)	79.2	380.1	152.5	430.1	24.91	62.14	173	144
STRaTA no early termination	Render Time (ms/frame)	*47.12*	*154.08*	*117.97*	*363.16*	*21.64*	*63.56*	*115.30*	*247.75*
	Rays Traced per sec (M)	*44.0*	*76.3*	*81.5*	*25.1*	*272.2*	*143.3*	*50.8*	*16.2*
	DRAM Energy (J)	*5.63* (66%)	*21.61* (67%)	*11.49* (60%)	*48.50* (75%)	*1.36* (40%)	*4.46* (41%)	*14.48* (61%)	*35.47* (78%)
	On-Chip Memory Energy (J)	*2.72* (32%)	*9.89* (31%)	*7.31* (38%)	*15.3* (24%)	*1.97* (57%)	*6.07* (56%)	*8.78* (37%)	*9.82* (21%)
	Compute Energy (J)	*0.15* (2%)	*0.53* (2%)	*0.38* (2%)	*0.76* (1%)	*0.12* (3%)	*0.35* (3%)	*0.46* (2%)	*0.49* (1%)
	Avg. Bandwidth (GB/s)	251.00	327.41	185.16	221.77	125.52	137.07	280.86	245.41
	$ Lines Transferred (M)	184.8	788.2	341.3	1258	42.4	136.1	506.0	950.0
Dual Streaming	Render Time (ms/frame)	**18.08**	**66.3**	**40.93**	**79.61**	*17.05*	*44.6*	*68.56*	*63.27*
	Rays Traced per sec (M)	**114.8**	**177.4**	**234.8**	**114.6**	*345.6*	*204.1*	*85.4*	*63.5*
	DRAM Energy (J)	**1.15** (42%)	**4.66** (40%)	**4.47** (57%)	**8.12** (50%)	*1.61* (53%)	*4.51* (50%)	**4.41** (41%)	**4.56** (43%)
	On-Chip Memory Energy (J)	**1.52** (55%)	**6.54** (57%)	**3.10** (40%)	**7.63** (47%)	**1.30** (43%)	**4.14** (46%)	*5.96* (56%)	*5.75* (54%)
	Compute Energy (J)	*0.09* (3%)	*0.36* (3%)	*0.23* (3%)	*0.53* (3%)	*0.10* (3%)	*0.32* (4%)	*0.37* (3%)	*0.33* (3%)
	Avg. Bandwidth (GB/s)	140.21	114.98	271.30	255.61	230.35	237.40	142.75	142.88
	$ Lines Transferred (M)	39.6	119.2	173.5	317.9	58.7	165.4	152.9	141.3
	Ray Stream $ Lines (M)	11.92 (30%)	45.81 (38%)	43.94 (25%)	146.34 (46%)	18.4 (31%)	80.56 (49%)	94.03 (61%)	76.22 (54%)
	Scene Stream $ Lines (M)	7.54 (19%)	8.0 (7%)	54.31 (31%)	72.27 (23%)	1.50 (3%)	2.26 (1%)	8.20 (5%)	19.18 (14%)
	Shading $ Lines (M)	17.43 (44%)	42.38 (36%)	45.00 (26%)	46.22 (15%)	30.53 (52%)	45.42 (27%)	32.46 (21%)	26.99 (19%)
	Hit Record $ Lines (M)	2.74 (7%)	22.97 (19%)	31.04 (18%)	53.93 (17%)	8.33 (14%)	37.21 (22%)	18.28 (12%)	18.89 (13%)
	Ray Duplication	4.55	3.14	4.18	15.19	3.00	8.55	15.15	16.02

Table 5.2: The performance results comparing OptiX, DXR, Embree, STRaTA and our dual streaming architecture. Note $ means cache, M means millions. Values highlighted in **red** indicate the best performance for that metric, excluding DXR results which are provided as a point of reference.

DXR and Embree results only include render time and rays traced per second, measured on actual hardware.

Figure 5.4 compares the render times per frame between the dual streaming hardware, STRaTA, OptiX, DXR and Embree. Figure 5.5 compares DRAM energy per frame between the dual streaming architecture and STRaTA. Notice that for all benchmark scenes our dual streaming hardware provides substantially superior performance as compared to STRaTA, and the difference is more substantial in larger scenes. It achieves lower render times (up to almost twice as fast in large scenes) and consumes less DRAM energy (about half of STRaTA in some scenes).

For the small scenes, Fairy Forest and Crytek Sponza, which STRaTA can mostly fit

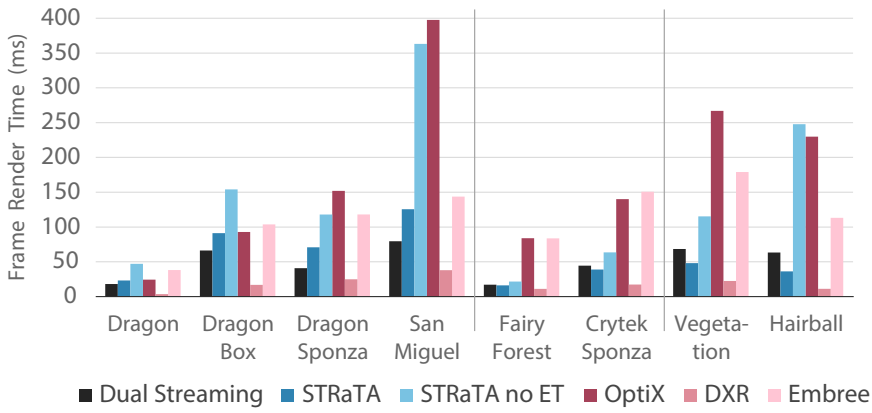


Figure 5.4: Render time per frame. ET means early ray termination. Lower is better.

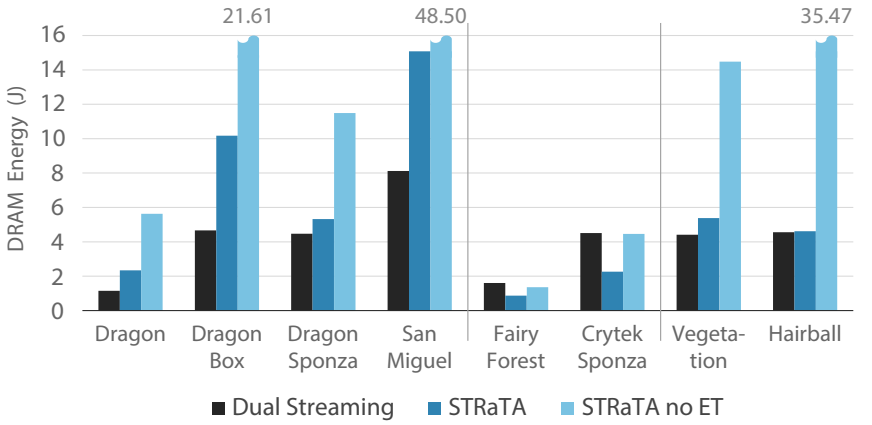


Figure 5.5: DRAM energy per frame. ET means early ray termination. Lower is better.

in the on-chip memory, our dual streaming hardware implementation can still achieve a similar render time, but the additional burden of streaming rays costs extra DRAM energy.

On the other hand, the lack of early ray termination in our implementation of the dual streaming architecture hurts the render time in scenes with high depth complexity, Hairball and Vegetation. This is due to the extra work that our dual streaming implementation endures (to find potentially all hits) and STRaTA can avoid via early ray termination (to find the first hit). This extra work can be clearly seen in the elevated ray-triangle intersection counts, shown in Figure 5.6, and the rates of ray duplication, shown in Figure 5.7. However, in the San Miguel scene, even though it also has substantial depth complexity causing several times more ray-triangle intersections, the savings of the dual streaming architecture more than make up for the extra computation. This result confirms that the dual streaming architecture has the potential to provide more savings for larger scenes.

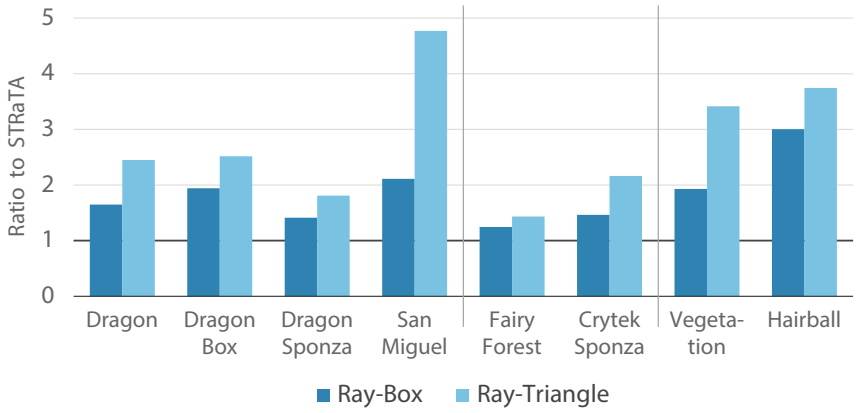


Figure 5.6: Number of ray-box and ray-triangle tests performed by our dual streaming hardware per frame, shown as a ratio relative to STRaTA. Lower is better.

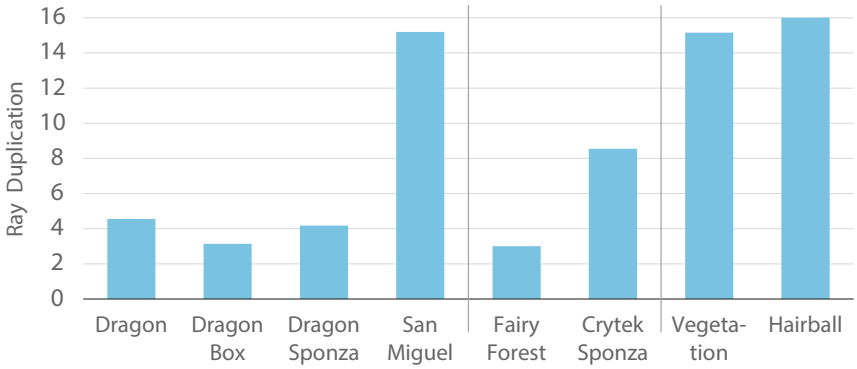


Figure 5.7: Dual streaming architecture ray duplication. Lower is better.

Even though the dual streaming architecture requires more intersection tests than STRaTA, all compute makes up around $3-5\%$ of the total energy spent per frame, as can be seen in Table 5.2. Thus, a five-fold increase in the number of intersection tests generates a tiny increase in compute energy. The remaining $95-97\%$ of the frame energy is spent by the on-chip memories and DRAM, which is by far the single largest consumer.

The breakdown of the memory traffic for the dual streaming architecture and STRaTA are shown in Figure 5.8. Notice that the scene stream only takes up a relatively small portion, even in large scenes. Although the total traffic generated by the dual streaming architecture is smaller than that of STRaTA for all but Dragon Sponza and small scenes, the dual streaming architecture substantially reduces scene the traffic for all scenes. Comparing Dragon Sponza to San Miguel, which is almost twice the size, we can see that both scenes have similar scene stream costs (Figure 5.8). However, the ray stream can contribute

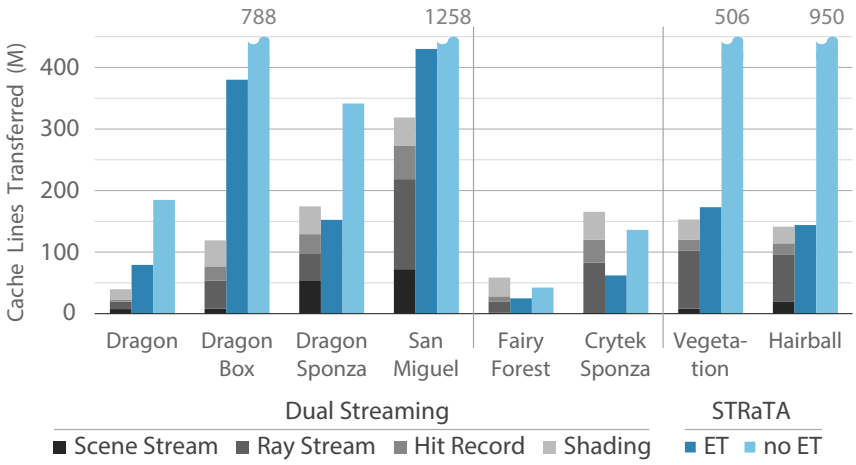


Figure 5.8: Memory traffic generated by the dual streaming architecture and STRaTA (with and without early ray termination, ET), in millions of cache lines (64B ea). Lower is better.

a substantial portion of the memory traffic in all scenes. In the case of San Miguel, ray duplications not only cause extra computation but also a substantial amount of ray stream traffic and extra hit record updates, although it still renders almost twice as fast with the dual streaming architecture and consumes almost half the DRAM energy, as compared to STRaTA. Note that the magnitude of the memory traffic is related to, but does not directly correlate with, the DRAM energy or performance which are also influenced by the order in which the memory requests are generated.

5.2.5 Early Ray Termination

In this section, we provide our test results that evaluate the two early ray termination approaches for the dual streaming architecture discussed in Section 4.2.3. Figure 5.9 compares the render times achieved by the pre-test and the post-test early ray termination approaches to the render time with no early ray termination. Notice that the pre-test can provide some improvement, especially for scenes with high depth complexity. The post-test, however, is less effective in our test scenes. Our experiments also revealed that combining both tests does not improve on using the pre-test alone, and thus the combination is not shown. We attribute the slightly better performance achieved using the pre-test (as compared to the post-test) to its ability to skip ray traversal through the current scene segment, while the post-test is only helpful in preventing unnecessary ray duplication when a closer hit has already been found.

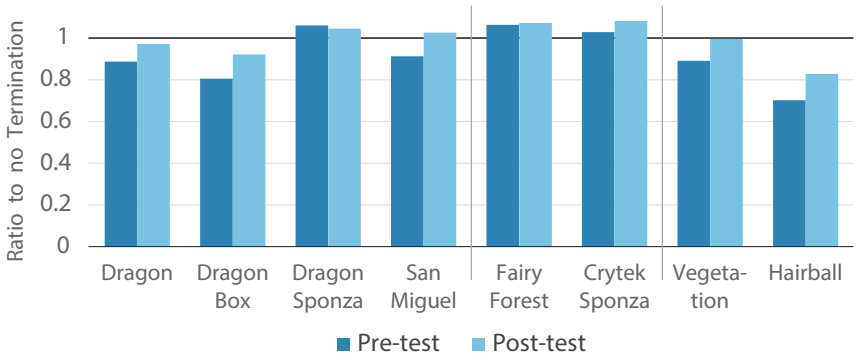


Figure 5.9: Effect of early ray termination on frame render times (ms/frame) of the dual streaming architecture shown as a ratio to the render time using no early ray termination. Lower is better.

It is also important to note that in some scenes our early ray termination strategies can even impact the overall render time and DRAM energy negatively. This is not surprising, since early ray termination requires random memory accesses that can cause TPs to stall. As such, optimizing early ray termination in the dual streaming architecture is left for future work.

We also compare the effect of early ray termination on STRaTA, shown in Table 5.2. Disabling it can incur a significant increase in frame times of up to 3× for San Miguel and almost 7× for Hairball. The total number of cache line transfers from DRAM at least doubles. These increases are expected because STRaTA reloads scene data as rays traverse back to parent treelets. Note that compared to STRaTA without early ray termination, the dual streaming architecture has lower frame render times for all scenes. For all but the small scenes, the dual streaming architecture uses less DRAM energy and the number of cache lines transferred is also significantly smaller. Detailed exploration of early ray termination within the dual streaming architecture is left for future work.

5.2.6 Scene Segment Schedulers

In this section, we provide our test results evaluating the four approaches to schedule scene segments into the working set, described in Section 4.2.1. We use the default configuration of the dual streaming architecture described in Section 5.2.2. Figure 5.10 compares the performance of scene segment schedulers from the perspective of the time to render a single image frame. The data is normalized to the performance of the opportunistic 1st

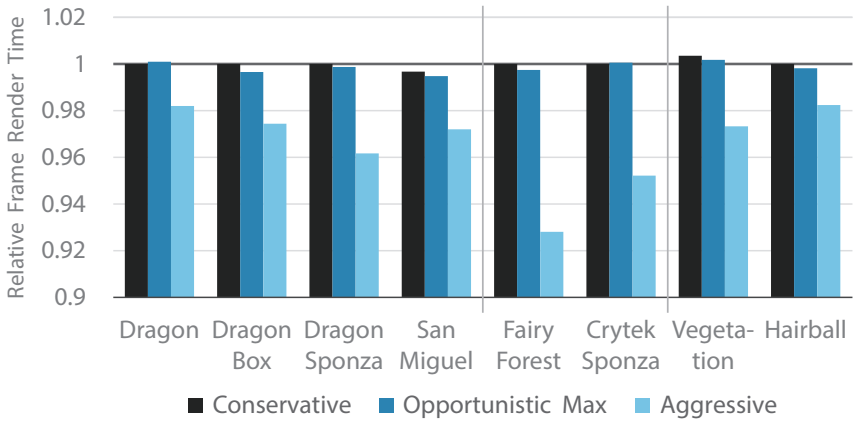


Figure 5.10: Frame render times of different scene segment schedulers relative to the *opportunistic 1st* scheduler. Lower values are better. Note the narrow data range within the plot.

scheduler, which is used by all of our other tests.

For each scene, the frame render times are very similar between the conservative and the opportunistic schedulers, within 0.5% of the opportunistic 1st frame times. Rendering the scenes using the aggressive scheduler leads to 2 - 7% lower render times (3.4% on average). Compared to the other schedulers, the aggressive scheduler succeeds in keeping more scene segments in the working set per cycle on average: 7 - 17% for all but the small scenes (39% more for Fairy Forest and 51% more for Crytek Sponza). This translates directly into a reduction in TP stalls due to waiting for ray buckets to be fetched from DRAM, and thus faster rendering speeds. A larger number of scene segments in the working set potentially enables a larger pool of rays to be distributed for processing at any given time.

The DRAM traffic and energy use are not shown because the difference between schedulers results in at most 0.7% and 1.3% respectively across all scenes.

5.2.7 Architectural Design Space Exploration

In this section, we evaluate the sensitivity of the dual streaming architecture performance to different configuration parameters. Unless specified, each test isolates a specific parameter which modifies the default configuration specified in Section 5.2.2. We evaluate the performance of each configuration by considering the render time, the total DRAM traffic and the DRAM energy used to generate a single image frame.

5.2.7.1 Number of TPs per TM

Let's consider the impact on performance when varying the number of TPs making up each TM, shown in Figure 5.11. The chip is comprised of 128 TMs. Note that the increases in the number of TPs per TM do not increase the sizes or the number of resources shared within each TM like the L1 data cache.

When the number of TPs per TM doubles from 8 to 16, the rendering performance measured in frames per second improves by 1.62× on average. Doubling again to 32 TPs per TM achieves a modest further rendering performance increase to 1.86× on average compared to 8 TPs per TM. Further increases in the number of TPs per TM do not improve the rendering frame rate mainly because the global hit record updater unit becomes over-subscribed, since it can service at most 16 simultaneous requests per cycle and maintains a small queue of requests storing at most 64 updates. A lesser factor is the ray bucket size: each bucket holds at most 63 rays which, even when full, may not provide enough computational work for all TPs within a single TM.

The DRAM energy decreases by $9 - 24\%$ when the number of TPs per TM increases from 8 to 16. This decrease can be attributed to an increase in the DRAM row buffer hit rate with more DRAM accesses reusing the same row buffer, thus requiring fewer new rows to be fetched and use less energy overall. As the number of TPs per TM continues to grow, the row buffer hit rate decreases somewhat thus increasing the total DRAM energy.

We select 16 TPs per TM for the default configuration of the dual streaming architecture because it provides high render frame rates while keeping the used DRAM energy low across all scenes.

5.2.7.2 Number of TMs in a Chip

Let's consider the impact on performance when varying the total number of TMs making up the dual streaming chip, shown in Figure 5.12. Each TM has 16 thread processors. Increasing the number of TMs making up the chip increases both the available compute in terms of the total number of TPs and the total size of on-chip memories shared between all TPs in a TM, like the L1 data cache and ray buffers.

As the number of TMs (and thus the total number of TPs) increases, the frame rendering performance measured in frames per second scales almost linearly up to 128 TMs. On

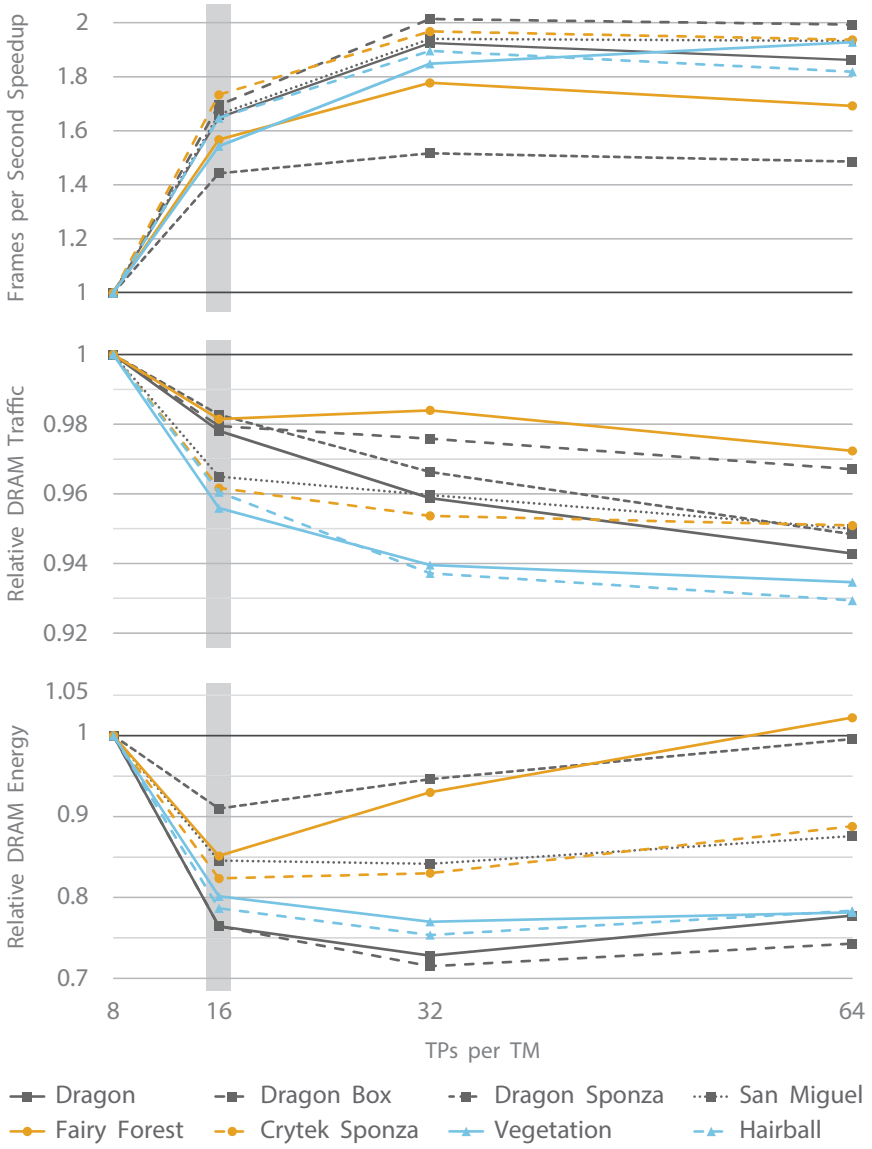


Figure 5.11: Performance of the dual streaming chip with 128 TMs while varying the number of TPs in each TM. The vertical gray bar highlights the default configuration with 16 TPs per TM. Lines are colored based on the scene type. Higher values are better in the top plot, while lower values are better in the other plots. Note the narrow data ranges within the plots.

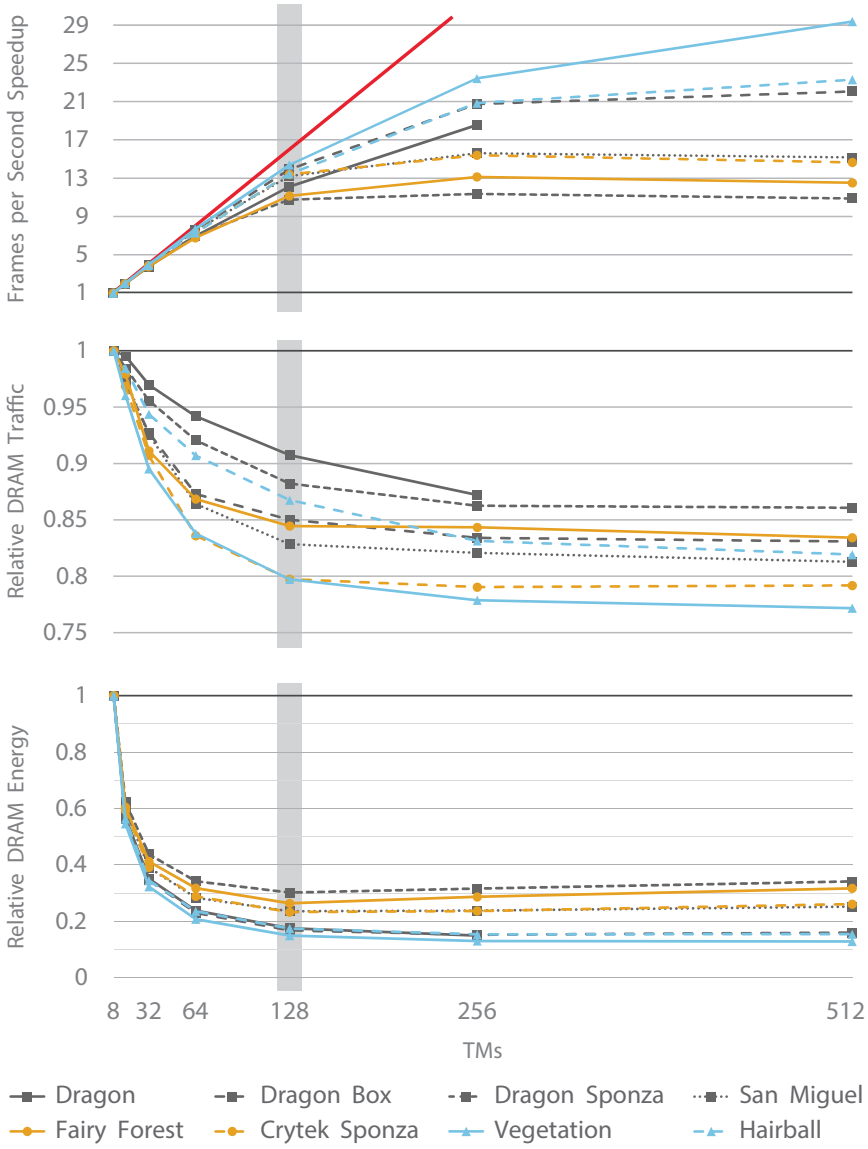


Figure 5.12: Performance of the dual streaming chip when varying the number of TMs, each containing 16 TPs. The vertical gray bar highlights the default configuration with 128 TMs. Lines are colored based on the scene type. The red line in the top plot denotes perfect scaling following the number of TMs. Higher values are better in the top plot, while lower values are better in other plots. Note the narrow data ranges within the plots.

average across all scenes, for a chip with 128 TMs, the number of frames per second increases 12.8× that of 8 TMs, which is 80% efficiency relative to the perfect linear scaling (16×). The rendering speed stops scaling as the number of TMs increases beyond 128 because the global hit record update unit becomes over-subscribed.

DRAM traffic is shared well with the increase in the number of TMs. Comparing the traffic between 512 and 8 TMs per chip, a 64× increase, results in a reduction in DRAM traffic of at most 23% (18% on average). The major contributor to this drop is a significant reduction in the number of cache lines transferred for ray writes as the number of TMs increases. The remaining DRAM traffic (scene and ray stream reads) remain fairly constant.

The amount of energy used by DRAM to render a single frame drops significantly as the number of TMs increases from 8 per chip. The configuration with 128 TMs uses on average 21% of the DRAM energy used by the configuration with 8 TMs. The large drop in energy corresponds to a huge drop in the background DRAM energy spent on maintaining open row buffers without servicing any reads or writes. This background energy takes up $74 - 83\%$ of the total for the 8 TM configuration and $25 - 40\%$ of the total for the 128 TM configuration. The background energy is correlated with the length of time taken to render a frame, thus much longer render times correspond to larger background energies.

We select 128 TMs per chip as the default configuration of the dual streaming architecture because it provides high render frame rate while reducing the DRAM traffic and energy significantly.

5.2.7.3 Hit Record Updater

To evaluate the impact the chip-wide hit record updater has on the dual streaming architecture, we consider two parameters: the hit record updater issue width and the internal queue size that stores the individual hit record updates while the data is fetched from DRAM. These tests rely on the default configuration with 128 TMs each with 16 TPs for a total of 2048 threads. A larger number of threads puts an increased pressure on the hit record updater unit, requiring increasing both its issue width and its internal queue size.

Increasing the issue width of the hit record updater from 16 to 64 has a negligible (<

1%) effect on the frame render time, DRAM traffic and DRAM energy across the tested scenes.

Increasing the internal queue size from 64 to 1024 results in at most 3% improvement in frame render times across all tested scenes but Dragon Sponza which achieves about 12% improvement using the queue holding at least 256 items. For this scene, hit record updater stalls waiting for data to arrive from DRAM rather than DRAM memory controller being full of requests. The performance does not improve much with the increase in queue size because the majority of hardware stalls occur while waiting for either ray or scene data. Increasing the queue size reduces the DRAM traffic and energy by at most 2% across all scenes.

We configure the hit record updater with issue width of 16, the same as the number of TPs in each TM, and the internal queue sized at 64 entries. Even though increasing this to 128 or 256 would improve the frame render times, the improvement is limited for tested scenes other than Dragon Sponza. Most importantly, the hit record updater is not a significant bottleneck for the dual streaming architecture configured with the 2048 threads.

5.2.7.4 Ray Bucket Size

Let's consider the impact on performance when varying the ray bucket sizes, shown in Figure 5.13. Smaller ray buckets can help assign more TMs to process rays from a given queue; however, smaller ray buckets store fewer rays each and can result in under-utilizing compute resources in a single TM because some TPs could stall without rays to process. Larger ray buckets, on the other hand, avoid work starvation within a single TM, but at the potential cost of poor work distribution across TMs.

The frame render times show these effects. Ray buckets 2KB in size provide the optimal rendering performance across all scenes except for Dragon. Using bucket sizes other than 2KB results in the frame render times increasing up to 8%. The Dragon scene however performs best when using smaller ray bucket sizes because for this particular scene there are fewer rays as ray depths increase. The dragon statue has few concave sections and is placed in empty space, both of which result in few interreflections and thus decreasing ray counts as depth increases. In fact, the ray count decreases with depth faster for the Dragon scene than for any other scene tested, as shown in Figure 5.14.

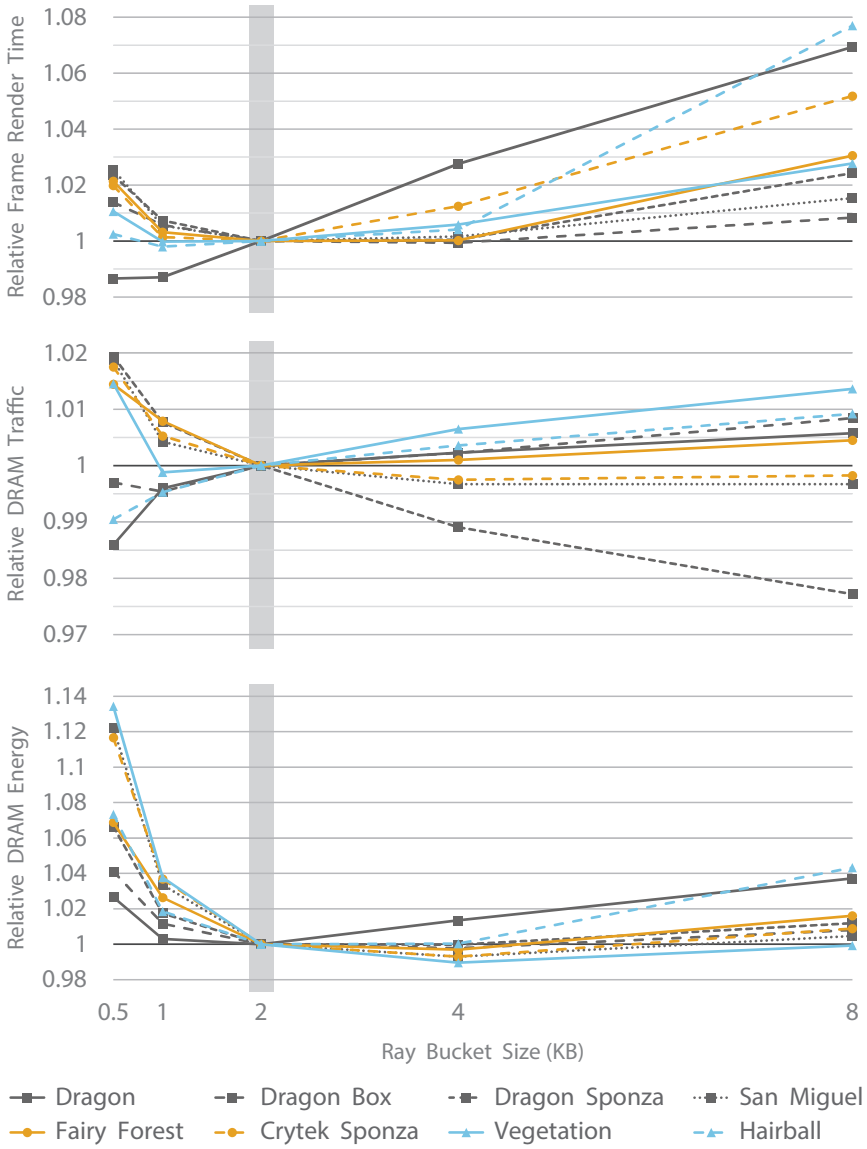


Figure 5.13: Performance of the dual streaming chip with different sizes of ray buckets. Data for each scene is normalized to the performance of the 2KB ray buckets. The vertical gray bar highlights the default configuration. Lines are colored based on the scene type. Lower values are better. Note the narrow data ranges within the plots.

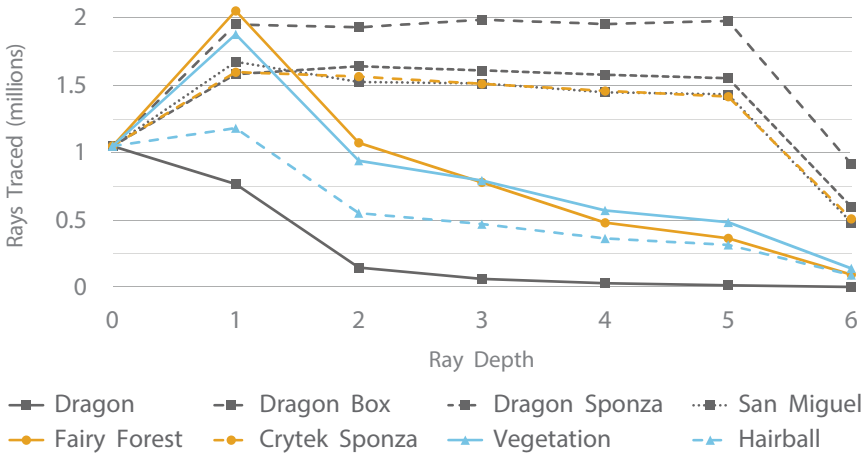


Figure 5.14: Total number of rays traced at different depths for each scene. The count at depth 0 includes only the primary rays originating at the camera. Only the shadow rays are traced at depth 6.

Using ray buckets smaller than 2KB in size can increase the total amount of DRAM energy used to render the frame by up to 13.4% when compared to 2KB ray buckets. Using 4KB ray buckets uses almost the same amount of DRAM energy as 2KB ray bucks (a reduction of less than 0.2% on average across all scenes).

We select 2KB size for the ray buckets as the default configuration of the dual streaming architecture because it provides high render frame rates while reducing the DRAM traffic and energy.

5.2.7.5 Image Size

Let's consider the impact on performance when varying the output image resolution, as shown in Figure 5.15. The chip configuration remains constant, with 128 TMs each with 16 TPs for a total of 2048 threads. As the image resolution grows, the number of pixels and thus rays to be traced, grows quadratically.

As shown in Figure 5.15, the frame render times increase similarly to the increases in the number of pixels. Each scene has a different scaling factor that still varies slightly depending on the output image resolution. The growth profile for the frame render times suggests that the dual streaming performance is limited by the ray stream. The DRAM traffic and energy also scale proportionally to the image resolution.

We select the image resolution of 1024×1024 because it is large enough to represent typical use-cases and generates a large set of random rays without incurring extremely

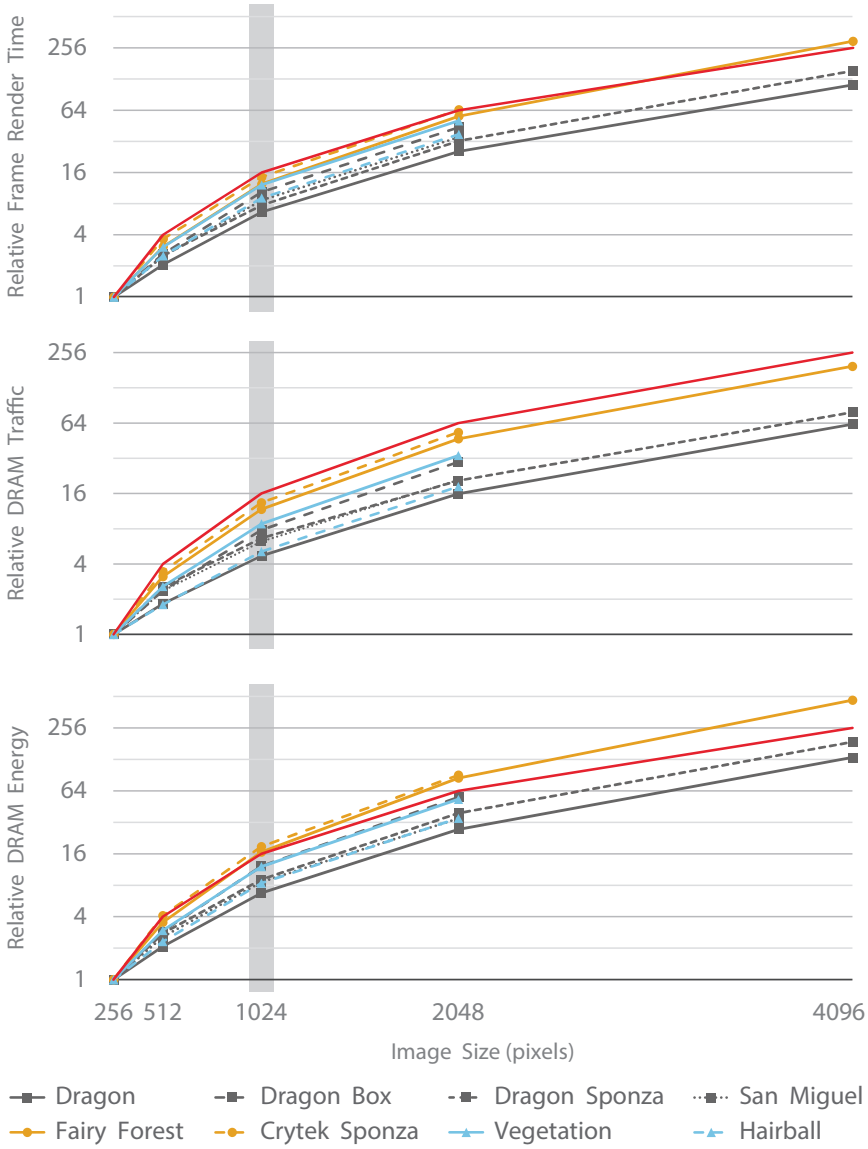


Figure 5.15: Performance of the dual streaming chip with 128 TMs while varying the resolution of out the output image, shown by a single dimension. The vertical gray bar highlights the default configuration with 1024×1024 image resolution. Lines are colored based on the scene type. The red lines show perfect linear scaling. Lower values are better. Note the logarithmic scale on the y-axis within the plots. The missing data generated too many rays for indexes used by the implementation of the dual streaming architecture.

long simulation times.

5.2.7.6 Scene Segment Size

Let's consider the impact on performance when varying the scene segment size, as shown in Figure 5.16. The scene buffer size remains constant at 4MB across all scene segment sizes, thus the scene buffer would contain fewer scene segments as the segment size increases. Larger scene segments require a fewer number to represent an entire scene, which leads to lower ray duplication rate and thus lower DRAM traffic from a reduction in the ray stream size. On the other hand, the amount of on-chip memory assigned to the scene buffer is limited by the chip area, which limits the working set size and thus how many scene segments can be processed simultaneously.

Scene segments 64KB in size result in reasonable frame render times across all tested scenes. Compared to 64KB scene segments, using 32KB scene segments increases the frame render time by 3%, while 128KB scene segments decrease the frame render time by 3% on average across all scenes.

Using the larger scene segment sizes reduces the DRAM traffic. Compared to 64KB scene segments, the total traffic is larger by 8% on average across all scenes when using 32KB scene segments. Using 128KB scene segments decreases the total DRAM traffic by 5% on average. The reduction in DRAM traffic for larger scene segment sizes is driven entirely by the reduction in the ray stream traffic because the ray duplication is lower. The reduction in ray duplication occurs because the scene can be represented with fewer scene segments, thus any given ray can enqueue into fewer number of ray queues. Compared to 64KB scene segments, smaller scene segments (32KB) increase the ray duplication by 20% while larger scene segments (128KB) decrease the ray duplication by 9% on average across all scenes.

Using the larger scene segment sizes reduces the DRAM energy used to render each frame. Compared to 64KB scene segments, using the smaller scene segments (32KB) increases the DRAM energy usage by 5% while using the larger scene segments (128KB) decreases the DRAM energy usage by 6% on average across all scenes. Unlike the other scenes, Dragon uses 3% less DRAM energy when the scene segment size is 16KB relative to 64KB. Most of the energy savings is due to the lower background power because the

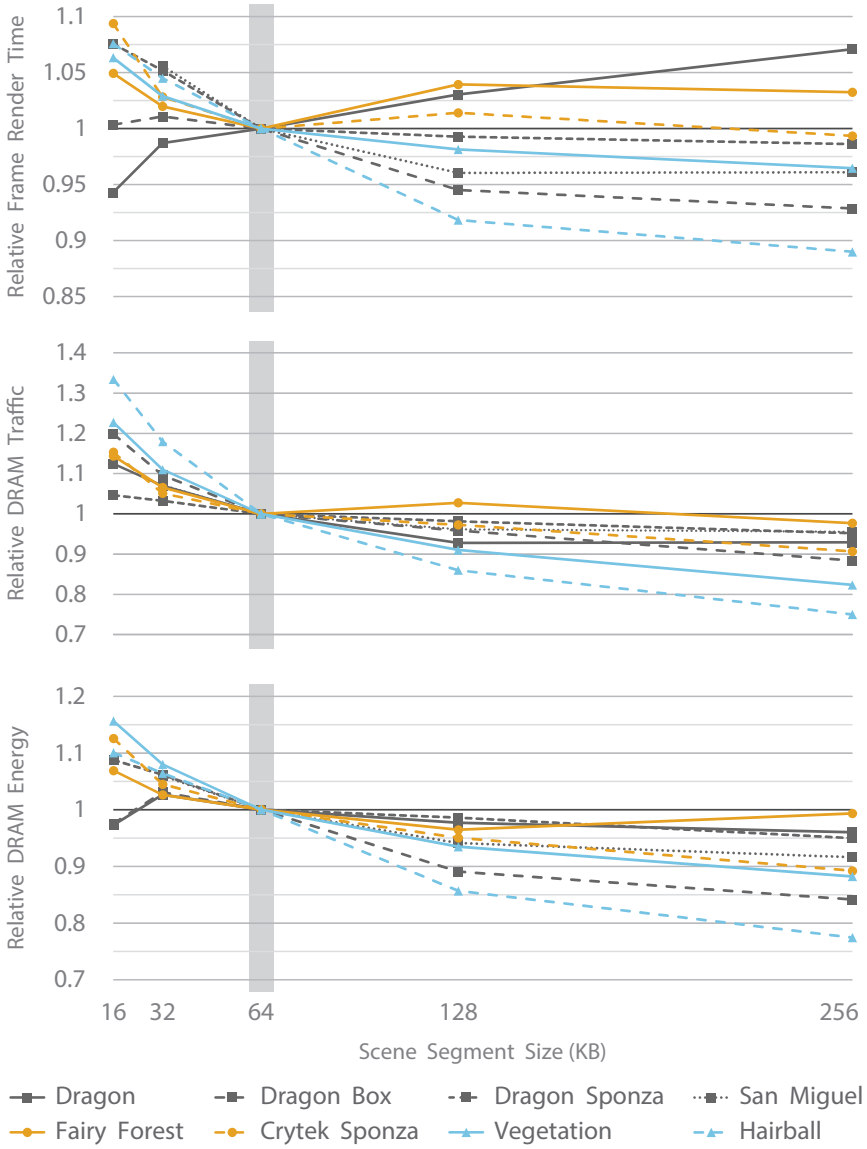


Figure 5.16: Performance of the dual streaming chip with different sizes of scene segments. Data for each scene is normalized to the performance of the 64KB scene segments. The vertical gray bar highlights the default configuration. Lines are colored based on the scene type. Lower values are better. San Miguel data for 16KB size is missing because the configuration generated too many segments for 16-bit address representation. Note the narrow data ranges within the plots.

frame rendering time is much lower for this scene when using 16KB scene segments.

Overall, scene segments 64KB in size result in reasonably low frame render times across all tested scenes, although increasing the size to 128KB could reduce the DRAM energy and traffic but with an increase in frame render times for some scenes (Fairy Forest, Crytek Sponza, and Dragon). We choose 64KB as the default scene segment size since on balance it provides better frame rendering times while conserving DRAM traffic and energy.

5.2.7.7 Scene Buffer Size

Let's consider the impact on performance when varying the size of the global scene buffer, as shown in Figure 5.17. All tested configurations use scene segments 64KB in size and the chip with 128 TMs each with 16 TPs (2048 total threads). A larger scene buffer contains more scene segments, which increases the number of scene segments that can be traversed in parallel, potentially allowing more rays to be traced simultaneously.

The frame render times are essentially the lowest when the scene buffer is 4MB in size. Decreasing the buffer size increases the frame render times by 2% for 2MB and 12% for 1MB on average across all scenes. Increasing the buffer size to 8MB has a negligible effect on the frame render time overall. The buffer size has a very small effect on the frame render times (less than 0.5%) for small scenes (Fairy Forest and Crytek Sponza) because they mostly fit into the on-chip scene buffer. The frame render time for the Dragon scene becomes much longer as the scene buffer size decreases because TPs stall waiting for rays, since there are not enough ray buckets available for simultaneous processing. The tested configuration uses 128 TMs. Thus a 1MB scene buffer with 16 different scene segments can offer at most 16 different ray queues to process simultaneously. When each ray queue contains a few ray buckets for processing, which happens at higher ray depths for the Dragon scene, the chip's rendering frame rate becomes limited by the total number of scene segments that can be processed in parallel.

Based on our experiments, there is not a specific global scene buffer size that clearly reduces the total DRAM traffic across all scenes. Most of the data is within 1% of the 4MB performance.

We choose 4MB as the size of the global scene buffer because it provides the lowest frame render times and DRAM energy across all scenes.

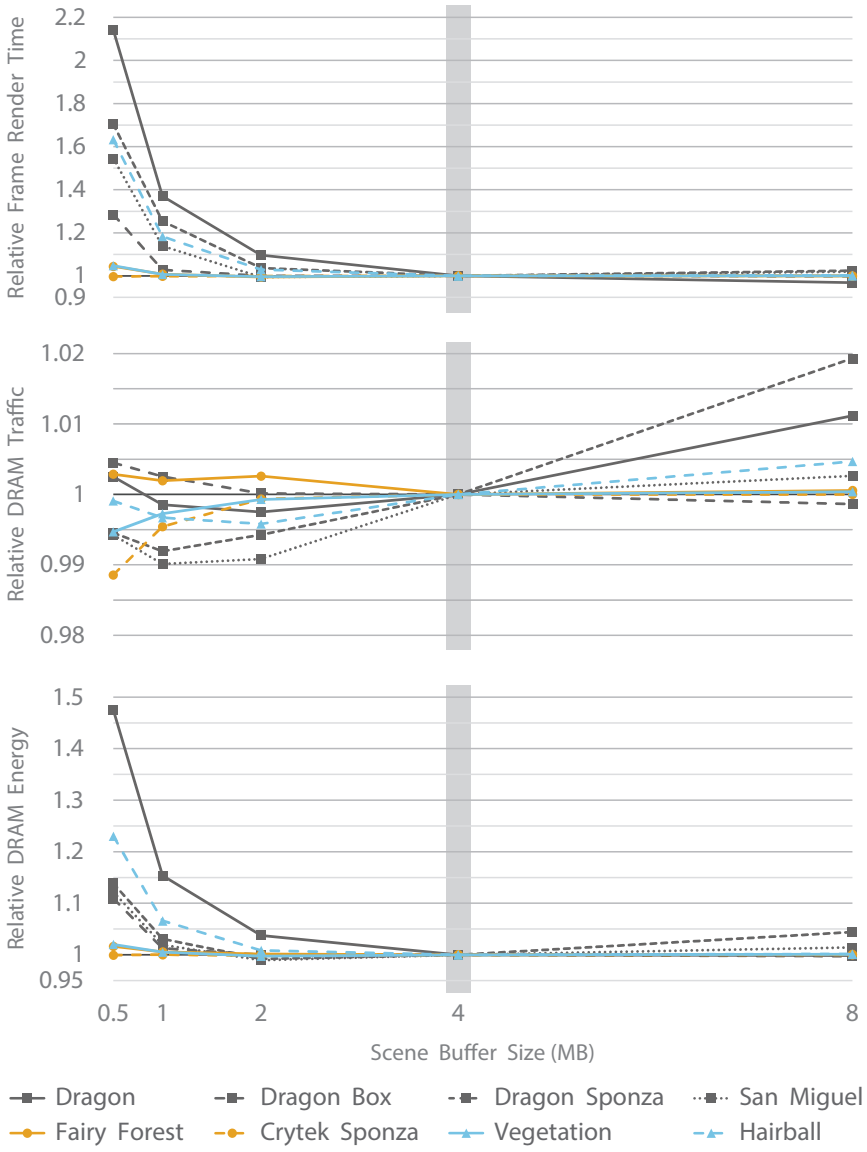


Figure 5.17: Performance of the dual streaming chip while varying the scene buffer size. Data for each scene is normalized to the performance of the 4MB scene buffer size. Lower values are better. Note the narrow data ranges within the plots.

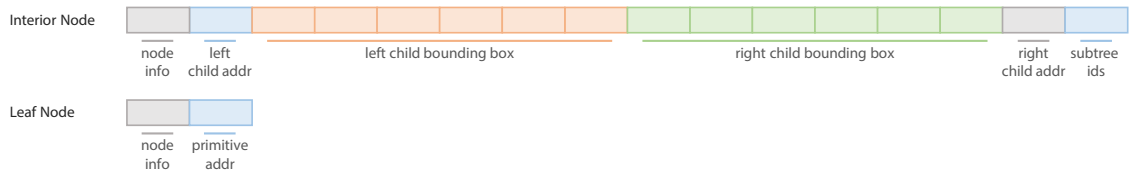
5.3 Comparison to STRaTA

Like STRaTA our hardware implementation uses the SPMD paradigm, utilizes fixed-function ray intersection pipelines, and relies on BVH treelets as a means of accessing scene data. However, beyond these similarities in basic components, which would be present in any parallel hardware, our dual streaming hardware design bears little resemblance to STRaTA. First of all, the dual streaming architecture fundamentally alters how data flows through the processor and how it is accessed: perfect prefetching for the dual stream-

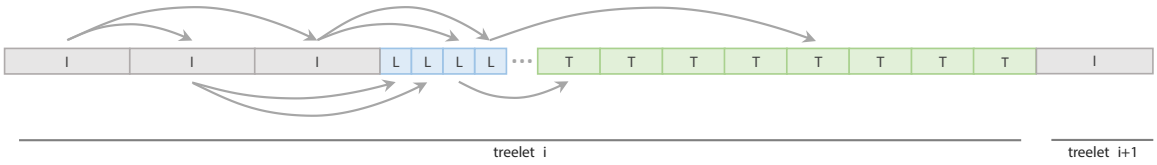
ing architecture versus on-demand cache loads for STRaTA. The traversal algorithm each architecture implements is also fundamentally different: the dual streaming architecture processes all rays within a given treelet before moving on to the next in a fixed order, whereas STRaTA processes treelets in an unknown order, on-demand as they fill with rays, and often some rays reenter treelets (thereby refetching the same scene data from DRAM). Another fundamental difference is that the dual streaming architecture traces ray wavefronts starting with the primary rays followed by secondary rays in separate passes, organized into ray streams that reside in DRAM. This enables many more rays in flight simultaneously, and combined with the deterministic traversal order, ensures that any treelet is loaded at most once per pass. Even if STRaTA was capable of spilling rays from on-chip ray queues into DRAM to allow more rays in flight, its traditional depth-first traversal would force it to (potentially) reload treelets many times. In comparison to STRaTA, the dual streaming architecture uses a completely different scheduler with different tasks, it contains a read-only scene buffer instead of a large L2 data cache, contains a dedicated hit record updater unit, and relies on an input ray staging buffer per TM. In other words, any similarity between the two architectures is limited to the design of individual TPs and TMs and the fact that treelets are used to split the scene data.

5.4 Scene Data Compression

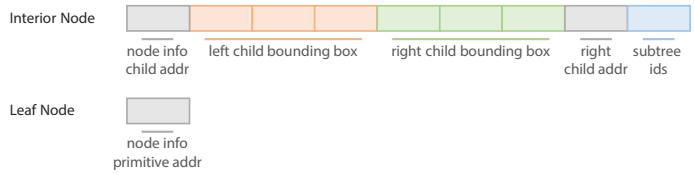
The dual streaming architecture stores the scene data in a convenient but uncompressed manner (Figure 5.18). The layout in memory can be improved in several ways. The first is a compact representation of the BVH nodes. In our implementation of the compressed scene stream layout for dual streaming, the size of interior nodes is reduced from 16 words (64B) to 9 words (36B) and the size of leaf nodes is halved from 2 words (8B) to 1 word (4B). The second improvement stores the triangle vertex data such that it can be reused across many triangles in an individual scene segment. Additionally, all addresses are stored as offsets from a given location rather than absolute values. This section discusses the effects on the performance of the dual streaming architecture provided by this scene compression scheme. Recent work has seen a resurgence of approaches aimed to compress scene data and improve ray tracing performance [1, 10, 35, 66, 71, 72, 86, 87, 90, 125, 145]. Investigating the effects of more aggressive compression schemes is left for future work.



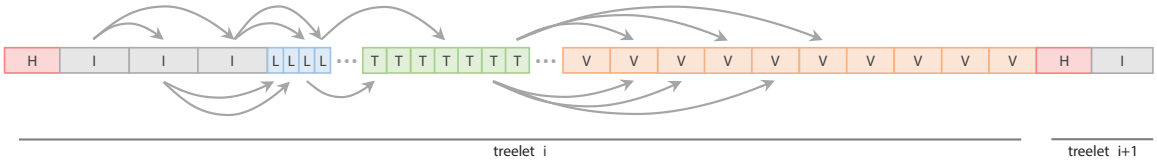
(a) Uncompressed BVH Nodes. Interior nodes use 16 words (64 bytes), while leaf nodes use 2 words (8 bytes).



(b) Uncompressed BVH Treelets. Each node is stored according to (a). Boxes are colored based on data type: gray for interior nodes (I), blue for leaf nodes (L), and green for triangles (T). Arrows indicate pointers from a node to its first child node or to the first triangle.



(c) Compressed BVH Nodes. Interior nodes use 9 words (36 bytes), while leaf nodes use 1 word (4 bytes).



(d) Compressed BVH Treelets. Each node is stored according to (c). Boxes are colored based on data type: red for treelet header (H), gray for interior nodes (I), blue for leaf nodes (L), green for triangles (T), and orange for triangle vertex data (V). Arrows indicate pointers from a node to its first child node or to the first triangle and from a triangle to its vertices.

Figure 5.18: In-memory layout for the uncompressed (a) - (b) and the compressed (c) - (d) scene streams that consist of BVH treelets.

To reduce the number of instructions used during ray traversal by decompressing axis-aligned bounding boxes, we modify the dedicated ray-box intersection pipeline unit to consume the quantized box representation directly. The implementation would require some bit swizzling and a special fused multiply-add instruction. We assume this adds just two cycles of latency to the intersection pipeline.

5.4.1 Memory Layout

The compressed scheme stores the acceleration structure nodes similarly to the uncompressed scheme (Section 4.1) but includes some small modifications, Figure 5.18. For internal nodes, the biggest reduction in size is because the AABBs are quantized to six

16-bit integers from six 32-bit floating point values, saving 6 words (24B) in storage per AABB. Each word packs the bounds for a single axis: the maximum is in the upper 16 bits and the minimum is in the lower 16 bits. The data is quantized relative to the fully uncompressed AABB of each scene segment.[2] Another word is saved from the internal node storage by combining the node header with the address of the left child. Instead of using a single word for each, the header needs only 8 bits and the address relies on the remaining 24 bits to store an offset from the address of the current node. The storage for leaf nodes can be halved by compacting the node header in a similar way.

Storing scene segments changes more significantly, Figure 5.18d. The compressed representation adds a 7 word header in the beginning, which stores the uncompressed AABB for the scene segment (6 words) and an address for the first triangle in the scene segment (1 word). Nodes are stored immediately after the scene segment header. All triangles within the scene segment follow, with unique vertex data saved at the end of the scene segment. Leaf nodes store the address of their first triangle as an offset from the address stored in the scene segment header. Each triangle stores pointers to the individual vertices as three 16-bit offsets from the current triangle address using 6 bytes. Neighboring triangles are interleaved sharing a word: the third vertex offset of a triangle $2i$ is stored in the upper 16 bits, while the first vertex offset of the triangle $2i + 1$ is stored in the lower 16 bits. Each vertex is stored as uncompressed floating point values, each using 3 words (12B).

5.4.2 Performance Effects

We show the performance of the outlined scene stream compression scheme in Table 5.3. Overall the scene data requires 62% less storage on average across all tested scenes but San Miguel (43% less). The number of treelets representing each scene is halved (57% fewer) for all tested scenes but San Miguel (only 22% fewer). Although the compression significantly reduces the number of cache lines transferred for the scene stream (at most 60%), the effect on the total data transferred from DRAM is not as pronounced ($4 - 23\%$) because the ray stream data transfer has not been reduced as much. The small reduction

[2]One can reduce the space for the bounding boxes in half again by avoiding the storage of redundant planes, like the Compact BVH method [35]. Applying such an approach for our test scenes would reduce the scene data stream by an additional $\sim$ 15%, however with a large increase in the complexity of the ray tracing algorithm implementation.

		Benchmark				Small		High Depth	
		Dragon	Dragon Box	Dragon Sponza	San Miguel	Fairy Forest	Crytek Sponza	Vegeta-tion	Hairball
Dual Streaming	Scene Size (MB)	129.29	129.28	914.16	1,493.54	24.98	38.00	149.91	369.97
	Num Treelets	1,586	1,600	11,276	17,421	293	455	1,682	3,972
	Render Time (ms/frame)	18.08	66.32	40.93	79.61	17.05	44.60	68.56	63.27
	Rays Traced per sec (M)	114.8	177.4	234.8	114.6	345.6	204.1	85.4	63.5
	Ray Duplication	4.55	3.14	4.18	15.19	3.00	8.55	15.15	16.02
	DRAM Energy (J)	1.15 (42%)	4.66 (40%)	4.47 (57%)	8.12 (50%)	1.61 (53%)	4.51 (50%)	4.41 (41%)	4.56 (43%)
	On-Chip Memory Energy (J)	1.52 (55%)	6.54 (57%)	3.10 (40%)	7.63 (47%)	1.30 (43%)	4.14 (46%)	5.96 (55%)	5.75 (54%)
	Compute Energy (J)	0.09 (3%)	0.36 (3%)	0.23 (3%)	0.53 (3%)	0.10 (3%)	0.32 (4%)	0.37 (3%)	0.33 (3%)
	Total Energy (J)	2.76	11.56	7.80	16.28	3.01	8.96	10.74	10.64
	Avg. Bandwidth (GB/s)	140.21	114.98	271.30	255.61	230.35	237.40	142.75	142.88
	Ray Stream $ Lines (M)	11.92 (30%)	45.81 (38%)	43.94 (25%)	146.34 (46%)	18.4 (31%)	80.56 (49%)	94.03 (61%)	76.22 (54%)
	Scene Stream $ Lines (M)	7.54 (19%)	8.0 (7%)	54.31 (31%)	72.27 (23%)	1.50 (3%)	2.26 (1%)	8.20 (5%)	19.18 (14%)
	Shading $ Lines (M)	17.43 (44%)	42.38 (36%)	45.00 (26%)	46.22 (15%)	30.53 (52%)	45.42 (27%)	32.46 (21%)	26.99 (19%)
	Hit Record $ Lines (M)	2.74 (7%)	22.97 (19%)	31.04 (18%)	53.93 (17%)	8.33 (14%)	37.21 (22%)	18.28 (12%)	18.89 (13%)
	Total $ Lines Trans. (M)	39.6	119.2	173.5	317.9	58.7	165.4	152.9	141.3
Dual Streaming Compressed Scene Stream	Scene size (MB)	48.70	48.70	344.10	845.71	9.38	14.38	58.51	134.74
	Num Treelets	679	671	4,767	13,592	132	204	776	1,625
	Render Time (ms/frame)	7.25	26.66	38.37	84.00	17.08	47.67	68.31	57.23
	Rays Traced per sec (M)	287.2	440.5	250.0	108.6	342.2	191.1	85.7	69.4
	Ray Duplication	3.72	2.60	4.01	15.93	2.65	8.00	12.94	11.98
	DRAM Energy (J)	0.71 (64%)	2.85 (56%)	3.91 (57%)	8.16 (53%)	1.58 (57%)	4.42 (51%)	4.17 (42%)	3.71 (42%)
	On-Chip Memory Energy (J)	0.37 (33%)	2.01 (40%)	2.65 (39%)	6.62 (43%)	1.08 (39%)	3.77 (44%)	5.33 (53%)	4.71 (53%)
	Compute Energy (J)	0.04 (4%)	0.22 (4%)	0.29 (4%)	0.70 (5%)	0.11 (4%)	0.41 (5%)	0.47 (5%)	0.39 (4%)
	Total Energy (J)	1.12	5.07	6.85	15.48	2.77	8.61	9.96	8.82
	Avg. Bandwidth (GB/s)	273.53	239.15	230.77	237.82	206.26	213.74	126.29	120.34
	Ray Stream $ Lines (M)	8.23 (27%)	32.56 (33%)	41.07 (30%)	152.52 (49%)	16.16 (29%)	75.29 (47%)	80.38 (60%)	54.39 (51%)
	Scene Stream $ Lines (M)	3.23 (10%)	3.37 (3%)	22.30 (16%)	57.97 (19%)	0.61 (1%)	0.94 (1%)	3.67 (3%)	7.73 (7%)
	Shading $ Lines (M)	16.93 (55%)	41.48 (42%)	44.54 (32%)	47.54 (15%)	30.31 (55%)	45.68 (29%)	33.33 (25%)	28.35 (26%)
	Hit Record $ Lines (M)	2.58 (8%)	22.22 (22%)	30.49 (22%)	54.12 (17%)	7.96 (14%)	37.28 (23%)	17.42 (13%)	17.15 (16%)
	Total $ Lines Trans. (M)	31.0	99.6	138.4	312.1	55.0	159.2	134.8	107.6

Table 5.3: The performance results comparing the effects of compressing the scene stream for the dual streaming architecture. Note $ means cache and M means millions.

in the ray duplication rate also supports this conclusion. The energy used and time taken to render a single frame are reduced.

5.5 Limitations

The main feature of the dual streaming algorithm is the refactoring of the ray tracing algorithm into two predictable data streams where the scene data is loaded at most once per ray pass. At a very high level this is in some ways a reversal from traditional algorithms. Instead of tracing a ray to completion while loading scene data (treelets) on demand, a portion of the scene is loaded once and all rays that intersect with it are streamed through. While the memory bandwidth required for scene data is reduced significantly, the dual streaming architecture requires bandwidth for ray data (Figure 5.8). This, in turn, becomes an interesting challenge for the current version of the proposed hardware - how to manage the memory bandwidth required for the ray traffic?

Architecture	GB/s	MRPS	Bytes/Ray
Aila et al. [1]	2.0 - 3.8 GB/fr.	3.3 MR/fr.	598 - 1137
RayCore [95]	0.4 - 0.6	18 - 20	20 - 33
Liktor et al. [86]	not given	155 - 335	61 - 298
(bench)	138 - 267	73 - 135	1015 - 3021
STRaTA (small)	99 - 192	233 - 366	271 - 437
(depth)	230 - 255	112 - 121	1891 - 2269
(bench)	115 - 271	115 - 235	648 - 2230
Dual Streaming (small)	230 - 237	204 - 346	667 - 1163
(depth)	143	64 - 85	1672 - 2250
Dual Streaming (bench)	230 - 274	109 - 441	542 - 2190
Compressed (small)	206 - 214	191 - 342	603 - 1119
Scene Stream (depth)	120 - 126	69 - 86	1473 - 1734

Table 5.4: Main memory utilization measured in bytes per ray for comparison architectures (lower is better). GB/s is the total main memory bandwidth and MRPS is millions of rays traced per second.

One way to compare to other systems is by considering how much data the system fetches per ray, shown in Table 5.4. For example, in terms of Bytes/Ray our dual streaming implementation is within a factor of two from the seminal BVH treelet architecture [1]. One reason for this difference is that the dual streaming architecture can not perform early ray termination. This can generate a lot of extra ray traffic between treelets. For scenes with high depth complexity (Vegetation and Hairball), this problem is increased because the number of treelets a ray intersects is proportional to the number of spatially distant leaf nodes it must visit.

We can compare against another architecture targeting mobile platforms, RayCore [95]. It benchmarks using small scenes that fit nicely into on-chip caches and therefore traditional algorithms that keep rays on chip result in very small traffic to main memory. While the dual streaming architecture still reduces the scene traffic significantly, small scenes show the overhead of ray streams.

While the dual streaming architecture reduces scene traffic by introducing fixed scene segment traversal order, other researchers have made good progress in compressing the BVH data and thus reducing the number of bytes transferred per ray, for example [86]. This method compresses BVH layouts in a manner similar to the method by Yoon and Manocha [146], but adds modifications to compress treelet-interior pointers and optimize layout for caches. A direct comparison of the Bytes/Ray metric is complicated by the dif-

ferent memory architecture and heavy instancing in some of their test scenes. Additionally, the authors use index nodes to achieve a 50% reduction in L2 to L1 bandwidth, which we do not attempt. The dual streaming architecture is largely orthogonal to such memory optimizations. Applying a similar scene data compression technique could help reduce scene traffic further, but more importantly it would reduce the number of scene segments, thus reducing both ray duplication and ray traffic to DRAM.

These comparisons offer only a partial evaluation because raw memory traffic does not consider DRAM management like row buffer hit rates (see Section 2.2.3.2) which can have a huge impact on the actual latency and power of the memory system. In fact, it is possible for the memory bandwidth to increase while reducing access latency [76]. The comparisons also point out the main challenge in extending the dual streaming architecture: managing, minimizing, and compressing ray traffic (see the discussion of future work in Chapter 6).

Because the dual streaming architecture processes wavefronts of rays, renderers and scenes that generate many ray bounces would require many passes, which could result in an undesired drop in processor utilization and an increase in memory traffic per ray. A limit to the number of ray bounces would bound this, but also introduce rendering bias, which, depending on the scene and light transport, may or may not be negligible.

The dual streaming architecture does not directly address building the acceleration structure and scene segments on chip. Because the dual streaming architecture is envisioned as a graphics accelerator, an external processor would generate the scene stream and load it into DRAM before rendering a frame. However, the dual streaming architecture could be modified to rely on its general purpose execution units for this task.

CHAPTER 6

FUTURE WORK AND CONCLUSION

Because the dual streaming architecture completely reorders the traversal within ray tracing to address the fundamental problems of the traditional traversal order, it also exposes new and interesting challenges. They represent fertile ground for additional optimizations and future research. We discuss some of these challenges here.

Treelet Assignment Optimizing treelet assignment to limit ray duplication and thus ray bandwidth is likely to yield better performance than optimizing to accelerate the traversal of individual rays. For example, it is unclear if treelets should prefer a shallow structure or if they should be constructed in a depth-first fashion producing deeper treelets. We would expect a combination of the two approaches to deliver superior performance.

Treelet Traversal Order Once a scene segment is processed, any and all of its child segments can be selected into the working set. Adjusting the predefined scene segment traversal order based on the structure of the BVH or altering the traversal order on-the-fly based on information gathered during the traversal of the parent scene segments can provide substantial performance benefits. For example, in our current implementation it is likely that rays are not traversed through the closer scene segment first, reducing the effectiveness of any early ray termination scheme. Modifying the scene segment traversal order to be more amenable for early ray termination without significant increase in memory traffic is an important open problem.

Early Ray Termination Unlike traditional ray tracing, implementing early ray termination is not trivial with the dual streaming algorithm. Since the ray stream excludes unique hit information shared by ray duplicates, the hit information must be gathered from the hit record separately. There are several alternatives to our current implementation. For example, if the hit record is not already on chip, it might be more beneficial for the pre-test approach to traverse the ray anyway, instead of stalling until the hit record

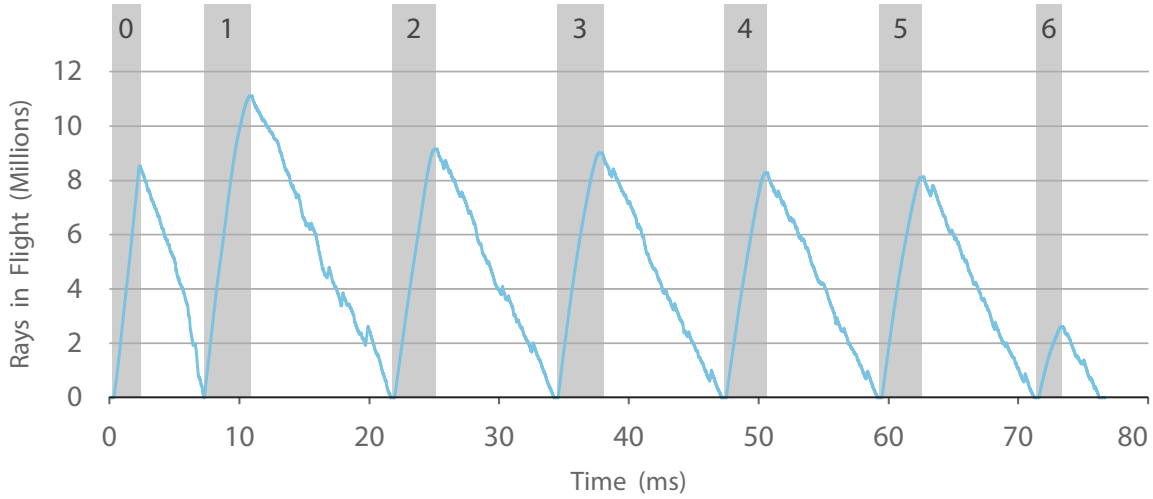


Figure 6.1: Number of rays simultaneously in flight for the San Miguel scene. Each ray wavefront is distinguished by a triangular spike in the number of rays. Vertical gray bars indicate durations of the ray generation phase with depths marked on top. Note the ratio of time spent on ray generation and traversal.

read is serviced. Alternatively, a ray waiting for the hit record data can be simply skipped until the data arrives. Also, the hit information can be requested and cached for the entire bucket of rays before processing or even scheduling it.

Processing Ray Wavefronts The proposed dual streaming architecture processes all rays at a given depth at once and continues onto the rays at the next depth only once the current wavefront finishes. As a result, the algorithm operates in phases, each of which can be categorized by a triangular shape seen in the total number of rays in flight (Figure 6.1). Improving the render times would likely rely on overlapping processing of different wavefronts, relaxing the fixed scene segment traversal order allowing reloading some of the scene segments, or allowing a ray to be traversed to completion and loading the necessary scene data on demand.

Data Compression The ray stream is the major component of the memory bandwidth used by the dual streaming architecture. Therefore, compressing the duplicated ray data might significantly improve the performance and reduce the energy cost [66]. While the scene stream uses only a fraction of the memory bandwidth, compressing the scene data can still be helpful in reducing the number of scene segments and thereby reducing the number of duplicated rays. We evaluated a simple scene stream compression scheme, which helps improve the rendering performance slightly, a more aggressive compression

is required to achieve significant gains. Reducing the ray duplication rate would reduce the total memory bandwidth further.

On-Chip Ray Storage Another way to reduce the effects of ray stream traffic to DRAM is by storing all of the required rays on chip, where the bandwidth is significantly higher than the off-chip DRAM bandwidth. Unfortunately, on-chip SRAM storage is very limited by the available area. An architecture could store the ray data within stacked DRAM memory placed above the chip or spread across many processing chips connected together.

Memory Optimizations It may be possible to partition streams between on-chip and off-chip DRAM, in order to reduce energy further. In particular, because the bandwidth requirements for the scene stream are low and the access latency is hidden by prefetching, the scene data can reside on a slower off-chip memory. Furthermore, it may be possible to lower the operating frequency of the off-chip memory serving the scene stream to significantly reduce the energy use without impacting performance negatively. Moreover, the dual streaming architecture has the potential to take the full advantage of the upcoming high bandwidth memory (HBM) systems [60] and hide their additional latency through streaming.

DRAM Modifications Ray streams effectively convert DRAM into a temporary staging buffer that writes and reads rays only once. Thus, after ray data is read from DRAM, there is no need to preserve it, which requires DRAM to write the contents of the row buffer back thus consuming energy and contributing to memory latency. A DRAM modified for ray streams could benefit from "destructive reads" which would avoid these costs.

Additional Streaming Opportunities Our scene segment traversal guarantees that the scene geometry is accessed at most once per ray wavefront pass. This structure can be used for rendering extremely large scenes that cannot fit in memory by streaming them from a disk or other high latency locations [32].

Dynamic Scenes The presented dual streaming architecture assumes the scenes are static. Supporting dynamic scenes presents several challenges. First, the rendering algorithm must integrate over the frame time interval by tracing rays through the dynamic scene, typically achieved with time sampling. Secondly, specifying geometry in motion requires storing a representation of the motion for each primitive, typically using two or more keyframes each at a specific frame time. This requires the acceleration structure

capable of storing and bounding such motion [42, 135]. Third, the acceleration structure should be able to get updated or rebuilt quickly enough to generate the image for the next frame.

Enabling support for the dual streaming architecture can be achieved in several ways. One way is to modify the BVH treelets to store the keyframes (for both nodes and triangles) in each treelet and simply update the triangle positions from frame to frame. However, as the scene evolves over time, the ray tracing performance will degrade because the SAH cost of the BVH tree would increase without updating or rebuilding it [74]. Alternatively, one could triangulate the surface of the volume representing the triangle motion, and intersect against it directly [119]. This method can use the acceleration structure and ray traversal as provided by the dual streaming architecture without modifications to account for time.

One could also consider tracing multiple animation frames simultaneously, which would improve the hardware utilization by tracing rays from different frames each at different depths.

Shading The proposed dual streaming architecture focuses on the ray intersection and traversal phase of the path tracing algorithm. The tests rely on simple diffuse materials without textures, but both real-time and offline rendering applications use more sophisticated material models. Fortunately, the existing SIMD GPU architectures are great at shading hit points in parallel provided the necessary material texture data is available (memory access latency is hidden by letting each SIMD multiprocessor compute several kernels simultaneously). SIMD architectures are not great at ray traversal because of the ray divergence both in terms of data and control-flow. However, adding dedicated ray traversal hardware into the graphics rendering pipeline could enable tracing rays from within shading kernels to generate superior reflections and shadows.

At the time of writing, several games that use the dedicated ray tracing hardware within NVIDIA's Turing architecture rely on hybrid rendering methods where the primary visibility and material properties are generated using the dedicated rasterization hardware. For a subset of pixels output from the rasterization pass, secondary effects like reflections, shadows, and even global illumination are generated during the shading pass. Games apply denoising algorithms to the ray tracing output to reconstruct and filter the

secondary effects across the image. Combining SIMD-style processing within the dual streaming architecture to create a hybrid rendering system is an interesting direction for future work.

6.1 Conclusion

We introduced the dual streaming approach that restructures ray traversal into a predictable process that allows both the scene data and the ray data to be streamed from the main memory in a highly structured way. This approach is tailored to the fundamental operation of DRAM memory, where data accessed sequentially from an open row buffer is dramatically more efficient in both energy and latency than more random accesses. This streaming approach also eliminates some major bottlenecks inherent in the traditional ray tracing traversal order.

We also provided a first hardware implementation of the dual streaming algorithm and test results using a cycle-accurate hardware simulator, showing that our implementation of the dual streaming architecture already outperforms STRaTA, a highly optimized architecture for traditional ray tracing, in typical large scenes. Finally, we included an extensive discussion for potential future improvements to the dual streaming implementation, providing a new avenue for further research on hardware accelerated ray tracing.

REFERENCES

[1] AILA, T., AND KARRAS, T. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG '10, pp. 113–122.

[2] AILA, T., KARRAS, T., AND LAINE, S. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), HPG '13, pp. 101–107.

[3] AILA, T., AND LAINE, S. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, pp. 145–149.

[4] AILA, T., LAINE, S., AND KARRAS, T. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.

[5] ALBERA, G., AND BAHAR, R. I. Power/performance advantages of victim buffer in high-performance processors. In *Proceedings IEEE Alessandro Volta Memorial Workshop on Low-Power Design* (March 1999), pp. 43–51.

[6] AMD. White paper: Introducing the Radeon Rays SDK. https://32ipi028l5q82yhj72224m8j-wpengine.netdna-ssl.com/wp-content/uploads/2016/08/169798-A_AMD_RadeonRays_Intro_FNL.pdf.

[7] ARVO, J., AND KIRK, D. Particle transport and image synthesis. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (1990), SIGGRAPH '90, pp. 63–66.

[8] BAKHODA, A., YUAN, G. L., FUNG, W. W. L., WONG, H., AND AAMODT, T. M. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (April 2009), pp. 163–174.

[9] BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of MICRO-33* (December 2000), pp. 245–257.

[10] BAUSZAT, P., EISEMANN, M., AND MAGNOR, M. A. The minimal bounding volume hierarchy. In *Vision, Modeling, and Visualization* (2010), pp. 227–234.

[11] BENTHIN, C., WALD, I., AND SLUSALLEK, P. Interactive ray tracing of free-form surfaces. In *Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa* (2004), AFRIGRAPH '04, pp. 99–106.

[12] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept. 1975), 509–517.

[13] Bigler, J., Stephens, A., and Parker, S. G. Design for parallel interactive ray tracing systems. In *2006 IEEE Symposium on Interactive Ray Tracing* (Sep. 2006), pp. 187–196.

[14] Bikker, J. Improving data locality for efficient in-core path tracing. *Computer Graphics Forum 31*, 6 (2012), 1936–1947.

[15] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The Gem5 simulator. *ACM SIGARCH Computer Architecture News 39*, 2 (Aug. 2011), 1–7.

[16] Bojnordi, M. N., and Ipek, E. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *International Symposium on Computer Architecture* (2012), ISCA '12, pp. 13–24.

[17] Bresenham, J. E. Algorithm for computer control of a digital plotter. *IBM Systems Journal 4*, 1 (1965), 25–30.

[18] Brunvand, E., Kopta, D., and Chatterjee, N. Why graphics programmers need to know about DRAM. In *ACM SIGGRAPH 2014 Courses* (2014), SIGGRAPH '14, pp. 21:1–21:75.

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

[20] Burley, B., Adler, D., Chiang, M. J.-Y., Driskill, H., Habel, R., Kelly, P., Kutz, P., Li, Y. K., and Teece, D. The design and evolution of Disney's Hyperion renderer. *ACM Transactions on Graphics 37*, 3 (July 2018), 33:1–33:22.

[21] Cassagnabère, C., Rousselle, F., and Renaud, C. Path tracing using the AR350 processor. In *Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia* (2004), GRAPHITE '04, pp. 23–29.

[22] Catmull, E., and Clark, J. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10*, 6 (1978), 350 – 355.

[23] Chatterjee, N., Balasubramonian, R., Shevgoor, M., Pugsley, S. H., Udipi, A. N., Shafiee, A., Sudan, K., Awasthi, M., and Chishti, Z. USIMM: the Utah SImulated Memory Module. Tech. Rep. UUCS-12-02, University of Utah, 2012.

[24] Chow, P. *The MIPS-X RISC Microprocessor*. Springer Science & Business Media, 1989.

[25] Christensen, P., Fong, J., Shade, J., Wooten, W., Schubert, B., Kensler, A., Friedman, S., Kilpatrick, C., Ramshaw, C., Bannister, M., Rayner, B., Brouillat, J., and Liani, M. RenderMan: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics (TOG) 37*, 3 (Aug. 2018), 30:1–30:21.

[26] Cleary, J. G., and Wyvill, G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 4*, 2 (1988), 65–83.

[27] Cook, R. L., Porter, T., and Carpenter, L. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (1984), SIGGRAPH '84, pp. 137–145.

[28] Dally, B. The challenge of future high-performance computing. Celsius Lecture, Uppsala University, Uppsala, Sweden, Feb 2013. http://media.medfarm.uu.se/play/video/3261.

[29] Deng, Y., Ni, Y., Li, Z., Mu, S., and Zhang, W. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys (CSUR) 50*, 4 (Aug. 2017), 58:1–58:41.

[30] Dmitriev, K., Havran, V., and Seidel, H.-P. Faster ray tracing with SIMD shaft culling. Tech. Rep. MPI-I-2004-4-006, Max-Planck-Institut für Informatik, 2004.

[31] Domingues, L., and Pedrini, H. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics* (2015), HPG '15, pp. 13–20.

[32] Eisenacher, C., Nichols, G., Selle, A., and Burley, B. Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering* (2013), EGSR '13, pp. 125–132.

[33] Ernst, M., and Greiner, G. Early split clipping for bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing* (Sept 2007), pp. 73–78.

[34] Ernst, M., and Greiner, G. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing* (Aug 2008), pp. 35–40.

[35] Fabianowski, B., and Dingliana, J. Compact BVH storage for ray tracing and photon mapping. In *Eurographics Ireland 2009* (2009), pp. 1–8.

[36] Farin, G. E. *NURB Curves and Surfaces: From Projective Geometry to Practical Use*. A. K. Peters, Ltd., Natick, MA, USA, 1995.

[37] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21*, 9 (Sept 1972), 948–960.

[38] Fuetterling, V., Lojewski, C., Pfreundt, F.-J., and Ebert, A. Efficient ray tracing kernels for modern CPU architectures. *Journal of Computer Graphics Techniques (JCGT) 4*, 5 (December 2015), 90–111.

[39] Garrard, A. Cold chips: ARTs RenderDrive architecture ray tracing hardware from before the GPU. In *Proceedings of The Conference on High Performance Graphics, Hot3D Session* (2018).

[40] Georgiev, I., Ize, T., Farnsworth, M., Montoya-Vozmediano, R., King, A., Lommel, B. V., Jimenez, A., Anson, O., Ogaki, S., Johnston, E., Herubel, A., Russell, D., Servant, F., and Fajardo, M. Arnold: A brute-force production path tracer. *ACM Transactions on Graphics (TOG) 37*, 3 (Aug. 2018), 32:1–32:12.

[41] Glassner, A. S. Space subdivision for fast ray tracing. *Computer Graphics and Applications, IEEE 4*, 10 (1984), 15–24.

[42] GLASSNER, A. S. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications 8*, 2 (March 1988), 60–70.

[43] GOLDSMITH, J., AND SALMON, J. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

[44] GOVINDARAJU, V., DJEU, P., SANKARALINGAM, K., VERNON, M., AND MARK, W. R. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (October 2008), MICRO 41, pp. 176–187.

[45] GRIBBLE, C., AND RAMANI, K. Coherent ray tracing via stream filtering. In *2008 IEEE Symposium on Interactive Ray Tracing* (Aug 2008), pp. 59–66.

[46] GU, Y., HE, Y., FATAHALIAN, K., AND BLELLOCH, G. Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), HPG '13, pp. 81–88.

[47] HACHISUKA, T., JAROSZ, W., BOUCHARD, G., CHRISTENSEN, P., FRISVAD, J. R., JAKOB, W., JENSEN, H. W., KASCHALK, M., KNAUS, C., SELLE, A., AND SPENCER, B. State of the art in photon density estimation. In *ACM SIGGRAPH 2012 Courses* (2012), SIGGRAPH '12, pp. 6:1–6:469.

[48] HALL, D. The AR350: Todays ray trace rendering processor. In *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware-Hot 3D Session* (2001), vol. 1, p. 2.

[49] HAVRAN, V. *Heuristic Ray Shooting Algorithms*. PhD Thesis, Czech Technical University, Prague, 2000.

[50] HAVRAN, V., BITTNER, J., AND ZARA, J. Ray tracing with rope trees. In *Proceedings of 13th Spring Conference On Computer Graphics* (1998), pp. 130–139.

[51] HENNESSY, J., AND PATTERSON, D. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. ISCA 2018 Turing Lecture.

[52] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach (5th ed.)*. Morgan Kaufmann, 2012.

[53] HENNING, J. L. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News 34*, 4 (Sept. 2006), 1–17.

[54] HURLEY, J., KAPUSTIN, A., RESHETOV, A., AND SOUPIKOV, A. Fast ray tracing for modern general purpose CPU. In *Proceedings of GraphiCon* (2002), vol. 2002.

[55] IMMEL, D., COHEN, M., AND GREENBERG, D. A radiosity method for non-diffuse environments. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986), SIGGRAPH '86, pp. 133–142.

[56] IZE, T., SHIRLEY, P., AND PARKER, S. Grid creation strategies for efficient ray tracing. In *2007 IEEE Symposium on Interactive Ray Tracing* (Sep. 2007), pp. 27–32.

[57] JACOB, B., NG, S., AND WANG, D. *Memory Systems - Cache, DRAM, Disk*. Elsevier, 2008.

[58] JAKOB, W. Mitsuba renderer. www.mitsuba-renderer.org, 2010.

[59] JAKOB, W., D'EON, E., JAKOB, O., AND MARSCHNER, S. A comprehensive framework for rendering layered materials. *ACM Transactions on Graphics (TOG) 33*, 4 (July 2014), 118:1–118:14.

[60] JDEC STANDARD. High bandwidth memory (HBM) DRAM. Tech. Rep. JESD325A, JDEC Solid State Technology Association, Nov 2015.

[61] JENSEN, H. Importance driven path tracing using the photon map. In *Rendering Techniques '95*, Eurographics. 1995, pp. 326–335.

[62] JENSEN, H. Global illumination using photon maps. In *Rendering Techniques '96*, Eurographics. 1996, pp. 21–30.

[63] JEVANS, D., AND WYVILL, B. Adaptive voxel subdivision for ray tracing. *Proceedings of Graphics Interface '89* (1988), 164–172.

[64] KAJIYA, J. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986), SIGGRAPH '86, pp. 143–150.

[65] KAY, T., AND KAJIYA, J. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986), SIGGRAPH '86, pp. 269–278.

[66] KEELY, S. Reduced precision for hardware ray tracing in GPUs. In *Proceedings of High Performance Graphics* (2014), HPG '14, pp. 29–40.

[67] KENSLER, A. Tree rotations for improving bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing* (Aug 2008), pp. 73–76.

[68] KIM, H., KIM, Y., OH, J., AND KIM, L. A reconfigurable SIMT processor for mobile ray tracing with contention reduction in shared memory. *IEEE Transactions on Circuits and Systems I: Regular Papers 60*, 4 (April 2013), 938–950.

[69] KIM, H.-Y., KIM, Y.-J., AND KIM, L.-S. Reconfigurable mobile stream processor for ray tracing. In *IEEE Custom Integrated Circuits Conference 2010* (September 2010), pp. 1–4.

[70] KIM, H.-Y., KIM, Y.-J., AND KIM, L.-S. MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization. *IEEE Journal of Solid-State Circuits 47*, 2 (Feb. 2012), 518–535.

[71] KIM, T., MOON, B., KIM, D., AND YOON, S. RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (March 2010), 273–286.

[72] KIM, T.-J., BYUN, Y., KIM, Y., MOON, B., LEE, S., AND YOON, S.-E. HCCMeshes: Hierarchical-culling oriented compact meshes. *Computer Graphics Forum 29*, 2 (2010), 299–308.

[73] KIRK, D., AND ARVO, J. Improved ray tagging for voxel-based ray tracing. *Graphics Gems II* (1991), 264–266.

[74] KOPTA, D., IZE, T., SPJUT, J., BRUNVAND, E., DAVIS, A., AND KENSLER, A. Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012), pp. 197–204.

[75] KOPTA, D., SHKURKO, K., SPJUT, J., BRUNVAND, E., AND DAVIS, A. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), HPG '13, pp. 121–128.

[76] KOPTA, D., SHKURKO, K., SPJUT, J., BRUNVAND, E., AND DAVIS, A. Memory considerations for low energy ray tracing. *Computer Graphics Forum 34*, 1 (2015), 47–59.

[77] KOPTA, D., SPJUT, J., BRUNVAND, E., AND DAVIS, A. Efficient MIMD architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design (ICCD)* (Oct. 2010), pp. 9–16.

[78] LAINE, S., AND KARRAS, T. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation 2* (2010).

[79] LAINE, S., AND KARRAS, T. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (Aug 2011), 1048–1059.

[80] LATTNER, C. LLVM and Clang: Next generation compiler technology. In *The BSD Conference* (2008), pp. 1–2.

[81] LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. Fast BVH construction on GPUs. In *Computer Graphics Forum* (2009), vol. 28, pp. 375–384.

[82] LEE, W.-J., LEE, S.-H., NAH, J.-H., KIM, J.-W., SHIN, Y., LEE, J., AND JUNG, S.-Y. SGRT: a scalable mobile GPU architecture based on ray tracing. In *ACM SIGGRAPH 2012 Posters* (2012), SIGGRAPH '12, pp. 44:1–44:1.

[83] LEE, W.-J., SHIN, Y., HWANG, S. J., KANG, S., YOO, J.-J., AND RYU, S. Reorder buffer: an energy-efficient multithreading architecture for hardware MIMD ray traversal. In *Proceedings of the 7th Conference on High-Performance Graphics* (2015), HPG '15, pp. 21–32.

[84] LEE, W.-J., SHIN, Y., LEE, J., KIM, J.-W., NAH, J.-H., JUNG, S., LEE, S., PARK, H.-S., AND HAN, T.-D. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), HPG '13, pp. 109–119.

[85] LEE, W.-J., SHIN, Y., LEE, J., LEE, S., RYU, S., AND KIM, J. Real-time ray tracing on future mobile computing platform. In *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications* (2013), SA '13, pp. 56:1–56:5.

[86] LIKTOR, G., AND VAIDYANATHAN, K. Bandwidth-efficient BVH layout for incremental hardware traversal. In *Proceedings of High Performance Graphics* (2016), HPG '16, pp. 51–61.

[87] Lin, D., Shkurko, K., Mallett, I., and Yuksel, C. Dual-split trees. In *Symposium on Interactive 3D Graphics and Games (I3D 2019)* (May 2019).

[88] MacDonald, J. D., and Booth, K. S. Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166.

[89] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer 35*, 2 (2002), 50–58.

[90] Mahovsky, J., and Wyvill, B. Memory-conserving bounding volume hierarchies with coherent raytracing. *Computer Graphics Forum 25*, 2 (2006), 173–182.

[91] Meister, D., and Bittner, J. Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 3 (March 2018), 1345–1353.

[92] Morton, G. M. A computer oriented geodetic data base and a new technique in file sequencing. *International Business Machines Company, New York* (1966).

[93] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. 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* (2007), MICRO 40, pp. 3–14.

[94] Nah, J.-H., Kim, J.-W., Park, J., Lee, W.-J., Park, J.-S., Jung, S.-Y., Park, W.-C., Manocha, D., and Han, T.-D. HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3 (March 2015), 389–401.

[95] Nah, J.-H., Kwon, H.-J., Kim, D.-S., Jeong, C.-H., Park, J., Han, T.-D., Manocha, D., and Park, W.-C. RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics (TOG) 33*, 5 (Sept. 2014), 162:1–162:15.

[96] Nah, J.-H., Park, J.-S., Park, C., Kim, J.-W., Jung, Y.-H., Park, W.-C., and Han, T.-D. T&I engine: Traversal and intersection engine for hardware accelerated ray tracing. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (2011), SA '11, pp. 160:1–160:10.

[97] Navrátil, P., Fussell, D., Lin, C., and Mark, W. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *2007 IEEE Symposium on Interactive Ray Tracing* (Sept. 2007), pp. 95 – 104.

[98] Naylow, B., and Thibault, W. Application of BSP trees to ray-tracing and CSG evaluation. Tech. Rep. GOT-ICS 86/03, School of Information and Computer Science, Georga Institute of Technology, 1986.

[99] NVIDIA. White paper: NVIDIA turing GPU architecture, graphics reinvented. www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[100] Pantaleoni, J., and Luebke, D. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics* (2010), HPG '10, pp. 87–95.

[101] Park, W.-C., Shin, H.-J., Lee, B., Yoon, H., and Han, T.-D. RayChip: Real-time ray-tracing chip for embedded applications. In *2014 IEEE Hot Chips 26 Symposium (HCS)* (Aug 2014), pp. 1–32.

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

[103] Patterson, D., and Ditzel, D. The case for the reduced instruction set computer. *SIGARCH Computer Architecture News 8*, 6 (Oct. 1980), 25–33.

[104] Patterson, D., and Sequin, C. RISC I: A reduced instruction set VLSI computer. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)* (1998), ISCA '98, pp. 216–230.

[105] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation (3rd ed.)*. Morgan Kaufmann Publishers Inc., 2016.

[106] Phong, B. T. Illumination for computer generated pictures. *Commununications of the ACM 18*, 6 (June 1975), 311–317.

[107] Pineda, J. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques* (1988), SIGGRAPH '88, pp. 17–20.

[108] Popov, S., Gunther, J., Seidel, H.-P., and Slusallek, P. Experiences with streaming construction of SAH KD-trees. In *2006 IEEE Symposium on Interactive Ray Tracing* (Sep. 2006), pp. 89–94.

[109] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Stackless KD-tree traversal for high performance GPU ray tracing. In *Computer Graphics Forum* (2007), vol. 26, pp. 415–424.

[110] Ramani, K., Gribble, C., and Davis, A. StreamRay: A stream filtering architecture for coherent ray tracing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ASPLOS XIV, pp. 325–336.

[111] Reshetov, A., Soupikov, A., and Hurley, J. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (SIGGRAPH '05) 24*, 3 (2005), 1176–1185.

[112] Schmittler, J., Wald, I., and Slusallek, P. SaarCOR: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (September 2002), HWWS '02, pp. 27–36.

[113] Schmittler, J., Woop, S., Wagner, D., Paul, W. J., and Slusallek, P. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (August 2004), HWWS '04, pp. 95–106.

[114] Shevtsov, M., Soupikov, A., Kapustin, A., and Novorod, N. Ray-triangle intersection algorithm for modern CPU architectures. In *Proceedings of GraphiCon'2007* (June 2007), pp. 33–39.

[115] Shin, Y., Lee, W.-J., Lee, J., Lee, S.-H., Ryu, S., and Kim, J. Energy efficient data transmission for ray tracing on mobile computing platform. In *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications* (2013), SA '13, pp. 64:1–64:5.

[116] Shkurko, K., Grant, T., Brunvand, E., Kopta, D., Spjut, J., Vasiou, E., Mallett, I., and Yuksel, C. SimTRaX: Simulation infrastructure for exploring thousands of cores. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI* (2018), GLSVLSI '18, pp. 503–506.

[117] Shkurko, K., Grant, T., Brunvand, E., Kopta, D., Spjut, J., Vasiou, E., Mallett, I., and Yuksel, C. SimTRaX: Simulation infrastructure for exploring thousands of cores. Tech. Rep. UUCS-18-001, School of Computing, University of Utah, 2018.

[118] Shkurko, K., Grant, T., Kopta, D., Mallett, I., Yuksel, C., and Brunvand, E. Dual streaming for hardware-accelerated ray tracing. In *Proceedings of High Performance Graphics* (2017), HPG '17, pp. 12:1–12:11.

[119] Shkurko, K., Yuksel, C., Kopta, D., Mallett, I., and Brunvand, E. Time interval ray tracing for motion blur. *IEEE Transactions on Visualization and Computer Graphics 24*, 12 (2018), 3225–3238.

[120] Spjut, J., Kensler, A., Kopta, D., and Brunvand, E. TRaX: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28*, 12 (Dec 2009), 1802 – 1815.

[121] Spjut, J., Kopta, D., Boulos, S., Kellis, S., and Brunvand, E. TRaX: A multi-threaded architecture for real-time ray tracing. In *2008 Symposium on Application Specific Processors* (June 2008), pp. 108–114.

[122] Stam, J. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (1998), SIGGRAPH '98, pp. 395–404.

[123] Stich, M., Friedrich, H., and Dietrich, A. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), HPG '09, pp. 7–13.

[124] Sung, K., and Shirley, P. Ray tracing with the BSP tree. In *Graphics Gems III*, D. Kirk, Ed. Academic Press Professional, Inc., 1992, pp. 271 – 274.

[125] Vaidyanathan, K., Akenine-Möller, T., and Salvi, M. Watertight ray traversal with reduced precision. In *Proceedings of High Performance Graphics* (2016), HPG '16, pp. 33–40.

[126] Vasiou, E., Shkurko, K., Mallett, I., Brunvand, E., and Yuksel, C. A detailed study of ray tracing performance: render time and energy cost. *The Visual Computer 34*, 6 (Jun 2018), 875–885.

[127] VIITANEN, T., KOSKELA, M., JÄÄSKELÄINEN, P., KULTALA, H., AND TAKALA, J. MergeTree: A HLBVH constructor for mobile systems. In *SIGGRAPH Asia 2015 Technical Briefs* (2015), SA '15, pp. 12:1–12:4.

[128] VIITANEN, T., KOSKELA, M., JÄÄSKELÄINEN, P., TERVO, A., AND TAKALA, J. PLOCTree: A fast, high-quality hardware BVH builder. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (Aug. 2018), 35:1–35:19.

[129] VOICA, A. A game of shadows: ray traced shadows vs. cascaded shadow maps. Imagination Technologies. blog.imgtec.com/multimedia/ray-traced-shadows-vs-cascaded-shadow-maps.

[130] VOICA, A. PowerVR GR6500: ray tracing is the future... and the future is now. Imagination Technologies. blog.imgtec.com/powervr-developers/powervr-gr6500-ray-tracing.

[131] WÄCHTER, C., AND KELLER, A. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques* (2006), EGSR '06, pp. 139–149.

[132] WALD, I., BENTHIN, C., AND BOULOS, S. Getting rid of packets - efficient SIMD single-ray traversal using multi-branching BVHs. In *Symposium on Interactive Ray Tracing (IRT08)* (2008), pp. 49–57.

[133] WALD, I., BOULOS, S., AND SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG) 26*, 1 (Jan. 2007).

[134] WALD, I., AND HAVRAN, V. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 61–69.

[135] WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. Ray tracing animated scenes using coherent grid traversal. In *ACM SIGGRAPH 2006 Papers* (2006), SIGGRAPH '06, pp. 485–493.

[136] WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (EUROGRAPHICS '01) 20*, 3 (2001), 153–164.

[137] WALD, I., WOOP, S., BENTHIN, C., JOHNSON, G., AND ERNST, M. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (July 2014), 143:1–143:8.

[138] WALTER, B., BALA, K., KULKARNI, M., AND PINGALI, K. Fast agglomerative clustering for rendering. In *2008 IEEE Symposium on Interactive Ray Tracing* (Aug 2008), pp. 81–86.

[139] WILLIAMS, A., BARRUS, S., MORLEY, K., AND SHIRLEY, P. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses* (2005), SIGGRAPH '05.

[140] Woop, S., Brunvand, E., and Slusallak, P. Estimating performance of a ray tracing ASIC design. In *2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 7–14.

[141] Woop, S., Feng, L., Wald, I., and Benthin, C. Embree ray tracing kernels for CPUs and the Xeon Phi architecture. In *ACM SIGGRAPH 2013 Talks* (2013), SIGGRAPH '13, pp. 44:1–44:1.

[142] Woop, S., Schmittler, J., and Slusallek, P. RPU: A programmable ray processing unit for realtime ray tracing. In *ACM SIGGRAPH 2005 Papers* (2005), SIGGRAPH '05, pp. 434–444.

[143] Wulf, W. A., and McKee, S. A. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News 23*, 1 (Mar. 1995), 20–24.

[144] Wyman, C., Hargreaves, S., Shirley, P., and Barr-Brisebois, C. Introduction to DirectX raytracing. In *ACM SIGGRAPH 2018 Courses* (August 2018), SIGGRAPH '18, pp. 9:1–9:1.

[145] Ylitie, H., Karras, T., and Laine, S. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics* (2017), HPG '17, pp. 4:1–4:13.

[146] Yoon, S.-E., and Manocha, D. Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum* (2006), vol. 25, pp. 507–516.

[147] Zhang, C., and Vahid, F. Using a victim buffer in an application-specific memory hierarchy. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (2004), DATE '04, p. 10220.